Notatki z AiSD. Nr 1. 16 marca 2005

Preliminaria

IIUWr. II rok informatyki. Przygotował: Krzysztof Loryś

1 Literatura

1.1 Podstawowa

1. T.H. Cormen, C.E. Leiserson i R.L. Rivest, *Wstęp do Algorytmów*, WNT,1997 (i kolejne; późniejsze wydania są nieco zmienione).
2. A.V. Aho, J.E. Hopcroft i J.D. Ullman, *Projektowanie i Analiza Algorytmów Komputerowych*, PWN, 1983 (oraz Helion 2003).
3. L. Banachowski, K. Diks i W. Rytter, *Algorytmy i Struktury Danych*, WNT, 1996.
4. G. Brassard i P. Bratley, *Algorithmics. Theory & Practice.*, Prentice Hall, 1988.

1.2 Uzupełniająca

1. D.E. Knuth, *The art of computer programming*, vol. I-III, Addison-Wesley, 1968-1973 (polskie wydanie WNT 2002).
2. D.C Kozen, *The design and analysis of algorithms*, Springer-Verlag, 1992.
3. S. Baase i A.vav Gelder, *Computer algorithms: introduction to design and analysis*, Addison-Wesley, 2000 .
4. L. Banachowski, A. Kreczmar i W. Rytter, *Analiza algorytmów i struktur danych*, WNT, 1987.
5. L. Banachowski, A. Kreczmar i W. Rytter, *Analysis of algorithms and data structures*, Addison-Wesley, 1991.
6. S.E. Goodman i S.T. Hedetniemi, *Introduction to the design and analysis of algorithms*, McGraw-Hill, 1977.
7. M.T.Goodrich i R.Tamassia, *Data structures and algorithms in JAVA,* Wiley, 1998.
8. E.M. Reingold, J. Nievergeld i N. Deo, *Algorytmy kombinatoryczne*, PWN, 1985.
9. R.Sedgewick, *Algorytmy w C++,* Wyd. ReadMe, 1999.
10. S.S.Skiena, *The algorithm design manual,* Springer-Verlag, 1997.
11. M.M.Sysło, N.Deo i J.S.Kowalik, *Algorytmy optymalizacji dyskretnej,* PWN, 1995.
12. J.P Tremblay i P.G Sorenson, *An introduction to data structures with applications*, McGraw-Hill, 1976.
13. M.A.Weiss, *Data structures and algorithm analysis,* Benjamin Cummings, 1992.
14. D.H. Greene i D.E. Knuth, *Mathematics for the analysis of algorithms*, Birkhäuser, 1982.

2 Problemy, algorytmy, programy

Zakładam, że wszyscy znają te pojęcia. Poniżej podajemy przykłady dwóch problemów oraz różnych algorytmów rozwiązujących je.

Przykład 1 *Mnożenie liczb naturalnych.*

PROBLEM.
dane: $a, b \in \mathcal{N}$
wynik: iloczyn liczb a i b

ALGORYTM 1. a razy dodać do siebie liczbę b

ALGORYTM 2. "Pomnożyć pisemnie"

ALGORYTM 3. Mnożenie *"po rosyjsku"*

1. oblicz ciąg $a_1, a_2, ..., a_k$ taki, że $a_1 = a$, $a_k = 1$, $a_{i+1} = \lfloor \frac{a_i}{2} \rfloor$ (dla $i = 1, ..., k-1$),
2. oblicz ciąg $b_1, b_2, ..., b_k$ taki, że $b_1 = b$, $b_{i+1} = 2b_i$ (dla $i = 1, ..., k-1$),
3. oblicz $\sum_{\substack{i=1 \\ a_i \text{ nieparzyste}}}^{k} b_i$

□

UWAGA: Później poznamy jeszcze dwa inne (niebanalne) algorytmy mnożenia liczb.

Przykład 2 *Obliczanie n-tej liczby Fibonacciego.*

PROBLEM.
dane: $n \in \mathcal{N}$
wynik: wartość n-tej liczby Fibonacciego modulo stała c

ALGORYTM 1. Metoda rekurencyjna

```
fibrek(intn)
{   if (n ≤ 1) return 1;
    return (fibrek(n − 1) + fibrek(n − 2)) mod c;
}
```

ALGORYTM 2. Metoda iteracyjna

```
fibiter(intn)
{   inti, t, f0, f1;
    f0 ← f1 ← 1;
    for (i = 2; i ≤ n; i++)
       {t ← f0; f0 ← f1; f1 ← (t + f0) mod c;}
    return f1;
}
```

ALGORYTM 3. Metoda *"macierzowa"*

Korzystamy z tego, że

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f_i \\ f_{i+1} \end{bmatrix} = \begin{bmatrix} f_{i+1} \\ f_{i+2} \end{bmatrix}$$

Stąd

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n-1} \cdot \begin{bmatrix} f_0 \\ f_1 \end{bmatrix} = \begin{bmatrix} f_{n-1} \\ f_n \end{bmatrix}$$

Wystarczy więc podnieść macierz do odpowiedniej potęgi (wykonując obliczenia modulo c) a następnie wynik przemnożyć przez wektor $[1,1]^T$. □

UWAGI:

- Będziemy zajmować się tylko problemami określonymi na nieskończonej dziedzinie (zbiorze danych). Dla problemu mnożenia jest nią $\mathcal{N}^2$, a dla problemu obliczania liczb Fibonacciego $\mathcal{N}$.
- Do zapisu algorytmów będziemy stosować różne formalizmy: od języka C, poprzez pseudopascal do opisu słownego.

3 Złożoność algorytmów i problemów

Efektywność (złożoność) algorytmów można porównywać empirycznie bądź teoretycznie. Wadą pierwszej metody jest jej zależność od implementacji oraz fakt, że zwykle można przetestować tylko niewielką grupę danych. Będziemy zajmować się głównie drugą metodą. Złożoność algorytmów będziemy określać funkcją rozmiaru danych.

Przykład 3 *Przykłady określenia rozmiaru danych.*

Problem	Rozmiar danych
Wyszukiwanie elementu w ciągu	*# elementów w ciągu*
Mnożenie macierzy	*Rozmiary macierzy*
Sortowanie ciągu liczb	*# elementów w ciągu*
Przechodzenie drzewa binarnego	*# węzłów w drzewie*
Rozwiązywanie układu równań	*# równań lub # zmiennych lub obie*
Problemy grafowe	*# wierzchołków lub # krawędzi lub obie.*

□

Będzie nas interesować:

- *Złożoność czasowa* - liczba jednostek czasu potrzebnych na wykonanie algorytmu.
- *Złożoność pamięciowa* - liczba jednostek pamięci (np. komórek, bitów) potrzebnych na wykonanie algorytmu.

Jednostka czasu - czas potrzebny na wykonanie elementarnej operacji. Aby nasze rozważania były precyzyjne musimy określić model komputera. Dla nas podstawowym modelem będzie maszyna RAM (jej krótki opis zamieszczony jest na końcu notatki). Zwykle będziemy przyjmować następujące:

- *kryterium jednorodne* - koszt każdej operacji maszyny RAM jest jednostkowy.

Kryterium jednorodne jest nierealistyczne w przypadku algorytmów operujących na wielkich liczbach. W takich przypadkach będziemy posługiwać się:

- *kryterium logarytmicznym* - koszt operacji maszyny RAM jest równy sumie długości operandów.

UWAGA: Stosując kryterium logarytmiczne należy uwzględniać koszt obliczania adresu w trakcie wykonywania rozkazów stosujących adresowanie pośrednie.

Oczywiście analizując algorytmy nie będziemy ich zapisywać w języku maszyny RAM. Będzie ona jedynie naszym punktem odniesienia podczas analizy kosztów konstrukcji algorytmicznych wyższego rzędu.

Przykład 4 *Dwa algorytmy sortowania ciągu liczb.*

```
Procedure insert(T[1..n])
    for i ← 2 to n do
        x ← T[i]; j ← i − 1
        while j > 0 and x < T[j] do
            T[j + 1] ← T[j]
            j ← j − 1
        T[j + 1] ← x
```

```
Procedure select(T[1..n])
    for i ← 1 to n − 1 do
        minj ← i; minx ← T[i]
        for j ← i + 1 to n do
            if T[j] < minx then minj ← j
                                minx ← T[j]
        T[minj] ← T[i]
        T[i] ← minx
```

Idea tych algorytmów:

- *Insert*: w $i-$tej iteracji element $T[i]$ wstawiamy w odpowiednie miejsce do uporządkowanego ciągu $T[1], \ldots, T[i-1]$.
- *Select*: po $i-1$-szej iteracji elementy $T[1], \ldots, T[i-1]$ są uporządkowane i mniejsze od każdego elementu $T[i], \ldots, T[n]$; w i-tej iteracji wybieramy minimalny element spośród $T[i], \ldots, T[n]$ i wstawiamy go na pozycję $T[i]$.

Złożoność czasowa (przy kryterium jednorodnym):

- *Select* - zawsze rzędu n^2.
  Uzasadnienie: Najpierw zauważamy, że instrukcja **if** sprowadza się do wykonania stałej liczby instrukcji maszyny RAM. Podobnie jest z inicjalizacją i jednokrotną iteracją pętli **for** . Instrukcje pętli wewnętrznej wykonują się $\Theta(n^2)$ razy. Każda ich iteracja to koszt stałej liczby instrukcji maszyny RAM, tak więc w sumie koszt wykonania tych instrukcji jest $\Theta(n^2)$. Koszt pozostałych instrukcji jest mniejszy i nie wyprowadza poza $\Theta(n^2)$.

- *Insert* - dla niektórych danych rzędu n^2, dla niektórych jedynie rzędu n.
  Uzasadnienie: Koszt procedury zależy od początkowego uporządkowania tablicy T. W najgorszym przypadku element $T[i]$ wstawiany jest na początek tablicy co wymaga i operacji przesunięcia. Łatwo sprawdzić, że gdy taka sytuacja ma miejsce dla każdego $i = 2, \ldots, n$ (tj. gdy początkowo tablica uporządkowana jest malejąco), to koszt procedury wynosi $\Theta(n^2)$. Z drugiej strony, gdy początkowo tablica jest uporządkowana rosnąco, to dla każdego $i = 2, \ldots, n$ pętla **while** ma koszt stały, a więc cała procedura wykonuje się w czasie liniowym (tj. $\Theta(n)$).

□

Powyższe spostrzeżenia prowadzą nas do pojęcia złożoności najgorszego i średniego przypadku:

- *złożoność najgorszego przypadku* (inaczej złożoność pesymistyczna) - maksimum kosztu obliczeń na danych rozmiaru n.

- *złożoność średniego przypadku* (inaczej złożoność oczekiwana) - średni koszt obliczeń na danych rozmiaru n.

Uwaga: Przy obliczaniu średniej złożoności należy uwzględniać rozkład prawdopodobieństwa z jakim algorytm będzie wykonywany na poszczególnych danych (zwykle jest to bardzo trudne do ustalenia).

Powracając do naszego przykładu widzimy, że złożoność pesymistyczna obydwu algorytmów sortowania jest rzędu n^2. Jak później pokażemy obydwa algorytmy mają także taką samą złożoność w średnim przypadku.

Stając przed dylematem wyboru któregoś spośród algorytmów należy postępować bardzo rozważnie i uwzględnić szereg czynników. W przypadku powyższych algorytmów czynnikami takimi są m.in.:

- Rozkład danych. Co prawda w średnim przypadku *insert* nie jest lepszy od *select*, jednak stwierdzenie to jest prawdziwe przy założeniu jednostajnego rozkładu danych. W praktyce często mamy do czynienia z innymi rozkładami, w tym często z danymi prawie uporządkowanymi. Taka sytuacja przemawia za wyborem *insert*.

- Wielkość rekordów. Często sortujemy nie tyle same klucze, co rekordy, zawierające klucze jako jedno ze swych pól. Jeśli rekordy są duże, to należy uwzględnić fakt, że operacja przestawienia elementów jest kosztowna. Ponieważ *select* wykonuje zawsze $O(n)$ operacji

przestawienia elementów, a *insert* może ich wykonywać nawet $\Omega(n^2)$, więc taka sytuacja może przemawiać za wyborem *select*.
W takiej sytuacji można też rozważyć użyteczność wersji *insert* operującej na kopiach kluczy i wskaźnikach do rekordów. Wskaźniki te służą do przestawienia rekordów zgodnie z otrzymaną poprzez sortowanie permutacją kluczy.

- Stabilność. Czasami zależy nam, by rekordy o jednakowych kluczach pozostawały w tablicy wynikowej w takim samym względnym porządku w jakim były początkowo. O procedurach sortowania zachowujących taki porządek mówimy, że są *stabilne*. Łatwo zauważyć, że *insert* jest stabilny, a *select* nie.

- Intensywność wykorzystania algorytmu. Jeśli algorytm ma być bardzo intensywnie wykorzystywany, np. jako część składowa większego systemu, wówczas nawet drobne usprawnienia mogą prowadzić do istotnej poprawy efektywności całego systemu. W tym przypadku warto zwrócić uwagę na możliwość dokonania takich usprawnień w algorytmach jak redukcja liczby rozkazów w najbardziej wewnętrznych pętlach algorytmu, możliwość zastosowania szybkich operacji maszynowych, itp... Warto też duży nacisk położyć na porównanie empiryczne algorytmów.

Na zakończenie jeszcze uwaga o *złożoności problemów*. Definiuje się ją jako złożoność najlepszego algorytmu rozwiązującego dany problem. Zwykle jest ona dużo trudniejsza do określenia niż złożoność konkretnego algorytmu.
Załóżmy, że chcemy określić złożoność problemu $\mathcal{P}$. Skonstruowanie algorytmu $\mathcal{A}$ rozwiązującego $\mathcal{P}$ pozwala nam jedynie na sformułowanie wniosku, że złożoność $\mathcal{P}$ jest nie większa niż złożoność $\mathcal{A}$ (mówimy, że algorytm $\mathcal{A}$ wyznacza *granicę górną* na złożoność $\mathcal{P}$). Aby dokładnie wyznaczyć złożoność $\mathcal{P}$ należy ustalić jeszcze granicę dolną, a więc wykazać, że żaden algorytm nie jest w stanie rozwiązać $\mathcal{P}$ szybciej. Twierdzenia tego typu są bardzo trudne i stanowią prawdziwe wyzwanie dla naukowców. W trakcie wykładu zapoznamy się tylko z kilkoma przykładami takich twierdzeń i to tylko dla ograniczonego (w stosunku do maszyny RAM) modelu obliczeń.

4 Notacja dla rzędów funkcji

Oznaczenia:
$\mathcal{R}^*$ - zbiór nieujemnych liczb rzeczywistych,
$\mathcal{R}^+$ - zbiór dodatnich liczb rzeczywistych,
analogiczne oznaczenia dla $\mathcal{N}$.

Definicja 1 *Niech $f : \mathcal{N} \to \mathcal{R}^*$ będzie dowolną funkcją.*

- $O(f(n)) = \{t : \mathcal{N} \to \mathcal{R}^* \mid \exists_{c\in\mathcal{R}^+}\exists_{n_0\in\mathcal{N}}\forall_{n\geq n_0} \quad t(n) \leq cf(n).\}$
- $\Omega(f(n)) = \{t : \mathcal{N} \to \mathcal{R}^* \mid \exists_{c\in\mathcal{R}^+}\exists_{n_0\in\mathcal{N}}\forall_{n\geq n_0} \quad t(n) \geq cf(n).\}$
- $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$.

Uwaga: Czasami $\Omega(f(n))$ jest definiowana jako $\{t : \mathcal{N} \to \mathcal{R}^* \mid \exists_{c\in\mathcal{R}^+}\forall_{n_0\in\mathcal{N}}\exists_{n\geq n_0} \quad t(n) \geq cf(n).\}$

5 Podstawowe struktury danych

Zakładam znajomość takich struktur danych (i ich implementacji) jak: tablice, rekordy, listy, kolejki, stos, drzewa, grafy (listy incydencji, macierz sąsiedztwa),... .

6 Dodatek: Krótki opis maszyny RAM

Części składowe:

- *taśma wejściowa* - ciąg liczb całkowitych; dostęp jednokierunkowy;
- *taśma wyjściowa*
- *pamięć:* komórki adresowane liczbami naturalnymi; każda komórka może pamiętać dowolną liczbę całkowitą.
- *akumulator* - komórka o adresie 0.
- procesor.

Instrukcje:

LOAD	*argument*	STORE	*argument*
ADD	*argument*	SUB	*argument*
MULT	*argument*	DIV	*argument*
READ	*argument*	WRITE	*argument*
JUMP	*etykieta*	JGTZ	*etykieta*
JZERO	*etykieta*	HALT	

Operacje przesłania i arytmetyczne mają dwa argumenty - drugim jest akumulator. W nim umieszczany jest wynik operacji arytmetycznych.

Rodzaje argumentów:

postać	znaczenie
$=$*liczba*	stała
liczba	adres
$\star$ *liczba*	adresowanie pośrednie

7 Schemat ogólny.

Typowe zadanie rozwiązywane metodą zachłanną ma charakter optymalizacyjny. Mamy dany skończony zbiór C. Rozwiązaniami zadania są pewne podzbiory zbioru C. Znamy kryterium pozwające na porównywanie jakości rozwiązań. Chcemy znale"xć rozwiązanie, które jest optymalne względem tego kryterium.

Przykład
PROBLEM:
Dane: liczba naturalna R oraz zbiór liczb naturalnych $c_1, c_2, \ldots, c_k$;
(interpretacja: R jest kwotą, którą chcemy rozmienić, a c_i są nominałami monet).
Zadanie: rozmienić kwotę R na moźliwie najmniejszą liczbę monet (przy założeniu, źe dysponujemy nieograniczoną liczbą monet każdego nominału).

W tym przykładzie zbiorem C jest zbiór monet (zauważ, źe zbiór ten jest skończony, ponieważ możemy przyjąć, źe należy do niego nie więcej niź R monet o nominale c_i, dla każdego $i = 1, \ldots, k$). Rozwiązaniami problemu są te podzbiory zbioru C, których elementy sumują się do kwoty R. Kryterium jakości rozwiązania jest liczba jego elementów. □

Algorytm zachłanny konstruuje rozwiązanie, nazwijmy je S, stopniowo (zwykle startując od zbioru pustego). W każdym z kolejnych kroków, algorytm rozważa jeden element z C, powiedzmy x. Element ten jest umieszczany w S, jeśli algorytm uzna, źe $S \cup \{x\}$ da się rozszerzyć do rozwiązania. Niezależnie od decyzji, x jest usuwany z C. Wybór x-a dokonywany jest na podstawie lokalnego (zachłannego) kryterium optymalności.

Algorytmy zachłanne nie podejmują źadnych (lub prawie źadnych) działań sprawdzających czy konstruowany zbiór ma szansę być optymalnym rozwiązaniem. W efekcie algorytmy zachłanne są proste i szybkie, ale mogą dawać rozwiązania nieoptymalne, a nawet nie dawać rozwiązań. Dlatego waźnym zadaniem jest analiza poprawności algorytmu zachłannego. Jeśli algorytm produkuje rozwiązania nieoptymalne, warta rozważenia może być kwestia "jak dalece nieoptymalne": czasami mogą one być akceptowalnie bliskie optymalnym. Z takimi zagadnieniami zapoznamy się, gdy będziemy omawiać algorytmy aproksymacyjne.

Przykład
Strategia zachłanna dla problemu wydawania reszty może polegać na tym, by w kolejnych krokach do konstruowanego rozwiązania wstawiać monetę o największym nominale nie przekraczającym kwoty pozostałej do wydania.

Można łatwo wykazać, że dla zbioru nominałów $\{1, 2, 5, 10, 20, 50, 100\}$ taka strategia prowadzi zawsze do rozwiązania optymalnego. Natomiast dla zbioru $\{1, 2, 5, 9, 10\}$ czasami daje rozwiązania nieoptymalne: np. dla $R = 14$ znajdzie rozwiązanie $10, 2, 2$, chociaż wystarczają dwie monety.

7.1 Konstrukcja minimalnego drzewa rozpinającego

Rozważamy grafy nieskierowane $G = (V, E; c)$, gdzie V oznacza zbiór wierzchołków, E - zbiór krawędzi, a $c : E \to R_+$ jest funkcją wagową. Wagą podgrafu $G' = (V', E')$ grafu G nazywamy sumę wag krawędzi z E'.

Definicja 2 Drzewem rozpinającym *grafu* $G = (V, E; c)$ *nazywamy dowolne drzewo* $T = (V, E')$*, takie, że* $E' \subseteq E$*. Drzewo rozpinające* T *nazywamy* minimalnym*, jeśli ma minimalną wagę spośród wszystkich drzew rozpinających grafu* G*.*

PROBLEM:
Dane: graf $G = (V, E; c)$
Zadanie: Wyznaczyć minimalne drzewo rozpinające $T = (V, E')$ grafu G.

Jak łatwo zauważyć zadanie sprowadza się do wyznaczenia zbioru krawędzi drzewa rozpinającego. Wiele znanych algorytmów rozwiązujących ten problem opartych jest na strategiach zachłannych. Poniżej przedstawiamy trzy z nich.

- Strategia 1: Algorytm Kruskala
  Rozpoczynamy od pustego E'. Zbiór C jest początkowo równy E. W kolejnym kroku rozpatrujemy krawęd"x z C o minimalnej wadze. Dodajemy ją do E', o ile nie powoduje to powstania cyklu.

- Strategia 2: Algorytm Prima
  Inicjujemy E' wstawiając do niego minimalną krawęd"x spośród krawędzi incydentnych z wierzchołkiem v (v - wybierany jest arbitrarnie). Podobnie jak poprzednio zbiór C jest początkowo równy E. W kolejnym kroku rozpatrujemy minimalną krawęd"x z C incydentną z jakąś krawędzią z E' . Dodajemy ją do E', o ile drugi z jej końców nie jest incydentny z E'.

- Strategia 3: Algorytm Sollina
  Strategia ta nieco odbiega od ogólnego schematu. Zbiór E' budujemy fazami. W każdej fazie wykonujemy dwa kroki:

  - Dla każdego wierzchołka z G znajdujemy najkrótszą incydentną z nim krawęd"x; krawędzie te dołączamy do zbioru E'.
  - Tworzymy nowy graf G'. Wierzchołki w G' (nazwijmy je superwierzchołkami) odpowiadają spójnym składowym w E'. Dwa superwierzchołki S_1 i S_2 łączymy krawędzią wtedy i tylko wtedy, gdy jakiś wierzchołek z S_1 był połączony w G krawędzią z jakimś wierzchołkiem z S_2. Jako wagę tej krawędzi przyjmujemy minimalną wagę krawędzi w G pomiędzy wierzchołkami z S_1 i S_2.
    Za G przyjmujemy G' i przechodzimy do nowej fazy.

UWAGI:

- algorytm Sollina jest szczególnie przystosowany do implementacji na maszynach równoległych;
- algorytm ten działa poprawnie, gdy wszystkie krawędzie mają różne wagi (to jednak zawsze potrafimy zagwarantować).

Dowody poprawności tych algorytmów są podobne. Łatwo zauważyć, że wszystkie algorytmy znajdują drzewa rozpinające. Strategie Kruskala i Sollina gwarantują bowiem, że w każdym momencie krawędzie z E' tworzą las drzew, a strategia Prima gwarantuje, że krawędzie z E' tworzą drzewo. O ile graf G jest spójny, to po zakończeniu działania algorytmu graf (V, E') także jest spójny (w przeciwnym razie minimalna krawęd"x spośród krawędzi o końcach w różnych składowych nie byłaby dołączona przez algorytm do E' - sprzeczność?!), a więc jest drzewem rozpinającym. Minimalność wyznaczonych drzew wynika z faktu, że w każdym momencie konstrukcji zbioru E' jest on rozszerzalny do minimalnego drzewa rozpinającego.

7.2 Szeregowanie zadań

Rozważymy teraz dwa przykłady prostych problemów teorii szeregowania zadań. Obydwa dotyczą szeregowania dla pojedynczego procesora i dają się rozwiązać algorytmami zachłannymi. Pod tym względem są wyjątkowe, ponieważ większość problemów szeregowania jest NP-trudna.

7.2.1 Szeregowanie zadań dla pojedynczego procesora

SCENARIUSZ: System z jednym serwerem (procesorem) ma do obsłużenia n zleceń. Zawczasu znany jest czas obsługi każdego zlecenia. Przez *czas przebywania zlecenia w systemie* rozumiemy czas jaki upłynął od momentu zgłoszenia zlecenia systemowi do momentu zakończenia jego obsługi. Zakładamy, że wszystkie zlecenia zgłoszone zostały jednocześnie i że obsługa zleceń odbywa się bez przerwań, więc czas przebywania zlecenia w systemie jest równy sumie czasu oczekiwania na zlecenie i czasu obsługi. Za czas jaki zlecenia przebywają w systemie, system płaci karę proporcjonalną do tego czasu, dlatego naszym zadaniem jest ustawienie zleceń w kolejności minimalizującej karę.

PROBLEM:
Dane: ciąg $t_1, \ldots, t_n$ dodatnich liczb rzeczywistych;
(interpretacja: t_j - czas obsługi j-tego zadania w systemie).
Zadanie: ustawić zadania w kolejności minimalizującej wartość:
$T = \sum_{i=1}^{n}$ (*czas przebywania i-tego zadania w systemie*)

Strategia zachłanna: Zadania ustawiamy w kolejności rosnących czasów obsługi.

DOWÓD POPRAWNOŚCI: Zauważamy , że jeśli zadania realizowane są w kolejności zadanej permutacją $\pi = (i_1, i_2, \ldots, i_n)$ liczb $(1, 2, \ldots, n)$, to związany z tym koszt wynosi:

$$T(\pi) = t_{i_1} + (t_{i_1} + t_{i_2}) + \ldots + (t_{i_1} + \ldots + t_{i_n}) = \sum_{k=1}^{n} (n - k + 1) t_{i_k}$$

Załóżmy, że π jest optymalnym porządkiem oraz że istnieją x, y takie, że $x < y$ oraz $t_{i_x} > t_{i_y}$. Zamieniając i_x oraz i_y miejscami otrzymujemy nowy porządek π' o koszcie:

$$T(\pi') = (n-x+1)t_{i_y} + (n-y+1)t_{i_x} + \sum_{k=1,k\neq x,y}^{n} (n-k+1)t_{i_k}.$$

Nowy porządek jest lepszy, ponieważ

$$T(\pi) - T(\pi') = (n-x+1)(t_{i_x} - t_{i_y}) + (n-y+1)(t_{i_y} - t_{i_x}) = (y-x)(t_{i_x} - t_{i_y}) > 0$$

co jest sprzeczne z założeniem optymalności π. □

7.2.2 Szeregowanie z terminami

Scenariusz: System z jednym procesorem ma do wykonania n zadań. Każde z nich wymaga jednej jednostki czasu procesora. Dla każdego zadania znany jest zysk jaki otrzyma system za jego wykonanie oraz termin. Za wykonanie zadania po terminie system nie otrzymuje żadnego zysku. Naszym celem jest ustawienie zadań w kolejności maksymalizującej zysk.

Definicja 3 *Ciąg zadań $\langle i_1, i_2 \dots i_n \rangle$ taki, że $\forall_{k=1\dots n}\ k \leq d_{i_k}$ nazywamy* wykonalnym. *Zbiór zadań jest* wykonalny, *jeśli wszystkie jego elementy można ustawić w ciąg wykonalny.*

Problem:
Dane: ciąg $d_1, \dots, d_n$ liczb naturalnych oraz
ciąg $g_1, \dots, g_n$ dodatnich liczb rzeczywistych;
(Interpretacja: d_i-termin dla i-tego zadania; g_i-zysk za i-te zadanie).
Zadanie: znale"xć wykonalny podzbiór zadań $S \subseteq \{1, \dots, n\}$ maksymalizujący sumę $\sum_{i \in S} g_i$.

Strategia zachłanna: Startując od zbioru pustego konstruujemy wykonalny zbiór zadań, na każdym kroku dodając do niego zadanie o największym g_i spośród zadań jeszcze nie rozważonych (pod warunkiem, że zbiór pozostaje wykonalny).

Dowód poprawności (szkic): Załóżmy, że nasz algorytm wybrał zbiór zadań I, podczas gdy istnieje zbiór optymalny $J \neq I$. Pokażemy, że dla obydwu zbiorów zysk za wykonanie zadań jest taki sam. Niech π_I, π_J-wykonalne ciągi zadań z I i J. Dowód przebiega w dwóch etapach:

1. Wykonując przestawienia otrzymujemy ciągi π'_I oraz π'_J takie, że wszystkie wspólne zadania (tj. z $I \cap J$) wykonują się w tym samym czasie.
2. Pokazujemy, że w pozostałych jednostkach czasowych π'_I oraz π'_J mają zaplanowane wykonanie zadań o tym samym zysku.

Ad.1. Niech $a \in I \cap J$ będzie zadaniem umieszczonym na różnych pozycjach w π_I oraz π_J. Niech będą to pozycje odpowiednio i oraz j. Bez zmniejszenia ogólności możemy założyć, że $i < j$. Niech ponadto $\forall_{k<i}$ na pozycji k ciągi π_I oraz π_J mają albo zadania spoza $I \cap J$ albo takie samo zadanie z $I \cap J$.
Zadanie a w ciągu π_I możemy zamienić miejscami z zadaniem znajdującym się na pozycji j, nazwijmy to zadanie b. Otrzymamy ciąg wykonalny, gdyż:

- $d_a \geq j$, ponieważ a znajduje się na pozycji j w π_J,
- $d_b > i$, ponieważ b znajduje się na pozycji j w π_I.

Liczba zadań z $I \cap J$ rozmieszczonych na różnych pozycjach w π_J i nowym ciągu π_I zmniejszyła się o co najmniej 1. Iterując postępowanie otrzymujemy tezę.

Ad.2. Rozważmy dowolną pozycję i, na której różnią się π'_I oraz π'_J.
Zauważamy, że zarówno π'_I jak i π'_J mają na tej pozycji umieszczone jakieś zadanie, nazwijmy je a i b odpowiednio. Gdyby bowiem π'_J miało tę pozycję wolną, to moglibyśmy umieścić na niej a otrzymując ciąg wykonalny dla $J \cup \{a\}$, co przeczy optymalności J. Z drugiej strony, gdyby π'_I miało tę pozycję wolną to algorytm zachłanny umieściłby b w rozwiązaniu.

Wystarczy teraz pokazać, że $g_a = g_b$:

- gdyby $g_a < g_b$, to algorytm zachłanny rozpatrywałby wcześniej b niż a; ponieważ zbiór $I \setminus \{a\} \cap \{b\}$ jest wykonalny (a więc wykonalny jest także i ten jego podzbiór, który był skonstruowany w momencie rozpatrywania b), więc b zostałoby dołączone do rozwiązania.
- gdyby $g_a > g_b$, to $J \setminus \{b\} \cap \{a\}$ dawałby większy zysk niż J?!

□

Pozostaje problem: jak można ustalać czy dany zbiór J złożony z k zadań jest wykonalny (oczywiście sprawdzanie wszystkich $k!$ ciągów nie jest najlepszym pomysłem). Poniższy prosty lemat mówi, że wystarczy sprawdzać wykonalność tylko jednego ciągu.

Lemat 1 *Niech J będzie zbiorem k zadań i niech $\sigma = (s_1, s_2 \ldots s_k)$ będzie permutacją tych zadań taką, że $d_{s_1} \leq d_{s_2} \leq \ldots \leq d_{s_k}$. Wówczas J jest wykonalny iff σ jest wykonalny.*

7.2.3 Prosta implementacja

Zakładamy, że zadania ułożone są według malejących zysków, tj. $g_1 \geq g_2 \geq \ldots \geq g_n$.

Prosta implementacja polega na pamiętaniu skonstruowanego fragmentu wykonywalnego ciągu zadań w początkowym fragmencie tablicy. Zadania umieszczane są tam według rosnących wartości terminów w sposób podobny jak w procedurze sortowania przez wstawianie.

Fakt 1 *Taka implementacja działa w czasie $\Omega(n^2)$.*

7.2.4 Szybsza implementacja

Kluczem do szybszej implementacji jest następujący lemat.

Lemat 2 *Zbiór zadań J jest wykonalny iff następująca procedura ustawia wszystkie zadania z J w ciąg wykonalny:*

> *$\forall_{i \in J}$ ustaw i-te zadanie na pozycji t, gdzie t jest największą liczbą całkowitą, taką że $0 < t \leq \min(n, d_i)$ i na pozycji t nie ustawiono jeszcze żadnego zadania.*

Efektywna realizacja tej procedury używa struktur danych do pamiętania zbiorów rozłącznych. Poznamy je, gdy będziemy omawiać problem UNION-FIND. Uzyskany czas działania algorytmu będzie bliski liniowego.

UWAGA: Trzeba jednak pamiętać, że jeśli zadania nie są, jak to założyliśmy, wstępnie uporządkowane według wartości g_i, to czas działania algorytmu zostanie zdominowany przez sortowania.

Notatki z AiSD. Nr 3. 16 marca 2005

Metoda Dziel i Zwyciężaj.

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

8 Schemat ogólny.

Algorytmy skonstruowane metodą "dziel i zwyciężaj" składają się z trzech zasadniczych kroków:

1. transformacja danych x na dane $x_1, \ldots, x_k$ o mniejszym rozmiarze;
2. rozwiązanie problemu dla danych x_i $(i = 1, \ldots, k)$;
3. obliczenie rozwiązania dla danych x na podstawie wyników otrzymanych w punkcie 2.

W naturalny sposób implementowane są jako procedury rekurencyjne:

```
function DiZ(x)
    if x jest małe lub proste then return AdHoc(x)
    1. przekształć x na x_1, ..., x_k o mniejszym rozmiarze niż x
    2. for i ← 1 to k do y_i ← DiZ(x_i)
    3. na podstawie y_1 ... y_k oblicz rozwiązanie y dla x
    return y
```

gdzie *AdHoc* jest algorytmem używanym do rozwiązania problemu dla "łatwych"danych.

9 Ważne równanie rekurencyjne.

Twierdzenie 1 *Niech $a, b, c \in \mathcal{N}$. Rozwiązaniem równania rekurencyjnego*

$$T(n) = \begin{cases} b & dla\ n = 1 \\ aT(n/c) + bn & dla\ n > 1 \end{cases}$$

dla n będących potęgą liczby c jest

$$T(n) = \begin{cases} \Theta(n) & jeżeli\ a < c, \\ \Theta(n \log n) & jeżeli\ a = c, \\ \Theta(n^{\log_c a}) & jeżeli\ a > c \end{cases}$$

Dowód. Niech: $n = c^k$, czyli $k = \log_c n$. Stosując metodę podstawiania otrzymujemy:

$$T(n) = a^k bn/c^k + a^{k-1} bn/c^{k-1} + \ldots + abn/c + bn = bn \sum_{i=0}^{k} \left(\frac{a}{c}\right)^i.$$

Rozważamy 3 przypadki:

1. $a < c$
   Wówczas $\frac{a}{c} < 1$, więc szereg $\sum_{i=0}^{k} \left(\frac{a}{c}\right)^i$ jest zbieżny do pewnego $m \in \mathcal{R}^+$. Stąd $T(n) = bmn$.

2. $a = c$
   Wówczas $\frac{a}{c} = 1$, więc $\sum_{i=0}^{k} \left(\frac{a}{c}\right)^i = k + 1 = \Theta(\log_c n)$. Stąd $T(n) = \Theta(n \log_c n)$.

3. $a > c$
   Wówczas $\frac{a}{c} > 1$, więc:

$$T(n) = bc^k \sum_{i=0}^{k} \left(\frac{a}{c}\right) = bc^k \frac{\left(\frac{a}{c}\right)^{k+1} - 1}{\left(\frac{a}{c}\right) - 1} =$$

$$b\frac{a^{k+1} - c^{k+1}}{a - c} = \frac{ba}{a - c} a^{\log_c n} - \frac{cb}{a - c} n = \frac{ba}{a - c} a^{\log_c n} - O(n).$$

   Ponieważ n jest $O(a^{\log_c n}) = O(n^{\log_c a})$, więc $T(n) = \Theta(a^{\log_c n})$.

□

Interpretacja: Twierdzenie określa złożoność algorytmów opartych na strategii dziel i zwyciężaj, które:

- redukują problem dla danych o rozmiarze n do rozwiązania tego problemu dla a zestawów danych, każdy o rozmiarze n/c.
- wykonują kroki 1 i 3 schematu ogólnego w czasie liniowym.

10 Przykłady.

10.1 Sortowanie

Nasze przykłady rozpoczniemy od zaprezentowania dwóch strategii Dziel i Zwyciężaj dla problemu sortowania.

Problem:

Dane:	tablica $T[1..n]$ elementów z uporządkowanego liniowo uniwersum
Zadanie:	uporządkować T

10.1.1 Strategia 1: Sortowanie przez scalanie.

Strategia ta oparta jest na tym, że dwa uporządkowane ciągi potrafimy szybko (w czasie liniowym) scalić w jeden ciąg. Aby posortować tablicę wystarczy więc podzielić ją na dwie części, niezależnie posortować każdą z części a następnie scalić je.

```
procedure mergesort(T[1..n])
  if n jest małe then insert(T)
  else
       X[1 .. ⌈n/2⌉] ← T[1 .. ⌈n/2⌉]
       Y[1 .. ⌊n/2⌋] ← T[1 + ⌈n/2⌉ .. n]
       mergesort(X); mergesort(Y)
       T ← merge(X, Y)
```

Czas działania algorytmu wyraża się równaniem $t(n) = t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + \Theta(n)$, którego rozwiązaniem jest $t(n) = \Theta(n \log n)$. Jak pó"xniej pokażemy jest to asymptotycznie optymalny czas działania algorytmów sortowania.

Mankamentem tego algorytmu jest fakt wykorzystywania dodatkowych tablic (poza tablicą wejściową) podczas scalania ciągów. Niestety nie jest znany sposób usunięcia tej wady. Co prawda znane są metody "scalania w miejscu" w czasie liniowym, lecz są one bardzo skomplikowane, co sprawia, że stałe w funkcjach liniowych ograniczających czas działania są nieakceptowalnie wielkie.

Przy okazji prezentacji tego algorytmu chcemy zwrócić uwagę na niezwykle ważny, a często zaniedbywany, aspekt implementacji algorymów typu Dziel i Zwyciężaj: staranne dobranie progu na rozmiar danych, poniżej którego nie opłaca się stosować algorytmu rekurencyjnie. Przykładowo powyżej zastosowaliśmy dla małych danych algorytm *insert*. Może on wymagać czasu kwadratowego, ale jest bardzo prosty i łatwy w implementacji, dzięki czemu w praktyce jest on dla małych danych szybszy od rekurencyjnej implementacji sortowania przez scalanie. Teoretyczne wyliczenie wartości progu jest zwykle trudne i zawodne. Jego wartość zależy bowiem także od efektywności implementacji a nawet od typu maszyny, na którym program będzie wykonywany. Dlatego, jeśli zależy nam na optymalnym "dostrojeniu" programu (na przykład z tego powodu, że jest on bardzo często wykonywaną procedurą, ważącą na efektywności całego systemu), powinniśmy wyznaczyć ten próg poprzez starannie dobrane eksperymenty.

10.1.2 Strategia 2: Quicksort

Najistotniejszym krokiem algorytmu sortowania przez scalanie jest krok 3 (łączenie wyników podproblemów). Natomiast krok 1 jest trywialny i sprowadza się do wyliczenia indeksu środka tablicy (ze względów technicznych połączyliśmy go powyżej z kopiowaniem elementów do tablic roboczych).

W algorytmie *Quicksort* sytuacja jest odwrotna: istotnym krokiem jest krok 1. Polega on na podziale elementów tablicy na dwa ciągi, takie, że każdy element pierwszego z nich jest nie mniejszy od każdego elementu drugiego z nich. Jeśli teraz każdy z tych ciągów zostanie niezależnie posortowany, to krok 3 staje się zbyteczny.

```
procedure Quicksort(T[1..n])
  if n jest małe then insert(T)
  else
       wybierz element dzielący x
       (* niech k równa się liczbie elementów tablicy T nie większych od x*)
       przestaw elementy tablicy T tak, że ∀_{i≤k} T[i] ≤ x
       Quicksort(T[1..k]); Quicksort(T[(k+1)..n]);
```

Analizie złożoności algorytmu *Quicksort* poświęcimy oddzielny wykład. Wówczas omówimy także sposoby implementacji kroku 1.

10.2 Mnożenie bardzo dużych liczb.

PROBLEM:
Dane: liczby naturalne a i b
komentarz: liczby a i b są długie.
Wynik: iloczyn $a \cdot b$

Dla prostoty opisu przyjmijmy, że obydwie liczby mają tę samą długość (równą $n = 2^k$). Narzucający się algorytm oparty na strategii Dziel i Zwyciężaj polega na podziale n-bitowych czynników na części $n/2$-bitowe, a następnie odpowiednim wymnożeniu tych części.

Niech $a = a_1 \cdot 2^s + a_0$ i $b = b_1 \cdot 2^s + b_0$, gdzie $s = n/2;\ 0 \leq a_1, a_0, b_1, b_0 < 2^s$. Iloczyn $a \cdot b$ możemy teraz zapisać jako

$$ab = c_2 \cdot 2^{2s} + c_1 \cdot 2^s + c_0,$$

gdzie $c_2 = a_1 b_1;\ c_1 = a_0 b_1 + a_1 b_0;\ c_0 = a_0 b_0$.

Jak widać jedno mnożenie liczb n-bitowych można zredukować do czterech mnożeń liczb $n/2$-bitowych, dwóch mnożeń przez potęgę liczby 2 i trzech dodawań. Zarówno dodawania jak i mnożenia przez potęgę liczby 2 można wykonać w czasie liniowym. Taka redukcja nie prowadzi jednak do szybszego algorytmu - czas działania wyraża się wzorem $T(n) = 4T(n/2) + \Theta(n)$, którego rozwiązaniem jest $T(n) = \Theta(n^2)$. Aby uzyskać szybszy algorytm musimy potrafić obliczać współczynniki c_2, c_1, c_0 przy użyciu trzech mnożeń liczb $n/2$-bitowych. Uzyskujemy to przez zastąpienie dwóch mnożeń podczas obliczania c_1 jednym mnożeniem i dwoma odejmowaniami:

$$c_1 = (a_1 + a_0)(b_1 + b_0) - c_0 - c_2.$$

```
multiply(a, b)
    n ← max(|a|, |b|)    (* |x| oznacza długość liczby x *)
    if n jest małe  then pomnóż a i b klasycznym algorytmem
                         return  obliczony iloczyn
    p ← ⌊n/2⌋
    a_1 ← ⌊a/2^p⌋; a_0 ← a  mod  2^p
    b_1 ← ⌊b/2^p⌋; b_0 ← b  mod  2^p
    z ← multiply(a_0, b_0)
    y ← multiply(a_1 + a_0, b_1 + b_0)
    x ← multiply(a_1, b_1);
    return 2^{2p}x + 2^p(y − x − z) + z
```

Fakt 2 *Złożoność czasowa powyższego algorytmu wynosi* $O(n^{\log 3})$.

Dowód: (Zakładamy, że a i b są liczbami n - bitowymi i n jest potęgą liczby 2.)
Wystarczy pokazać, że czas działania algorytmu wyraża się wzorem:

$$T(n) = \begin{cases} k & \text{dla } n = 1 \\ 3T(n/2) + \Theta(n) & \text{dla } n > 1 \end{cases}$$

Jedyną wątpliwość może budzić fakt, że wskutek przeniesienia liczby $a_1 + a_0$ i $b_1 + b_0$ mogą być $(n/2) + 1$-bitowe. W takiej sytuacji $a_1 + a_0$ zapisujemy w postaci $a'2^{n/2} + a''$, a $b_1 + b_0$ w postaci $b'2^{n/2} + b''$, gdzie a' i b' są bitami z pozycji $(n/2) + 1$ liczb a i b, a a'' i b'' są złożone z pozostałych bitów. Obliczenie y możemy teraz przedstawić jako:

$$(a_1 + a_0)(b_1 + b_0) = a'b'2^n + (a'b'' + a''b')2^{n/2} + a''b''$$

Jedynym składnikiem wymagającym rekurencyjnego wywołania jest $a''b''$ (obydwa czynniki są $n/2$-bitowe). Pozostałe mnożenia wykonujemy w czasie $O(n)$, ponieważ jednym z czynników jest pojedynczy bit (a' lub b') lub potęga liczby 2.
Tak więc przypadek, gdy podczas obliczania y występuje przeniesienie przy dodawaniu, zwiększa złożoność jedynie o pewną stałą, nie zmieniając klasy złożoności algorytmu. □

10.2.1 Podział na więcej części

Pokażemy teraz, że powyższą metodę można uogólnić. Niech $k \in \mathcal{N}$ będzie dowolną stałą. Liczby a i b przedstawiamy jako

$$a = \sum_{i=0}^{k-1} a_i \cdot 2^{in/k} \qquad b = \sum_{i=0}^{k-1} b_i \cdot 2^{in/k}$$

gdzie wszystkie a_i oraz b_i są liczbami co najwyżej n/k-bitowymi. Naszym zadaniem jest policzenie liczb $c_0, c_1, \ldots, c_{2k}$ takich, że dla $j = 0, \ldots, 2k$:

$$c_j = \sum_{r=0}^{j} a_r b_{j-r}.$$

Niech liczby $w_1, \ldots, w_{2k+1}$ będą zdefiniowane w następujący sposób:

$$w_t = (\sum_{i=0}^{k-1} a_i t^i) \cdot (\sum_{i=0}^{k-1} b_i t^i).$$

Łatwo sprawdzić, że $w_t = \sum_{j=0}^{2k} c_j t^j$. Otrzymaliśmy więc układ $2k+1$ równań z $2k+1$ niewiadomymi $c_0, c_1, \ldots, c_{2k}$. Fakt ten możemy zapisać jako

$$U \cdot [c_0, c_1, \ldots, c_{2k-1}, c_{2k}]^T = [w_1, w_2, \ldots, w_{2k}, w_{2k+1}]^T,$$

gdzie elementy macierzy $U = u_{ij}$ są równe $u_{ij} = i^j$ (dla $i = 1, \ldots, 2k+1$ oraz $j = 0, \ldots, 2k$). Ponieważ U jest macierzą nieosobliwą (jest macierzą Vandermonde'a), istnieje rozwiązanie powyższego układu. Jeśli $w_1, \ldots, w_{2k+1}$ potraktujemy jako wartości symboliczne, to rozwiązując ten układ wyrazimy c_i jako kombinacje liniowe tych wartości. To stanowi podstawę dla następującego algorytmu:

1. Oblicz rekurencyjnie wartości $w_1, \ldots, w_{2k+1}$.
2. Oblicz wartości $c_0, \ldots, c_{2k}$.
3. **return** $\sum_{i=0}^{2k} c_i 2^{in/k}$.

Fakt 3 *Powyższy algorytm działa w czasie* $\Theta(n^{\log_k(2k+1)})$.

Pomijamy formalny dowód tego faktu. Wynika on z tego, że w kroku 1 wywołujemy $2k+1$ razy rekurencyjnie funkcję dla danych o rozmiarze n/k oraz z tego, że kroki 2 i 3 wykonują się w czasie liniowym.

Jakkolwiek zwiększając k możemy z wykładnikiem nad n dowolnie blisko przybliżyć się do 1, to jednak zauważmy, że otrzymane algorytmy mają znaczenie czysto teoretyczne. Stałe występujące w kombinacjach obliczanych w punkcie 2 już dla niewielkich wartości k są bardzo duże i sprawiają, że algorytm działa szybciej od klasycznego mnożenia pisemnego dopiero dla danych o astronomicznie wielkich rozmiarach.

10.3 Równoczesne znajdowanie minimum i maksimum w zbiorze.

PROBLEM: Minmax
Dane: zbiór $S = \{a_1, a_2, \ldots, a_n\}$ [1]
Wynik: $min\{a_i \mid i = 1, \ldots, n\}$ $max\{a_i \mid i = 1, \ldots, n\}$

KOMENTARZ: Ograniczamy się do klasy algorytmów, które danych wejściowych używają jedynie w operacjach porównania. Interesują nas dwa zagadnienia:

- znalezienie algorytmu z tej klasy, który rozwiązuje problem Minmax, używając jak najmniejszej liczby porównań.
- dokładne wyznaczenie dolnej granicy na liczbę porównań.

[1] Termin "zbiór" jest użyty tu w dość swobodnym znaczeniu. W zasadzie S jest multizbiorem - elementy mogą się w nim powtarzać. O ile jednak nie będzie to "xródłem dwuznaczności, w przyszłości termin "zbiór" będziemy stosować także w odniesieniu do multizbiorów.

Proste podejście do tego problemu polega na tym, by szukane liczby znale"xć niezależnie, np. najpierw minimum a potem maksimum. Takie rozwiązanie wymaga $2n-2$ porównań. Jego nieoptymalność wynika z tego, źe algorytm podczas szukania maksimum nie wykorzystuje w żaden sposób informacji jakie nabył o elementach zbioru S w czasie wyszukiwania minimum. W szczególności w trakcie szukania maksimum będą brały udział w porównaniach te elementy S-a, które podczas szukania minimum były porównywane z większymi od siebie elementami, a więc nie mogą być maksimum. To spostrzeźenie prowadzi do następującego algorytmu:

```
MinMax1(S)
    S_m ← S_M ← ∅
    for i = 1 to n div 2 do
        porównaj a_i z a_{n-i+1}; mniejszą z tych liczb wstaw do zbioru S_m, a większą - do zbioru S_M
    m ← min{a | a ∈ S_m}
    M ← max{a | a ∈ S_M}
    if n parzyste then return (m, M)
                  else return (min(m, a_⌈n/2⌉), max(M, a_⌈n/2⌉))
```

Fakt 4 *Algorytm MinMax1 wykonuje* $\lceil \frac{3}{2}n-2 \rceil$ *porównań na elementach zbioru* S.

Dowód. Niech $n = 2m$. W pętli **for** wykonywanych jest m porównań, a w dwóch następnych wierszach po $m-1$ porównań. W sumie mamy $3m-2 = \lceil \frac{3}{2}n-2 \rceil$ porównań.

Gdy $n = 2m+1$, algorytm najpierw znajduje minimum i maksimum w zbiorze $S \setminus \{a_{\lceil n/2 \rceil}\}$. Zbiór ten ma $2m$ elementów, a więc liczba wykonanych porównań wynosi $3m-2$. Ostatnia instrukcja wymaga dwóch porównań, co w sumie daje $3m$ porównań. Jak łatwo sprawdzić $3m = 3(n-1)/2 = \lceil 3(n-1)/2 - 1/2 \rceil = \lceil \frac{3}{2}n-2 \rceil$. □

Inne podejście polega na zastosowaniu strategii Dziel i Zwyciężaj: zbiór S dzielimy na dwie części, w każdej części znajdujemy minimum i maksimum, jako wynik dajemy mniejsze z minimów i większe z maksimów. Poniższa, niestaranna implementacja tego pomysłu nie osiąga jednak liczby porównań algorytmu $MinMax1$ i powinna stanowić ostrzeżenie przed niefrasobliwym implementowaniem niedopracowanych algorytmów. Poprawienie tej implementacji pozostawiamy jako zadanie na ćwiczenia.

```
Procedure MinMax2(S)
  if |S|= 1 then return (a_1, a_1)
  else
    if |S|= 2 then return (max(a_1, a_2), min(a_1, a_2))
    else
      podziel S na dwa równoliczne (z dokładnością do jednego elementu) podzbiory S_1, S_2
      (max1, min1) ← MinMax2(S_1)
      (max2, min2) ← MinMax2(S_2)
      return (max(max1, max2), min(min1, min2))
```

10.3.1 Granica dolna.

Algorytm *MinMax1* wyznacza górną granicę na złożoność problemu jednoczesnego znajdowania minimum i maksimum. Teraz pokażemy, że ta granica jest także granicą dolną, a więc dokładnie wyznaczymy złożoność problemu.

Twierdzenie 2 *Każdy algorytm rozwiązujący powyższy problem (i używający elementów zbioru S jedynie w porównaniach) wykonuje co najmniej $\lceil \frac{3}{2}n - 2 \rceil$ porównań.*

Dowód: Rozważmy następującą grę między algorytmem a złośliwym adwersarzem:

- Sytuacja początkowa: adwersarz twierdzi, że zna trudny dla algorytmu zbiór S, tj. taki, dla którego wskazanie przez algorytm minimum i maksimum będzie wymagało wykonania co najmniej $\lceil \frac{3}{2}n - 2 \rceil$ porównań. Algorytm nie zna S; wie tylko, że liczy on n elementów.
- Cel gry
  - algorytmu: wskazanie indeksów elementów minimalnego i maksymalnego w zbiorze S przy użyciu mniej niż $\lceil \frac{3}{2}n - 2 \rceil$ porównań;
  - adwersarza: zmuszenie algorytmu do zadania co najmniej $\lceil \frac{3}{2}n - 2 \rceil$ porównań.
- Ruchy
  - algorytmu: pytanie o porównanie dwóch elementów ze zbioru S;
  - adwersarza: odpowied"x na to pytanie.
- Koniec gry następuje, gdy algorytm wskaże minimum i maksimum w S. Wówczas adwersarz ujawnia zbiór S.

Tezę twierdzenia udowodnimy, jeśli pokażemy, że adwersarz zawsze, niezależnie od algorytmu, posiada strategię wygrywającą.

Strategia dla adwersarza:

- W trakcie gry adwersarz dzieli S na 4 rozłączne zbiory:

  $A = \{i \mid a_i$ jeszcze nie był porównywany $\}$,
  $B = \{i \mid a_i$ wygrał juź jakieś porównanie i nie przegrał żadnego $\}$,
  $C = \{i \mid a_i$ przegrał juz jakieś porównanie i nie wygrał żadnego$\}$,
  $D = \{i \mid a_i$ wygrał juź jakieś porównanie i jakieś juź przegrał $\}$.

  Początkowo oczywiście $|A| = n$ oraz $|B| = |C| = |D| = 0$.

- Adwersarz rozpoczyna grę z dowolnymi kandydatami na wartości elementów a_i. W trakcie gry będzie, w razie konieczności, modyfikował te wartości, tak by spełniony był warunek

  $$(*) \qquad \forall_{a\in A}\forall_{b\in B}\forall_{c\in C}\forall_{d\in D} \;\; b > d > c \text{ oraz } b > a > c.$$

  Zauważ, że wystarczy w tym celu zwiększać wartości elementów o indeksach z B i zmniejszać wartości elementów o indeksach z C. Takie zmiany są bezpieczne dla adwersarza, ponieważ pozostawiają prawdziwymi jego odpowiedzi na dotychczasowe pytania.

Fakt 5 *Powyższa strategia adwesarza jest zawsze wygrywająca.*

Dowód faktu: W trakcie gry wszystkie elementy przechodzą ze zbioru A do B lub C, a dopiero stąd do zbioru D. Ponadto dla danych spełnających (*):

- jedno porównanie może usunąć co najwyżej dwa elementy ze zbioru A,
- dodanie jednego elementu do zbioru D wymaga jednego porównania,
- porównania, w których bierze udział element z A, nie zwiększają mocy zbioru D.

Dopóki A jest niepusty lub któryś ze zbiorów B lub C zawiera więcej niż jeden element algorytm nie może udzielić poprawnej odpowiedzi. Na opróżnienie zbioru A algorytm potrzebuje co najmniej $\lceil n/2 \rceil$ porównań. Następnych $n-2$ porównań potrzebnych jest na przesłanie wszystkich, poza dwoma, elementów do zbioru D. $\square$ (faktu i twierdzenia)

11 Dodatek

Poniższa tabela pokazuje w jaki sposób zmieniają się liczności zbiorów po wykonaniu różnych typów porównań (przy założeniu warunku (*)). Typ XY oznacza, iż porównywany jest element zbioru X z elementem zbioru Y. Mała litera x oznacza element zbioru X.

Typ porównania	$\lvert A\rvert$	$\lvert B\rvert$	$\lvert C\rvert$	$\lvert D\rvert$	warunek
AA	-2	+1	+1	bz	
AB	-1	bz	+1	bz	$a < b$
AC	-1	+1	bz	bz	$a > c$
AD	-1	+1	bz	bz	$a > d$
	-1	bz	+1	bz	$a < d$
BB	bz	-1	bz	+1	
BC	bz	bz	bz	bz	$b > c$
BD	bz	bz	bz	bz	$b > d$
CC	bz	bz	-1	+1	
CD	bz	bz	bz	bz	$c < d$
DD	bz	bz	bz	bz	

Notatki z AiSD. Nr 4. 16 marca 2005

Metoda Dziel i Zwyciężaj (cd.)

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

12 Dalsze przykłady

12.1 Sieci przełączników

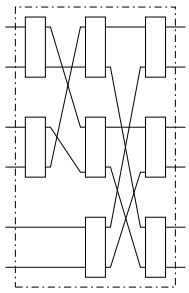
Przełącznikiem dwustanowym nazywamy urządzenie o dwóch portach wejściowych i dwóch portach wyjściowych, które:

- w stanie 1 przesyła dane z wejścia i na wyjście i (dla $i = 0, 1$),
- w stanie 2 przesyła dane z wejścia i na wyjście $(i + 1) \bmod 2$ (dla $i = 0, 1$).

Stan 1: "na wprost" Stan 2: "na ukos"

Rysunek 1: *Przełącznik dwustanowy*

Łącząc ze sobą przełączniki otrzymujemy *sieci przełączników*, które poprzez różne ustawienia przełączników mogą realizować różne permutacje danych.



Rysunek 2: *Przykład sieci przełączników o 6 wejściach*

Problem:
Dane: liczba naturalna n.
Zadanie: skonstruować sieć $Perm_n$ przełączników realizującą wszystkie permutacje n elementów.

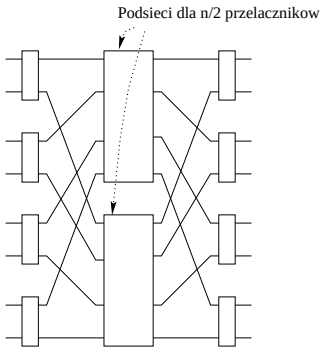
Kryteriami określającymi jakość sieci są:

- liczba przełączników;
- głębokość sieci, tj. długość maksymalnej drogi od portu wejściowego do portu wyjściowego. Długość ta mierzona jest liczbą przełączników znajdujących się na drodze (oczywiście połączenia między przełącznikami są jednokierunkowe).

12.1.1 Konstrukcja

Dla prostoty ograniczymy się do konstrukcji sieci dla n będącego potęgą liczby 2.

Konstrukcja oparta jest za zasadzie Dziel i Zwyciężaj i sprowadza się do sprytnego rozesłania danych wejściowych do dwóch (zbudowanych rekurencyjnie) egzemplarzy sieci o $n/2$ wejściach, a następnie na umiejętnym połączeniu portów wyjściowych tych podsieci. Istota tej konstrukcji przedstawiona jest na poniższym rysunku.



Rysunek 3: *Konstrukcja sieci dla* $n = 8$

Porty wejściowe sieci połączone przełącznikiem w pierwszej warstwie oraz porty wyjściowe połączone przełącznikiem w ostatniej warstwie będziemy nazywać portami *sąsiednimi*.

Fakt 6 *Tak skonstruowana sieć ma głębokość* $2\log n - 1$ *i zawiera* $\Theta(n \log n)$ *przełączników.*

Dowód: Głębokość sieci wyraża się równaniem

$$G(2^k) = \begin{cases} 1 & \text{dla } k = 1 \\ G(2^{k-1}) + 2 & \text{dla } k > 1 \end{cases}$$

a liczba przełączników - równaniem:

$$P(2^k) = \begin{cases} 1 & \text{dla } k = 1 \\ 2P(2^{k-1}) + 2^k & \text{dla } k > 1 \end{cases}$$

□

12.1.2 Poprawność konstrukcji

Niech π będzie dowolną permutacją n-elementową. Pokażemy, że istnieje ustawienie przełączników sieci realizujące π, tj. takie, że dane z i-tego portu sieci zostaną przesłane na $\pi(i)$-ty port wyjściowy (dla $i = 1, \ldots, n$). Istnienie takiego ustawienia będzie konsekwencją istnienia odpowiedniego dwukolorowania wierzchołków w następującym grafie $G_\pi = (V, E)$.

- Zbiór $V = V_I \cup V_O \cup V_M$ składa się z:
  - n wierzchołków (podzbiór V_I) odpowiadających portom wejściowym sieci;
  - n wierzchołków (podzbiór V_O) odpowiadających przełącznikom z ostatniej warstwy sieci (po dwa wierzchołki na każdy przełącznik);
  - n wierzchołków (podzbiór V_M) dodanych ze względów technicznych.
- Wszystkie wierzchołki z V etykietujemy:
  - wierzchołek z V_I odpowiadający i-temu portowi wejściowemu otrzymuje etykietę i;
  - wierzchołki z V_O, z pary odpowiadającej j-temu przełącznikowi ostatniej warstwy, otrzymują etykiety i" i k", takie, że $2j - 1 = \pi(i)$ oraz $2j = \pi(k)$ (innymi słowy na porty wyjściowe j-tego przełącznika mają być wysłane wartości z i-tego oraz k-tego portu wejściowego sieci);
  - wierzchołki z V_M otrzymują w dowolny sposób różne etykiety ze zbioru $\{1', 2', \ldots, n'\}$.
- Zbiór $E = E_I \cup E_O \cup E_M$ składa się z:
  - $n/2$ krawędzi (podzbiór E_I) łączących wierzchołki o etykietach $2i - 1$ i $2i$, a więc takie, które odpowiadają sąsiednim portom wejściowym;
  - $n/2$ krawędzi (podzbiór E_O) łączących wierzchołki odpowiadające temu samemu przełącznikowi ostatniej warstwy;
  - $2n$ krawędzi (podzbiór E_M) łączących wierzchołki o etykietach i i i' oraz wierzchołki o etykietach i' i i".

Fakt 7 *Graf G_π jest sumą rozłącznych cykli parzystej długości.*

Dowód: Stopień każdego wierzchołka w G_π jest równy 2. □

Z faktu tego wprost wynika istnienie kolorowania wierzchołków G_π dwoma kolorami (powiedzmy białym i czarnym). Kolorowanie to ma następujące własności:

- Wierzchołki odpowiadające sąsiednim portom (zarówno wejściowym jak i wyjściowym) otrzymują różne kolory.
- Wierzchołki o etykietach i oraz i" otrzymują ten sam kolor (dla każdego $i = 1, \ldots, n$).

Stąd wnioskujemy istnienie ustawienia przełączników realizującego π:

- Przełączniki z pierwszej warstwy ustawiamy tak, by dane z portów białych (dokładniej: których odpowiadające wierzchołki otrzymały kolor biały) były przesłane do górnej podsieci $Perm_{n/2}$.
- Przełączniki w górnej podsieci $Perm_{n/2}$ ustawiamy tak, by permutowała swoje dane zgodnie z permutacją π. Dokładniej:
  Niech $K = k_1, \ldots, k_{n/2}$ będzie ciągiem etykiet białych wierzchołków z V_I w kolejności ich występowania w ciągu $\{1, 2, \ldots, n\}$ a $L = l_1, \ldots, l_{n/2}$ ciągiem etykiet białych

wierzchołków z V_O w kolejności ich występowania w ciągu $\pi(1), \ldots, \pi(n)$. Niech $\pi_a : \{1, \ldots, n/2\} \to \{1, \ldots, n/2\}$ będzie permutacją taką, że $\pi_a(i) = j$ wtedy i tylko wtedy gdy $l_j = k_i''$. Przełączniki ustawiamy tak, by podsieć realizowała permutację π_a. Takie ustawienie istnieje na mocy indukcji. Podobne rozważania przeprowadzamy dla dolnej podsieci $Perm_{n/2}$.

- Dla przełączników ostatniej warstwy stosujemy następującą regułę:
  Niech i" będzie etykietą białego wierzchołka z pary odpowiadającej j-temu przełącznikowi, a k" - etykietą czarnego wierzchołka z tej pary. Jeśli i poprzedza k w permutacji π (tzn. $i = \pi^{-1}(2j-1)$ oraz $k = \pi^{-1}(2j)$) przełącznik ustawiamy na wprost, w przeciwnym razie ustawiamy na ukos.

12.2 Para najbliżej położonych punktów

PROBLEM:
Dane: Zbiór $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ współrzędnych punktów na płaszczy"xnie.
Zadanie: Znale"xć dwa najbliżej położone względem siebie punkty w P, tj. znale"xć i,j takie, że $d(x_i, y_i, x_j, y_j) = min\{d(x_k, y_k, x_l, y_l) \mid 1 \leq k < l \leq n\}$, gdzie $d(x_k, y_k, x_l, y_l) = \sqrt{(x_k - x_l)^2 + (y_k - y_l)^2}$.

Siłowe rozwiązanie, polegające na wyliczeniu i porównaniu odległości między każdą parą punktów, wymaga czasu $\Omega(n^2)$. Przedstawimy teraz strategię Dziel i Zwyciężaj, która daje algorytm działający w czasie $\Theta(n \log n)$.

TERMINOLOGIA: mówiąc o punkcie r będziemy mieć na myśli punkt o współrzędnych (x_r, y_r).

12.2.1 Strategia Dziel i Zwyciężaj

1. (a) Sortujemy punkty z P według współrzędnych x i zapamiętujemy je w tablicy X;
   (b) Sortujemy punkty z P według współrzędnych y i zapamiętujemy je w tablicy Y;
   (c) Znajdujemy prostą l dzielącą P na dwa równoliczne (z dokładnością do 1) podzbiory:
   - P_L- podzbiór punktów leżących na lewo od l,
   - P_R - podzbiór punktów leżących na prawo od l.

   Punkty znajdujące się na prostej l (o ile są takie) kwalifikujemy do tych podzbiorów w dowolny sposób.

2. { rekurencyjnie }
   $(i_1, j_1) \leftarrow$ para punktów z P_L o najmniejszej odległości;
   $(i_2, j_2) \leftarrow$ para punktów z P_R o najmniejszej odległości.

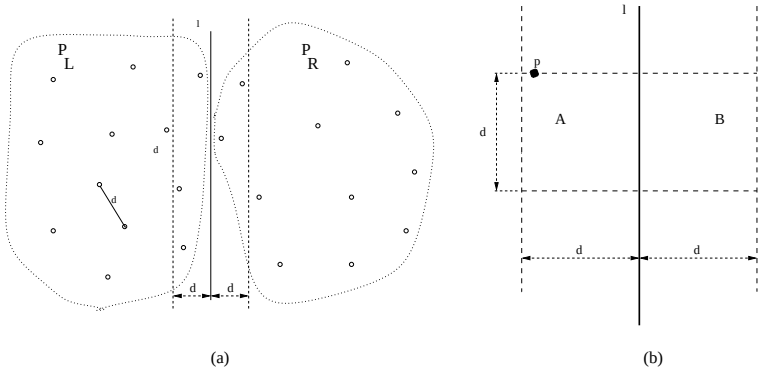
3. Niech (i', j') będzie tą parą punktów znalezioną w kroku 2, dla której odległość (oznaczmy ją przez d) jest mniejsza.
   Sprawdzamy czy istnieje para punktów (t, s) odległych o mniej niż d takich, że $t \in P_L$ i $s \in P_R$. Jeśli istnieje, przekazujemy ją jako wynik procedury, w przeciwnym razie jako wynik przekazujemy parę (i', j').

□

Wyjaśnienia wymaga sposób realizacji kroku 3. Oznaczmy przez P_C zbiór tych punktów z P, które leżą w odległości nie większej niż d od prostej l. Niech Y' oznacza tablicę Y, z której usunięto wszystkie punkty spoza P_C. Korzystamy z następującego spostrzeżenia:

Fakt 8 *Jeśli (t, s) jest parą punktów odległych o mniej niż d taką, że $t \in P_L$ i $s \in P_R$, to t i s należą do P_C. Ponadto w tablicy w Y' pomiędzy t a s leży nie więcej niż 6 punktów.*

Dowód: Gdyby jeden z punktów leżał w odległości większej niż d od prostej l, to odległość między nimi byłaby większa niż d. Oczywiste jest też, że współrzędne y-kowe tych punktów różnią się nie więcej niż o d. Tak więc punkty t i s leżą w prostokącie o wymiarach $d \times 2d$ jak pokazano na rysunku **??**.



Rysunek 4: *(a) W kroku 3 pary (t, s) należy szukać tylko w zaznaczonym pasie. (b) Jeśli p ma tym punktem z pary (t, s), który ma większą współrzędną y, to drugi punkt z tej pary musi znajdować się w kwadracie B. Ponadto wszystkie punkty znajdujące się w Y między t a s muszą leżeć w A lub w B.*

W części A leżą tylko punkty z P_L. Ponieważ każde dwa z nich odległe są od siebie o co najmniej d, więc może ich tam znajdować się co najwyżej 4. Z analogicznego powodu w części B może znajdować się nie więcej niż 4 punkty z P_R. Tak więc w całym prostokącie znajduje się nie więcej niż 8 punktów.

Krok 3 sprowadza się więc do utworzenia tablicy Y', a następnie do obliczenia odległości każdego punktu z Y' do co najwyżej siedmiu punktów następujących po nim w tej tablicy.

12.2.2 Koszt:

Krok 1:

– Sortowanie - $\Theta(n \log n)$.

– Znalezienie prostej l i podział P na podzbiory - koszt stały.

Krok 2: $2T(n/2)$

Krok 3:

– Utworzenie Y' - $\Theta(n)$.

– Szukanie pary (t, s) - $\Theta(n)$.

Stąd koszt całego algorytmu wyraźa się równaniem $T(n) = 2T(n/2) + \Theta(n \log n)$, którego rozwiązaniem jest $\Theta(n \log^2 n)$. Koszt ten można zredukować do $\Theta(n \log n)$. Wystarczy zauważyć , źe sortowanie punktów w każdym wywołaniu rekurencyjnym jest zbyteczne. Zbiór P możemy przekazywać kolejnemu wywołaniu rekurencyjnemu jako tablice X i Y. Na ich podstawie można w czasie liniowym utworzyć odpowiednie tablice dla zbiorów P_L i P_R. Tak więc sortowanie wystarczy przeprowadzić jeden raz - przed pierwszym wywołaniem procedury rekurencyjnej.

Po takiej modyfikacji czas wykonania procedury rekurencyjnej wyraźa się równaniem $T(n) = 2T(n/2) + \Theta(n)$, którego rozwiązaniem jest $\Theta(n \log n)$. Dodany do tego czas sortowania nie zwiększa rzędu funkcji.

Notatki z AiSD. Nr 5. 16 marca 2005

Programowanie dynamiczne

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

13 Wstęp

Zastosowanie metody Dziel i Zwyciężaj do problemów zdefiniowanych rekurencyjnie jest w zasadzie ograniczone do przypadków, gdy podproblemy, na które dzielimy problem, są niezależne. W przeciwnym razie metoda ta prowadzi do wielokrotnego obliczania rozwiązań tych samych podproblemów. Jednym ze sposobów zaradzenia temu zjawisku jest tzw. *spamiętywanie*, polegające na pamiętaniu rozwiązań podproblemów napotkanych w trakcie obliczeń. W przypadku, gdy przestrzeń wszystkich moźliwych podproblemów jest nieduźa, efektywniejsze od spamiętywania moźe być zastosowanie metody programowania dynamicznego. Metoda ta polega na obliczaniu rozwiązań dla wszystkich podproblemów, począwszy od podproblemów najprostszych.

Przykład 1.

Problem:

Dane: Liczby naturalne n, k.

Wynik: $\binom{n}{k}$.

Naturalna metoda redukcji problemu obliczenia $\binom{n}{k}$ korzysta z zależności $\binom{n}{k}=\binom{n-1}{k-1}+\binom{n-1}{k}$. Zastosowanie metody Dziel i Zwyciężaj byłoby jednak w tym przypadku nierozważne, ponieważ w trakcie liczenia $\binom{n-1}{k-1}$ jak i $\binom{n-1}{k}$ wywoływalibyśmy rekurencyjnie procedurę dla tych samych danych (tj. dla $n-2$ i $k-1$), co w konsekwencji prowadziłoby do tego, źe niektóre podproblemy byłyby rozwiązywane wykładniczą liczbę razy.

Do spamiętywania moźemy wykorzystać tablicę $tab[1..n, 1..k]$.

```
for i=1 to n do
    for j = 0 to k do tab_{i,j} ←"?"
            .....................
function nPOk(n, k)
    if tab_{n-1,k-1} ="?" then tab_{n-1,k-1} ←nPOk(n − 1, k − 1)
    if tab_{n-1,k} ="?" then tab_{n-1,k} ←nPOk(n − 1, k)
    tab_{n,k} = tab_{n-1,k-1} + tab_{n-1,k}
    return tab_{n,k}
```

Za zastosowaniem w tym przypadku programowania dynamicznego przemawia fakt, iź liczba różnych podproblemów jakie mogą pojawić się w trakcie obliczania $\binom{n}{k}$ jest niewielka, a mianowicie $O(n^2)$. Podobnie jak w metodzie spamiętywania, algorytm dynamiczny oblicza początkowy fragment trójkąta Pascala i umieszcza go w tablicy *tab*. W przeciwień-

stwie jednak do poprzedniej metody, która jest metodą "top-down" i jest implementowana rekurencyjnie, algorytm dynamiczny jest metodą "bottom-up" i jest implementowany iteracyjnie. To pozwala w szczególności na wyeliminowanie kosztów związanych z obsługą rekursji.

```
for i = 1 to n do tab_{i,0} ← 1
          .....
function nPOk(n, k)
    for j = 1 to k do
        tab_{j,j} ← 1
        for i = j + 1 to n do tab_{i,j} ← tab_{i-1,j-1} + tab_{i-1,j}
    return tab_{n,k}
```

Fakt, że metoda programowania dynamicznego oblicza w sposób systematyczny rozwiązania wszystkich podproblemów, pozwala często na poczynienie dodatkowych oszczędności w stosunku do metody spamiętywania. W tym przykładzie możemy znacznie zredukować koszty pamięciowe. Jak łatwo zauważyć, obliczenie kolejnej przekątnej trójkąta Pascala wymaga znajomości jedynie wartości z poprzedniej przekątnej. Tak więc zamiast tablicy $n \times k$ wystarcza tablica $n \times 2$, a nawet tablica $n \times 1$.

□

Podobnie jak w przypadku metody dziel i zwyciężaj, kluczem do zastosowania programowania dynamicznego jest znalezienie sposobu dzielenia problemu na podproblemy w taki sposób, by optymalne rozwiązanie problemu można było w prosty sposób otrzymać z optymalnych rozwiązań podproblemów (mówimy wówczas, że problem wykazuje *optymalną podstrukturę*). Wskazaniem za stosowaniem wówczas programowania dynamicznego a nie metody dziel i zwyciężaj jest sytuacja, gdy sumaryczny rozmiar podproblemów jest duży. Oczywiście, jak już wspominaliśmy, aby algorytm dynamiczny był efektywny, przestrzeń wszystkich możliwych podproblemów nie może być zbyt liczna.

Przykład 2.

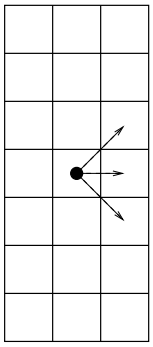
Problem:

Dane: Tablica $\{a_{i,j}\}$ liczb nieujemnych $(i = 1, \ldots, n;\ j = 1, \ldots, m)$

Wynik: Ciąg indeksów $i_1, \ldots, i_m$ taki, że $\forall_{j=1,\ldots,m-1} |i_j - i_{j+1}| \leq 1$, minimalizujący sumę $\sum_{j=1}^{m} a_{i_j,j}$

Interpretacja: Ciąg $i_1, \ldots, i_m$ wyznacza trasę wiodącą od pierwszej do ostatniej kolumny tablicy a. Startujemy z dowolnego pola pierwszej kolumny i kończymy na dowolnym polu ostatniej kolumny. W każdym ruchu przesuwamy się o jedno pole: albo w prawo na wprost albo w prawo na ukos (jak pokazano na rysunku 5). Chcemy znale"xć trasę o minimalnej długości, rozumianej jako suma liczb z pól znajdujących się na trasie.

Jak łatwo sprawdzić liczba wszystkich prawidłowych tras jest wykładnicza, więc rozwiązanie siłowe nie wchodzi w rachubę.



Rysunek 5: *Możliwe kierunki ruchu w tablicy a.*

Rozważmy najpierw nieco prostsze zadanie, polegające na znalezieniu długości optymalnej trasy. Potem pokażemy w jaki sposób zorganizować obliczenia, by wyznaczenie samej trasy było proste.

Niech $d_{i,k}$ oznacza minimalną długość trasy wiodącej od dowolnego pola pierwszej kolumny do pola $a_{i,k}$, a $P(i,k)$ problem wyznaczenia $d_{i,k}$. Rozwiązanie $P(i,k)$ (dla $k > 1$) można łatwo otrzymać z rozwiązań trzech prostszych podproblemów, a mianowicie $P(i-1,k-1)$, $P(i,k-1)$ i $P(i+1,k-1)$ (w przypadku $P(1,k)$ i $P(n,k)$ - dwóch podproblemów). Problem wykazuje więc optymalną podstrukturę.

Jeśli za rozmiar $P(i,k)$ przyjmiemy wartość k, to problem rozmiaru k redukujemy do trzech podproblemów rozmiaru $k-1$. To zbyt duże rozmiary, by opłaciło się stosować metodę dziel i zwyciężaj. Z drugiej strony przestrzeń wszystkich podproblemów jest stosunkowo niewielka - składa się z nm elementów (zawiera wszystkie $P(i,j)$ dla $i = 1,\ldots,n, j = 1,\ldots,m$), możemy więc zastosować programowanie dynamiczne.

for $j = 1$ **to** n **do** $d_{0,j} \leftarrow d_{n+1,j} \leftarrow \infty$
for $i = 1$ **to** n **do** $d_{i,1} \leftarrow a_{i,1}$
for $j = 2$ **to** m **do**
  for $i = 1$ **to** n **do** $d_{i,j} \leftarrow a_{i,j} + \min\{d_{i-1,j-1}, d_{i,j-1}, d_{i+1,j-1}\}$
return $\min\{d_{i,m} \mid i = 1,\ldots,n\}$

Pozostaje wyjaśnić, w jaki sposób można odtworzyć optymalną trasę. Niech i_0 będzie wartością i, dla której osiągane jest $\min\{d_{i,m} \mid i = 1,\ldots,n\}$, a więc $a_{i_0,m}$ jest ostatnim polem optymalnej trasy. Przedostatnie pole możemy wyznaczyć sprawdzając, która z trzech różnic $d_{i_0,m} - d_{k,m-1}$ (dla $k \in \{i_0-1, i_0, i_0+1\}$) jest równa $a_{i_0,m}$. Postępując dalej rekurencyjnie wyznaczymy całą trasę. Łatwo zauważyć, że zamiast sprawdzać powyższe różnice, możemy ograniczyć się do sprawdzenia, która z wartości $d_{k,m-1}$ (dla $k \in \{i_0-1, i_0, i_0+1\}$) jest minimalna.

```
procedure trasa(i, j)
{  if j = 1 then return i
   Niech i' będzie takie, że d_{i',j-1} = min{d_{k,j-1} | k ∈ {i-1, i, i+1}}
   return concat( trasa(i', j-1), i))
}
            .....
write( trasa(i_0, m))
```

□

Programowanie dynamiczne jest częstą metodą rozwiązywania problemów optymalizacyjnych. Przykład 2 stanowi ilustrację klasycznego sposobu rozwiązania takiego problemu: najpierw znajdujemy wartość optymalnego rozwiązania a dopiero potem, na podstawie wyliczeń tej wartości, konstruujemy optymalne rozwiązanie.

14 Dalsze przykłady

14.1 Najdłuższy wspólny podciąg.

14.1.1 Definicja problemu

Definicja 4 *Ciąg* $Z = \langle z_1, z_2, \ldots, z_k \rangle$ *jest* podciągiem *ciągu* $X = \langle x_1, x_2, \ldots, x_n \rangle$*, jeśli istnieje ściśle rosnący ciąg indeksów* $\langle i_1, i_2, \ldots, i_k \rangle$ $(1 \leq i_j \leq n)$ *taki, że*

$$\forall_{j=1,2,\ldots k} \;\; x_{i_j} = z_j.$$

Jeśli Z *jest podciągiem zarówno ciągu* X *jak i ciągu* Y*, to mówimy, że* Z *jest* wspólnym podciągiem *ciągów* X *i* Y*.*

Konwencja: Dla wygody, w dalszej części ciągi będziemy traktować jako napisy nad ustalonym alfabetem.

Przykład:
'BABA' jest wspólnym podciągiem ciągów 'ABRACADABRA' i 'RABARBAR', ale nie jest ich najdłuższym wspólnym podciągiem (dłuższym jest np. 'RAAAR').

□

Oznaczenia:

- $LCS(X, Y) = \{Z \;\; | \;\; Z$ jest wspólnym podciągiem X i Y o maksymalnej długości$\}$
- przez X_i oznaczamy i-literowy prefiks ciągu $X = \langle x_1, x_2, \ldots, x_n \rangle$, tj. podciąg $\langle x_1, x_2, \ldots, x_i \rangle$; w szczególności przez X_0 oznaczamy ciąg pusty.

Problem:
Dane: ciągi $X = \langle x_1, x_2, \ldots, x_m \rangle$ i $Y = \langle y_1, y_2, \ldots, y_n \rangle$
Wynik: dowolny ciąg Z z $LCS(X, Y)$

14.1.2 Redukcja problemu

Problem znalezienia ciągu Z z $LCS(X,Y)$ możemy zredukować do prostszych problemów na podstawie następującej obserwacji:

- jeśli ostatnia litera X i ostatnia litera Y są takie same, to litera ta musi być ostatnim elementem każdego ciągu z $LCS(X,Y)$.
- jeśli X i Y różnią się na ostatniej pozycji (tj. $x_m \neq y_n$), to istnieje ciąg w $LCS(X,Y)$, który na ostatniej pozycji ma literę różną od x_m lub istnieje ciąg w $LCS(X,Y)$, który na ostatniej pozycji ma literę różną od x_m.

W pierwszym przypadku problem znalezienia ciągu z $LCS(X_m, Y_n)$ redukujemy do podproblemu znalezienia ciągu z $LCS(X_{m-1}, Y_{n-1})$. Rozwiązaniem będzie konkatenacja znalezionego ciągu i ostatniej litery X-a. W drugim przypadku problem redukujemy do dwóch podproblemów: znalezienie ciągu z $LCS(X_{m-1}, Y_n)$ i znalezienie ciągu z $LCS(X_m, Y_{n-1})$. W tym przypadku rozwiązaniem będzie dłuższy ze znalezionych ciągów.

14.1.3 Algorytm

Najpierw koncentrujemy się na obliczeniu wartości rozwiązania optymalnego, którą w tym przypadku jest długość elementów z $LCS(X,Y)$. Sposobu na obliczenie tej wartości dostarcza nam obserwacja poczyniona w poprzednim paragrafie.

Fakt 9 *Niech $d_{i,j}$ oznacza długość elementów z $LCS(X_i, Y_j)$. Wówczas:*

$$d_{i,j} = \begin{cases} 0 & \text{jeśli } i = 0 \text{ lub } j = 0, \\ 1 + d_{i-1,j-1} & \text{jeśli } i,j > 0 \text{ i } x_i = y_j, \\ \max(d_{i,j-1}, d_{i-1,j}) & \text{jeśli } i,j > 0 \text{ i } x_i \neq y_j \end{cases}$$

Tablicę d możemy obliczać kolejno wierszami (lub kolumnami), a wynik odczytamy z $d_{m,n}$.

```
Procedure LCS(X_m, Y_n)
    for i ← 1 to m do d_{i,0} ← 0
    for j ← 0 to n do d_{0,j} ← 0
    for i ← 1 to m do
        for j ← 1 to n do
            if x_i = y_j then d_{i,j} ← 1 + d_{i-1,j-1}
            else d_{i,j} ← max{d_{i-1,j}, d_{i,j-1}}
```

Aby wypisać jakiś element z LCS musimy przejść tablicę d jeszcze raz, począwszy od elementu $d_{n,m}$, w podobny sposób jak to robiliśmy w Przykładzie 2.

Jeśli zależy nam na szybkości algorytmu, możemy nieco przyspieszyć tę jego fazę. W tym celu, w trakcie obliczania tablicy d, możemy w dodatkowej tablicy zapamiętywać "drogę dojścia"do poszczególnych elementów tablicy d. Elementy dodatkowej tablicy przyjmowałyby jedną z trzech różnych wartości, w zależności od tego czy $d_{i,j}$ powstał przez dodanie 1 do $d_{i-1,j-1}$, czy przez przepisanie $d_{i-1,j}$, czy też wreszcie przez przepisanie $d_{i,j-1}$.

14.1.4 Koszt algorytmu

Obliczenie każdego elementu tablicy d odbywa się w czasie stałym. Tak więc całkowity koszt wypełnienia tablicy d jest równy $\Theta(n \cdot m)$. Koszt skonstruowania najdłuższego podciągu na podstawie tablicy d jest liniowy.

14.2 Wyznaczanie optymalnej kolejności mnożenia macierzy.

14.2.1 Definicja problemu

Mamy obliczyć wartość wyrażenia $\mathcal{M}$ postaci $M_1 \times M_2 \times \cdots \times M_n$, gdzie M_i są macierzami. Zakładamy, że wyrażenie jest poprawne, tj. liczba kolumn macierzy M_i jest równa liczbie wierszy macierzy M_{i+1} (dla $i = 1, \ldots, n-1$).

Ponieważ mnożenie macierzy jest działaniem łącznym, wartość $\mathcal{M}$ możemy liczyć na wiele sposobów. Wybór sposobu może w istotny sposób wpłynać na liczbę operacji skalarnych jakie wykonamy podczas obliczeń.

PRZYKŁAD Niech macierze M_1, M_2, M_3 mają wymiary odpowiednio $d \times 1$, $1 \times d$ i $d \times 1$. Rozważmy dwa sposoby obliczenia ich iloczynu:

- $(M_1 \times M_2) \times M_3$
  W wyniku pierwszego mnożenia otrzymujemy macierz $d \times d$, więc jego koszt (niezależnie od przyjętej metody mnożenia macierzy) wynosi co najmniej d^2. W drugim mnożeniu także musimy wykonać $\Theta(d^2)$ operacji.
- $M_1 \times (M_2 \times M_3)$
  Koszt obliczenia $M_2 \times M_3$ wynosi $O(d)$. W jego wyniku otrzymujemy macierz 1×1, więc koszt następnego mnożenia wynosi także $O(d)$.

□

Dalsze rozważania będziemy przeprowadzać przy następującym założeniu[2]:

`Koszt pomnożenia macierzy o wymiarach` $a \times b$ `i` $b \times c$ `wynosi` abc.

PROBLEM:

Dane: $d_0, d_1, \ldots, d_n$ - liczby naturalne
INTERPRETACJA: $d_{i-1} \times d_i$ - wymiar macierzy M_i.

Zadanie: Wyznaczyć kolejność mnożenia macierzy $M_1 \times M_2 \times \cdots \times M_n$, przy której koszt obliczenia tego iloczynu jest minimalny.

14.2.2 Rozwiązanie siłowe

Rozwiązanie siłowe, polegajace na sprawdzeniu wszystkich moźliwych sposobów wykonania obliczeń, jest nieakceptowalne. Liczba tych sposobów dana jest wzorem

$$S(n) = \begin{cases} 1 & \text{jeśli } n = 1 \\ \sum_{i=1}^{n-1} S(i)S(n-i) & \text{jeśli } n > 1 \end{cases}$$

[2]Jest to koszt mnożenia wykonanego metodą tradycyjną; pó"xniej poznamy inne, szybsze metody.

UZASADNIENIE WZORU: Każde z $n-1$ mnożeń jakie występują w ciągu $M_1 \times \ldots \times M_n$ może być ostatnim, jakie wykonamy licząc ten iloczyn. Liczba sposobów mnożenia macierzy, w których i-te mnożenie jest ostatnim, jest równa iloczynowi $S(i)$ (tj. liczby sposobów, na które można pomnożyć i pierwszych macierzy) oraz $S(n-i)$ tj. liczby sposobów, na które można pomnożyć $n-i$ ostatnich macierzy).

Rozwiązaniem powyższego równania jest $S(n) =$"$n-$ta liczba Catalana"$= \frac{1}{n}\binom{2n-2}{n-1} = \Omega(\frac{4^n}{n^2})$. Tak więc koszt sprawdzania wszystkich moźliwych iloczynów jest wykładniczy.

14.2.3 Rozwiązanie dynamiczne

Zauważamy, źe problem wykazuje optymalną podstrukturę. Jeśli bowiem k-te mnożenie jest ostatnim, jakie wykonamy w optymalnym sposobie obliczeń, to iloczyny $M_1 \times \ldots \times M_k$ oraz $M_{k+1} \times \ldots \times M_n$ teź musiały być obliczone w optymalny sposób.

Na podstawie tej własności możemy ułożyć następujący algorytm rekurencyjny obliczający optymalny koszt obliczeń.

```
function matmult(i, j)
    if i = j then return 0
    opt ← ∞
    for k ← i to j − 1 do
        opt ← min(opt, d_{j−1} d_k d_j + matmult(i, k) + matmult(k + 1, j))
    return opt
```

Algorytm ten, jakkolwiek szybszy od metody siłowej, nadal działa w czasie wykładniczym ($\Theta(3^n)$). Przyczyna tkwi w wielokrotnym wykonywaniu obliczeń dla tych samych wartości parametrów (i, j). Unikniemy tego mankamentu stosując programowanie dynamiczne. Niech

$$m_{i,j} = \text{"minimalny koszt obliczenia } M_i \times M_{i+1} \times \cdots \times M_j\text{"}$$

Dla wygody przyjmujemy, źe $m_{i,j} = 0$ (dla $i \geq j$). Wówczas

$$m_{i,j} = \min_{i \leq k < j}(m_{i,k} + m_{k+1,j} + d_{i-1}d_k d_j).$$

Składnik $m_{i,k}$ jest kosztem obliczenia $M_i \times M_{i+1} \times \cdots \times M_k$, składnik $m_{k+1,j}$ - kosztem obliczenia $M_{k+1} \times M_{k+2} \times \cdots \times M_j$, natomiast $d_{i-1}d_k d_j$ to koszt obliczenia iloczynu dwóch powstałych macierzy.

```
procedure dyn − matmult(d[0..n]);
    int m[1..n, 1..n], p[1..n, 1..n]
     for i ← 1 to n do m_ii ← 0;
     for s ← 1 to n − 1 do
         for i ← 1 to n − s do
             j ← i + s
             m_ij ← min_{i≤k<j}(m_{i,k} + m_{k+1,j} + d_{i−1} d_k d_j)
             p_ij ← "to k, przy którym osiągane było minimum dla m_ij"
     return p[1..n, 1..n]
```

Algorytm oblicza wartości $m_{i,i+s}$ (na podstawie powyższego wzoru) oraz wartości $p_{i,i+s}$, które umoźliwiają pó"xniejsze skonstruowanie rozwiązania.

Koszt algorytmu. Tablicę $m_{i,j}$ liczymy przekątna za przekątną począwszy od głównej przekątnej. Koszt policzenia jednego elementu $m_{i,i+s}$ na s-tej przekątnej wynosi $\Theta(s)$. Ponieważ na s-tej przekątnej znajduje się $n - s$ elementów, koszt algorytmu wynosi

$$T(n) = \sum_{s=0}^{n-1} \Theta(s) \cdot (n - s) = \Theta(n^3).$$

Odtworzenie rozwiązania Odtworzenia rozwiązania dokonujemy w standardowy sposób na podstawie tablicy p. Zwróć uwagę, źe znalezienie rozwiązania na podstawie samych tylko wartości m_{ij} wymagałoby czasu $\Theta(n^2)$.

Notatki z AiSD. Nr 6. 16 marca 2005

Programowanie dynamiczne c.d.

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

2.2 Rozpoznawanie języków bezkontekstowych

2.2.1 Definicja problemu

Rozpoczynamy od przypomnienia podstawowych pojęć związanych z gramatykami bezkontekstowymi (pojęcia te powinny być znane z wykładu Wstęp do Informatyki).

Definicja 5 Gramatyką bezkontekstową *nazywamy system* $G = \langle V_N, V_T, P, S\rangle$, *gdzie*

- V_N *i* V_T *są skończonymi rozłącznymi zbiorami (nazywamy je odpowiednio* alfabetem symboli nieterminalnych *i* alfabetem symboli terminalnych*);*
- P *jest skończonym podzbiorem zbioru* $V_N \times (V_N \cup V_T)^*$ *(elementy* P *nazywamy* produkcjami*);*
- $S \in V_N$ *i jest nazywany* symbolem początkowym gramatyki.

Zwyczajowo produkcje (A, α) zapisujemy jako $A \to \alpha$.

Definicja 6 *Jeśli każda produkcja gramatyki bezkontekstowej* G *jest postaci:*

- $A \to BC$ *lub*
- $A \to a$,

gdzie $A, B, C \in V_N$ *i* $a \in V_T$, *to mówimy, że* G *jest w* normalnej postaci Chomsky'ego.

Definicja 7 *Niech* $G = \langle V_N, V_T, P, S\rangle$; $\alpha, \beta, \gamma \in (V_N \cup V_T)^*$ *oraz* $A \in V_N$. *Mówimy, że ze słowa* $\alpha A \beta$ *można w* G wyprowadzić *słowo* $\alpha\gamma\beta$, *co zapisujemy* $\alpha A\beta \Rightarrow \alpha\gamma\beta$, *jeśli* $A \to \gamma$ *jest produkcją z* P.

Definicja 8 *Język* $L(G)$ *generowany przez gramatykę* $G = \langle V_N, V_T, P, S\rangle$ *definiujemy jako*

$$L(G) = \{w \mid w \in V_T^* \text{ oraz } S \stackrel{*}{\Rightarrow} w\},$$

gdzie $\stackrel{*}{\Rightarrow}$ *oznacza tranzytywne domknięcie relacji* $\Rightarrow$.

Przykład 1. Niech

- $V_N = \{S, T, L, R\}$;
- $V_T = \{(,)\}$;
- $P = \{S \to SS\ ,\ S \to LT\ ,\ S \to LR\ ,\ T \to SR\ ,\ L \to (\ ,\ R \to)\ \}$

Jak łatwo sprawdzić $L(G)$ jest językiem zawierający wszystkie słowa zbudowanie z poprawnie rozstawionych nawiasów.
Przykładowe wyprowadzenie słowa w=(()()):

$$S \Rightarrow LT \Rightarrow LSR \Rightarrow LSSR \Rightarrow LLRSR \Rightarrow LLRLRR \Rightarrow (LRLRR \Rightarrow$$

$$(LRL)R \Rightarrow (L)L)R \Rightarrow (L)()R \Rightarrow (()()R \Rightarrow (()())$$

□

Dla ustalonej gramatyki bezkontekstowej $G = \langle V_N, V_T, P, S\rangle$ w normalnej postaci Chomsky'ego rozważamy następujący problem.

PROBLEM:
Dane: słowo $w = a_1 \dots a_n$ ($a_i \in V_T$ dla $i = 1, \dots, n$)
Wynik: "TAK'' - jeśli $w \in L(G)$
''NIE'' - w przeciwnym przypadku.

2.2.2 Algorytm naiwny

Niech $M(w)$ oznacza zbiór słów wyprowadzalnych z w w jednym kroku.

$F_0 \leftarrow \{S\}$
for $i = 1$ **to** $2|w| - 1$ **do**
  $F_{i+1} \leftarrow \bigcup_{u \in F_i} M(u)$
if $w \in F_{2|w|-1}$ **then return** "TAK" **else return** "NIE"

POPRAWNOŚĆ: Każda produkcja gramatyki w normalnej postaci Chomsky'ego albo zwiększa o jeden długość wyprowadzanej frazy albo zamienia symbol nieterminalny na terminalny. Tak więc każde słowo z języka o długości n jest wyprowadzane z S po $2n - 1$ krokach.

KOSZT: Czynnikiem determinującym koszt algorytmu jest koszt pętli wewnętrznej, a ten w głównym stopniu zależy od wielkości zbiorów F_i. Niestety, nawet dla tak prostych gramatyk jak ta z Przykładu 1, zbiory F_i mogą zawierać wykładniczo wiele słów.

2.2.3 Algorytm dynamiczny

IDEA:

Jeśli $w = a_1 \dots a_n$ jest słowem z języka $L(G)$, to pierwsza produkcja zastosowana w jego wyprowadzeniu (o ile $n > 1$) musi mieć postać $S \to AB$. Ponieważ dalsze wyprowadzenie z symbolu A jest niezależne od wyprowadzenia z symbolu B, więc musi istnieć i ($1 \leq i \leq n-1$) takie, że $A \stackrel{*}{\Rightarrow} a_1 \dots a_i$ oraz $B \stackrel{*}{\Rightarrow} a_{i+1} \dots a_n$.

Na postawie tej obserwacji możemy łatwo zbudować algorym rekurencyjny, jednak czas jego działania może być wykładniczy. W szczególności algorytm taki wielokrotnie może próbować wyprowadzać ten sam fragment słowa w z tego samego symbolu nieterminalnego.

PRZYKŁAD 2. Niech gramatyka zawiera (między innymi) produkcje $S \to AB$ oraz $A \to AA$. Na drugim poziomie rekursji rekurencyjna procedura może być wywoływana dla A i podsłów $a_1 \dots a_i$ (dla $i = 1, \dots, n-1$); wewnątrz każdego z tych wywołań będzie ona znów wywoływana m.in. dla A i podsłów $a_1 \dots a_j$ $(j = 1, \dots, i-1)$.

□

Podejście dynamiczne polega na obliczeniu dla każdego podsłowa słowa w (począwszy od podsłów jednoliterowych a skończywszy na całym w) zbioru nieterminali, z których da się to podsłowo wyprowadzić. Innymi słowy, celem jest wyznaczenie zbiorów $m_{i,j}$ $(1 \le i \le j \le n)$:

$$m_{i,j} = \{A \mid A \in V_N \ \& \ A \stackrel{*}{\Rightarrow} a_i \dots a_j\}$$

Odpowiedzią algorytmu będzie wartość wyrażenia $S \in m_{1,n}$.

Zbiory $m_{i,j}$ wyznaczyć można na podstawie następujących zależności:

$$m_{i,i} = \{A \mid (A \to a_i) \in P\} \text{ dla } i = 1, \dots, n$$

$$m_{i,j} = \bigcup_{k=i}^{j-1} m_{i,k} \otimes m_{k+1,j} \text{ dla } 1 \le i < j \le n$$

gdzie $m_{i,k} \otimes m_{k+1,j} = \{A \mid (A \to BC) \in P \text{ dla pewnych } B \in m_{i,k} \text{ oraz } C \in m_{k+1,n}\}$

KOSZT: Łatwo sprawdzić, że algorytm wykonuje $\Theta(n^3)$ operacji $\otimes$. Ponieważ koszt jednej operacji $\otimes$ jest stały (patrz Uwagi implementacyjne) , $\Theta(n^3)$ opisuje koszt całego algorytmu.

Uwagi implementacyjne. Elementy obliczanej tablicy są zbiorami. To stanowi istotną różnicę w stosunku do poprzednich przykładów, gdzie elementy tablicy były prostego typu. Przyjęcie odpowiedniej struktury danych do pamiętania zbiorów $m_{i,j}$ oraz wybór metody obliczania wyniku operacji $\otimes$ może mieć istotny wpływ na koszt algorytmu.

Przykładowo: zbiory $m_{i,j}$ możemy pamiętać jako wektory charakterystyczne lub jako listy. W pierwszym przypadku potrzebujeny $\sim (1/2)n^2|V_N|$ bitów na zapamiętanie tablicy. W drugim przypadku ponosimy spore koszty pamięciowe związane z używaniem wska"xników - jednak mogą one być opłacalne, gdy w średnim przypadku rozmiar zbiorów $m_{i,j}$ jest nieduży. Wówczas rozsądną metodą obliczania $m_{i,k} \otimes m_{k+1,j}$ może okazać się zwykłe przeglądanie list:

```
for each B ∈ m_{i,k} do
    for each C ∈ m_{k+1,j} do
        if BC jest prawą stroną produkcji z P
        then m_{i,j} ← { symbol z lewej strony tej produkcji }
```

Przy odpowiednim zapamiętaniu informacji o produkcjach, koszt takiego obliczenia nie zależy od liczby produkcji i jest proporcjonalny do iloczynu długości list, co w rozważanym

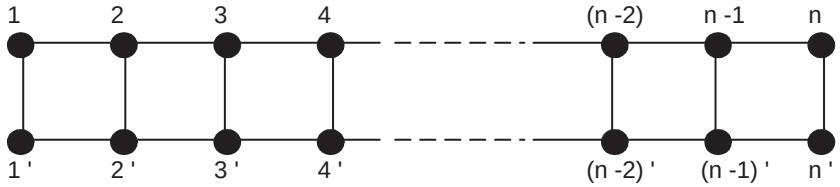
przypadku może być znacznie mniejsze od $|V_N|^2$. Jeśli liczba produkcji jest niewielka opłacalne może być zastosowanie innego sposobu:

```
for each (A → BC) ∈ P do
    if B ∈ m_{i,k} & C ∈ m_{k+1,j} then m_{i,j} ← m_{i,j} ∪ {A}
```

Sposób ten jest szczególnie atrakcyjny przy wektorowej reprezentacji zbiorów, ponieważ wówczas czas odpowiedzi na pytanie o przynależność elementu do zbioru jest stały i koszt powyższej pętli wynosi $\Theta(|P|)$.

2.3 Drzewa rozpinające drabin

Definicja 9 Drabiną *n-elementową nazywamy graf D_n przedstawiony na rysunku 6.*



Rysunek 6: *Drabina n elementowa.*

PROBLEM:

Dane: liczby naturalne n, k;
ciąg par liczb naturalnych $\{u_i, v_i\}$ $(i = 1, \ldots, m)$
INTERPRETACJA: pary $\{u_i, v_i\}$ określają wyróżnione krawędzie w n-elementowej drabinie;

Wynik: Liczba drzew rozpinających o k krawędziach wyróżnionych.

Ideę algorytmu przedstawimy rozważając prostszy problem, a mianowicie problem wyznaczania liczby drzew rozpinających w D_n bez uwzględniania krawędzi wyróżnionych. Co prawda, w takim przypadku można w prosty sposób wyprowadzić zwięzły wzór na tę liczbę, lecz nie to jest naszym celem.

W dalszym ciągu, mówiąc o drabinie D_i, będziemy mieć na myśli podgraf drabiny D_n indukowany przez wierzchołki $\{1, \ldots, i, 1', \ldots, i'\}$.

Fakt 10 *Niech T będzie dowolnym drzewem rozpinającym drabiny D_{i+1}, dla dowolnego $i \geq 1$. Wówczas $T \cap D_i$ jest albo*

- *drzewem rozpinającym drabiny D_i albo*
- *lasem rozpinającym drabiny D_i złożonym z dwóch drzew; jedno z tych drzew zawiera wierzchołek i, a drugie - wierzchołek i'.*

Analogiczna własność zachodzi gdy T jest lasem rozpinającym drabiny D_{i+1}, złożonym z dwóch drzew, przy czym jedno z tych drzew zawiera wierzchołek $(i+1)$, a drugie - weirzchołek $(i+1)'$. □

Niech S_i oznacza zbiór drzew rozpinających drabiny D_i, a N_i – zbiór lasów rozpinających, o których mowa w Fakcie 10, w drabinie D_i. Naszym celem jest policzenie wartości $|S_n|$.

IDEA ALGORYTMU: Kolejno dla $i = 1, \ldots, n$ liczymy wartości $|S_i|$ oraz $|N_i|$, korzystając z zależności przedstawionych w poniższym fakcie:

Fakt 11 *(a)* $|S_1| = |N_1| = 1$

(b) Dla każdego $i > 1$:

$$|S_i| = 3|S_{i-1}| + |N_{i-1}|,$$
$$|N_i| = 2|S_{i-1}| + |N_{i-1}|.$$

DOWÓD:

(a) Oczywiste.

(b) Niech $K_i = \{(i-1, i),\ ((i-1)', i'),\ (i, i')\}$ będzie zbiorem krawędzi, którymi D_i różni się od D_{i-1}.

Z dowolnego drzewa rozpinającego $T \in S_{i-1}$ można utworzyć trzy różne drzewa rozpinające z S_i poprzez dodanie do T dowolnych dwóch krawędzi ze zbioru K_i. Ponadto, dodając wszystkie krawędzie z K_i do dowolnego lasu z N_{i-1} można utworzyć jedno drzewo z S_i. To uzasadnia pierwszy ze wzorów.

Dodając krawęd"x $(i-1, i)$ do drzewa $T \in S_{i-1}$ otrzymujemy las z N_i. Jedno z jego drzew zawiera wierzchołek i, a drugie z drzew składa się z izolowanego wierzchołka $\{i'\}$. Analogicznie, otrzymujemy jeden las dodając krawęd"x $((i-1)', i')$ do T. Ponadto z każdego lasu z N_{i-1}, po dodaniu dwóch poziomych krawędzi $(i-1, i)$ oraz $((i-1)', i')$, otrzymujemy jeden las z N_i. To uzasadnia drugi wzór.

□

Teraz w prosty sposób możemy tę metodę uogólnić do rozwiązania problemu liczenia drzew rozpinających z wyróżnionymi krawędziami. W tym celu zamiast dwóch zbiorów S_i i N_i rozważamy $2(k+1)$ zbiory: $S_i(j),\ N_i(j)$, gdzie parametr j $(j = 0, \ldots, k)$ oznacza liczbę krawędzi wyróżnionych. Przykładowo: $S_i(j)$ będzie się równać liczbie drzew rozpinających w drabinie D_i zawierających dokładnie j krawędzi wyróżnionych. Wyprowadzenie wzorów analogicznych do tych z Faktu 11 pozostawiamy jako proste ćwiczenie.

Sortowanie

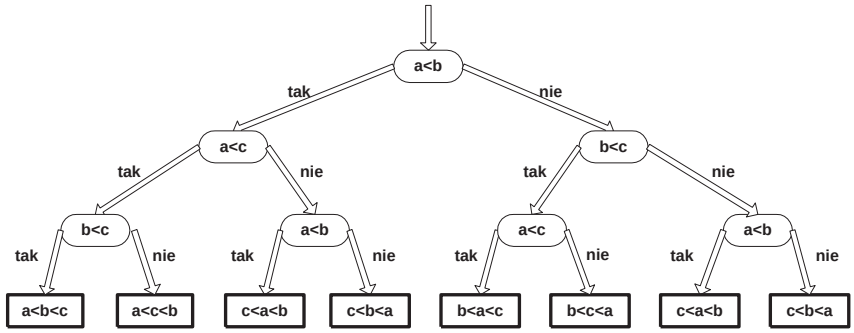
IIUWr. II rok informatyki. Przygotował: Krzysztof Loryś

3 Dolne granice.

Rozważając dolne ograniczenia na złożoność problemu sortowania ograniczymy się, podobnie jak w przypadku problemu jednoczesnego znajdowania minimum i maksimum, do klasy algorytmów, które na elementach ciągu wejściowego wykonują jedynie operacje porównania. Działanie takich algorytmów można w naturalny sposób reprezentować *drzewami decyzyjnymi*. Niezbyt formalnie można je zdefiniować jako skończone drzewa binarne, w których każdy wierzchołek wewnętrzny reprezentuje jakieś porównanie, każdy liść reprezentuje wynik obliczeń a krawędzie odpowiadają obliczeniom wykonywanym przez algorytm pomiędzy kolejnymi porównaniami.

Ponieważ od drzew decyzyjnych wymagamy by były skończone, jedno drzewo nie może reprezentować działania algorytmu dla dowolnych danych. Z reguły przyjmujemy, że algorytm reprezentowany jest przez nieskończoną rodzinę drzew decyzyjnych $\{D_i\}_{i=1}^{\infty}$, gdzie drzewo D_n odpowiada działaniu algorytmu na danych o rozmiarze n.

Przykład
Rysunek 7 przedstawia drzewo decyzyjne odpowiadające działaniu algorytmu *SelectSort* na ciągach 3-elementowych.

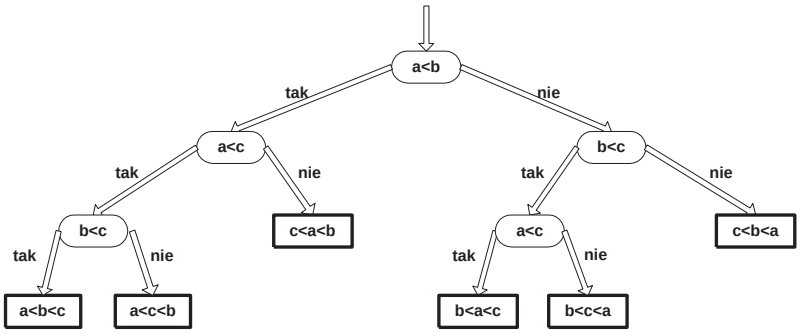


Rysunek 7: *Drzewo D_3 dla algorytmu sortowania przez selekcję.*

□

Jak łatwo zauważyć, algorytm Select Sort wykonuje niektóre porównania niepotrzebnie. Są to porównania "a<b" znajdujące się w odległości 2 od korzenia. Po ich usunięciu otrzymamy inne, mniejsze, drzewo decyzyjne dla sortowania ciągów 3-elementowych (patrz Rysunek 8). Pod względem liczby liści jest ono optymalne.

Fakt 12 *Niech $\mathcal{A}$ będzie algorytmem sortującym, a $\{D_i\}_{i=1}^{\infty}$ – odpowiadającą mu rodziną drzew decyzyjnych. Wówczas drzewo D_n posiada co najmniej $n!$ liści, dla każdego n.*



Rysunek 8: *Optymalne drzewo decyzyjne dla algorytmów sortujących ciągi 3-elementowe.*

UZASADNIENIE: Każda permutacja ciągu wejściowego może być wynikiem, a każdy liść drzewa D_n odpowiada jednemu wynikowi. □

Wprost z Faktu 12 mamy następujące:

Twierdzenie 3 *Niech $\mathcal{A}$ będzie algorytmem sortującym, a $\{D_i\}_{i=1}^{\infty}$ – odpowiadającą mu rodziną drzew decyzyjnych. Wówczas drzewo D_n ma wysokość co najmniej $\Omega(n \log n)$.*

UZASADNIENIE: Drzewo binarne o $n!$ liściach (a takim jest D_n) musi mieć wysokość co najmniej $\log(n!)$. Ze wzoru Stirlinga, $n!$ możemy z dołu oszacować przez $(n/e)^n$, co daje nam:

$$\log n! \geq n\,(\log n - \log e) \geq n \log n - 1.44n$$

□

Ponieważ wysokość drzewa D_n odpowiada liczbie porównań wykonywanych w najgorszym przypadku przez algorytm A dla danych o rozmiarze n, otrzymujemy dolne ograniczenie na złożoność czasową (w najgorszym przypadku) algorytmów sortowania.

Wniosek 1 *Każdy algorytm sortujący za pomocą porównań ciąg n - elementowy wykonuje co najmniej $cn \log n$ porównań dla pewnej stałej $c > 0$.*

3.1 Ograniczenie na średnią złożoność

Działanie algorytmu sortowania, który dane wykorzystuje wyłącznie w porównaniach, zależy jedynie od względnego porządku pomiędzy elementami. W szczególności nie zależy ono od bezwzględnych wartości elementów. Dlatego badając złożoność takich algorytmów możemy ograniczać się do analizy zachowania algorytmu na permutacjach zbioru $\{1, 2, \ldots, n\}$, a średnia złożoność algorytmu na danych rozmiaru n może być policzona jako suma:

$$\sum_{\sigma - permutacja\ zbioru\ \{1,2,\ldots,n\}} P[\sigma]c(\sigma),$$

gdzie $P[\sigma]$ jest prawdopodobieństwem wystąpienia permutacji σ jako danych wejściowych, a $c(\sigma)$ jest równa liczbie porównań wykonywanych na tych danych. W języku drzew decyzyjnych można ją wyrazić jako średnią wysokość drzewa, tj.

$$\sum_{v - liść\ T} p_v d_v,$$

gdzie p_v oznacza prawdopodobieństwo dojścia do liścia v, a d_v - jego głębokość.

Teraz łatwo widać, że dla wielu rozkładów danych średnia złożoność algorytmu także wynosi $\Omega(n \log n)$. Wystarczy bowiem, by istniały stałe c i d takie, że prawdopodobieństwa dojścia do liści znajdujących się na głębokości nie mniejszej niż $cn \log n$ sumują się do wartości nie mniejszej d. W szczególności otrzymujemy:

Twierdzenie 4 *Jeżeli każda permutacja ciągu n-elementowego jest jednakowo prawdopodobna jako dana wejściowa, to wówczas każde drzewo decyzyjne sortujące ciągi n-elementowe ma średnią głębokość co najmniej* $\log n!$.

Uzasadnienie: Na głębokości nie większej niż $\log(n/e)^n - 1$ znajduje się mniej niż $n!/2$ liści. Tak więc co najmniej $n!/2$ liści osiągalnych z prawdopodobieństwem $1/n!$ leży na głębokości większej, co implikuje, że średnia wysokość drzewa decyzynego jest większa niż $(1/n!)(n!/2)\log((n/e)^n)$. □

4 Quicksort

O algorytmie *Quicksort* wspomnieliśmy omawiając strategię dziel i zwyciężaj. Podany tam schemat algorytmu można zapisać w następujący sposób:

```
procedure quicksort(A[1..n], p, r)
    if r − p jest małe  then insert − sort(A[p..r])
    else choosepivot(A, p, r)
         q ← partition(A, p, r)
         quicksort(A, p, q)
         quicksort(A, q + 1, r)
```

Kluczowe znaczenie dla efektywności algorytmu mają wybór *pivota*, tj. elementu dzielącego, dokonywany w procedurze *choosepivot*, oraz implementacja procedury *partition* dokonującej przestawienia elementów tablicy A.

4.1 Implementacja procedury partition

Zakładamy, że w momencie wywołania $partition(A, p, r)$ pivot znajduje się w $A[p]$. Procedura przestawia elementy podtablicy $A[p..r]$ dokonując jej podziału na dwie części: w pierwszej – $A[p..q]$ – znajdą się elementy nie większe od pivota, w drugiej – $A[q + 1, r]$ – elementy nie mniejsze od pivota. Granica tego podziału, wartość q, jest przekazywana jako wynik procedury.

```
procedure partition(A[1..n], p, r)
    x ← A[p]
    i ← p − 1
    j ← r + 1
    while i < j do
        repeat j ← j − 1 until A[j] ≤ x
        repeat i ← i + 1 until A[i] ≥ x
        if i < j then zamień A[i] i A[j] miejscami
                 else return j
```

Fakt 13 *Koszt procedury partition*$(A[1..n], p, r)$ *wynosi* $\Theta(r-p)$.

4.2 Wybór pivota

Istnieje wiele metod wyboru pivota implementowanych w procedurze *quicksort*. Decydując się na którąś z nich musimy dokonać kompromisu między jakością pivota a czasem działania algorytmu. Nierozważne wybory pivotów mogą w skrajnym przypadku prowadzić do takich podziałów tablicy A, w których jedna z podtablic jest jednoelementowa, a to implikuje liniową głębokość rekursji i, w konsekwencji, kwadratowy czas działania procedury *quicksort*.

Wydawać się może, że idealnym pivotem jest mediana[3], ponieważ daje zrównoważone podziały tablicy A, co ogranicza głębokość rekursji do $\log n$. Ponadto istnieją algorytmy wyznaczające medianę w czasie liniowym (poznamy je pó"xniej), więc czas działania procedury *quicksort* wyraża się równaniem $t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + \Theta(n)$, co daje optymalnie asymptotyczny czas $\Theta(n \log n)$. Problem w tym, że stała ukryta pod Θ jest zbyt duża, by taki algorytm był praktyczny.

4.2.1 Prosta metoda deterministyczna

Najprostszą, dość często stosowaną, metodą jest wybór pierwszego elementu tablicy $A[p..r]$ jako elementu dzielącego. W naszym algorytmie sprowadza się ona do pominięcia wywołania *choosepivot*(A, p, r).

Metoda ta oczywiście może prowadzić do nierównomiernych podziałów. W szczególności, czas kwadratowy jest osiągany, gdy dane wejściowe są uporządkowane. Z drugiej strony, na losowych danych algorytm działa bardzo szybko.

4.2.2 Prosty wybór zrandomizowany

Jako pivot obieramy losowy element spośród elementów $A[p..r]$.

[3]Medianą zbioru S nazywamy taki jego element, który jest większy od dokładnie $\lfloor |S|/2 \rfloor$ elementów zbioru S. Definicja w naturalny sposób uogólnia się na wielozbiory.

```
procedure choosepivot(A[1..n], p, r)
    i ← random(p, q);
    zamień A[p] i A[i] miejscami
```

Przy takim wyborze pivota również może się zdarzyć, że algorytm będzie działać w czasie kwadratowym, jednak prawdopodobieństwo takiego zdarzenia jest zaniedbywalnie małe.

Zasadnicza różnica w stosunku do metody deterministycznej polega na tym, że teraz przebieg algorytmu zależy nie tylko od danych wejściowych, ale także od generatora liczb losowych (pseudolosowych). W szczególności teraz nie istnieją dane wejściowe lepsze i gorsze. Na każdych algorytm może działać jednakowo szybko i na każdych może się zdarzyć, że będzie działać w czasie kwadratowym.

4.2.3 Mediana z małej próbki

Często stosowaną metodą jest wybieranie jako pivota mediany z trzech losowo wybranych elementów tablicy. To prowadzi do istotnego zmniejszenia prawdopodobieństwa nierównomiernych podziałów. Ceną jest konieczność wykonania dwóch dodatkowych porównań i przede wszystkim dwóch dodatkowych wywołań generatora liczb losowych.

"Medianę z trzech" stosuje się także w wersji deterministycznej. Najczęściej wybiera się ją wówczas spośród pierwszego, środkowego i ostaniego elementu tablicy.

Eksperymentalnie stwierdzono, że zastosowanie "mediany z trzech' zamiast prostego wyboru pivota prowadzi do przyspieszenia *quicksortu* o kilka do kilkunastu procent (zależnie od zastosowanej wersji wyboru elementów i sprawności implementacyjnej przeprowadzającego eksperymenty).

Metodę tę można rozszerzać na liczniejsze próbki, jednak uzyskane zyski czasowe są znikome.

4.3 Średni koszt algorytmu

Załóżmy, że jako pivot wybierany jest z jednakowym prawdopodobieństwem dowolny element tablicy. Pokażemy, że przy tym założeniu średni koszt algorytmu *quicksort* wynosi $\Theta(n \log n)$. Dla uproszczenia analizy założymy ponadto, że wszystkie elementy sortowanej tablicy są różne.

Niech $n = r - p + 1$ oznacza liczbę elementów w $A[p..r]$ i niech

$$rank(x, A[p..r]) \stackrel{df}{=} |\{j : p \leq j \leq r \text{ i } A[j] \leq x\}| .$$

Ponieważ w momencie wywoływania procedury *partition* w $A[p]$ znajduje się losowy element z $A[p..r]$, więc wówczas

$$\forall_{i=1,..,n} \quad \Pr[rank(A[p], A[p..r]) = i] = \frac{1}{n}.$$

Wynik procedury *partition* w oczywisty sposób zależy od wartości $rank(A[p], A[p..r])$. Gdy jest ona równa i (dla $i = 2, \ldots, n$), wynikiem *partition* jest $p + i - 2$. Ponadto, gdy $rank(A[p], A[p..r]) = 1$, wynikiem jest p. Tak więc zmienna q z procedury *quicksort* przyjmuje wartość p z prawdopodobieństwem $2/n$, a każdą z pozostałych wartości (tj. $p + 1, p + 2, \ldots, r - 1$) z prawdopodobieństwem $1/n$. Stąd oczekiwany czas działania procedury *quicksort* wyraża się równaniem

$$\begin{cases} T(1) = & 1 \\ T(n) = & \frac{1}{n}\left[(T(1) + T(n-1)) + \sum_{d=1}^{n-1}(T(d) + T(n-d))\right] + \Theta(n) \end{cases}$$

Zmienna $d = q - p + 1$ oznacza długość pierwszej z podtablic.

Ponieważ $T(1) = \Theta(1)$ a $T(n-1)$ w najgorszym przypadku jest równe $\Theta(n^2)$, więc

$$\frac{1}{n}(T(1) + T(n-1)) = O(n).$$

To pozwala nam pominąć ten składnik, ponieważ będzie on uwzględniony w ostatnim członie sumy. Tak więc:

$$T(n) = \frac{1}{n}\sum_{d=1}^{n-1}(T(d) + T(n-d)) + \Theta(n).$$

W tej sumie każdy element $T(k)$ jest dodawany dwukrotnie (np. $T(1)$ raz dla $q = 1$ i raz dla $q = n - 1$), więc możemy napisać:

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \tag{1}$$

Ponieważ mamy silne przesłanki, by przypuszczać, że rozwiązanie tego równania jest rzędu $\Theta(n \log n)$, ograniczymy się do sprawdzenia tego faktu. Niech

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \leq an \log n + b$$

dla pewnych stałych $a, b > 0$. Naszym zadaniem jest pokazanie, że takie stałe a i b istnieją.

Bierzemy b wystarczająco duże by $T(1) \leq b$. Dla $n > 1$ mamy:

$$T(n) \quad = \quad \frac{2}{n}\sum_{k=1}^{n-1}(ak \log k + b) + \Theta(n) \leq \frac{2a}{n}\sum_{k=1}^{n-1} k \log k + \frac{2b}{n}(n-1) + \Theta(n)$$

Proste oszacowanie $\sum_{k=1}^{n-1} k \log k$ przez $\frac{1}{2}n^2 \log n$ nie prowadzi do celu, ponieważ musimy pozbyć się składnika $\Theta(n)$. Oszacujmy więc $\sum_{k=1}^{n-1} k \log k$ nieco staranniej:

Fakt 14 $\sum_{k=1}^{n-1} k \log k \leq \frac{1}{2}n^2 \log n - \frac{1}{8}n^2$

Dowód. Rozbijamy sumę na dwie części:

$$\sum_{k=1}^{n-1} k \log k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \log k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \log k$$

Szacując $\log k$ przez $\log \frac{n}{2}$ dla $k < \lceil \frac{n}{2} \rceil$ oraz przez $\log n$ dla $k \geq \lceil \frac{n}{2} \rceil$, otrzymujemy:

$$\sum_{k=1}^{n-1} k \log k \leq ((\log n) - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k = \log n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \leq$$
$$\frac{1}{2} n(n-1) \log n - \frac{1}{2}(\frac{n}{2} - 1)\frac{n}{2} \leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$$

□

Teraz możemy napisać

$$\frac{2a}{n}\left(\frac{1}{2} n^2 \log n - \frac{1}{8} n^2\right) + \frac{2b}{n}(n-1) + \Theta(n) \leq an \log n - \frac{a}{4} n + 2b + \Theta(n) =$$
$$an \log n + b + \left(\Theta(n) + b - \frac{a}{4} n\right)$$

Składową $(\Theta(n) + b - \frac{a}{4} n)$ możemy pominąć, dobierając a tak, by $\frac{a}{4} n \geq \Theta(n) + b$. Zauważmy, że taki dobór zależy jedynie od stałej b oraz od stałej ukrytej pod Θ, a więc za a można przyjąć odpowiednio dużą stałą.

To kończy sprawdzenie, że $T(n) \leq an \log n + b$ dla pewnych stałych $a, b > 0$.

4.4 Inne usprawnienia

Quicksort jest dość powszechnie uważany za najszybszą (a przynajmniej jedną z najszybszych) metodę sortowania. Jego znaczenie spowodowało, że wiele wysiłku włożono w opracowanie modyfikacji, mających na celu uzyskanie jak największej efektywności. Poniżej wymieniamy kilka z nich:

- Trójpodział. W przypadku, gdy spodziewamy się, że sortowane klucze mogą się wielokrotnie powtarzać (np. gdy przestrzeń kluczy jest mała), opłacalne może być zmodyfikowanie procedury *partition* tak, by dawała podział na trzy części: elementy mniejsze od pivota, równe pivotowi i większe od pivota. Oczywiście *quicksort* jest rekurencyjnie wywoływany jedynie do pierwszej i trzeciej części. W przypadku, gdy liczba elementów równych pivotowi jest znaczna, może to przynieść istotne przyspieszenie.

- Eliminacja rekursji.

  - Tak jak w przypadku wszystkich algorytmów opartych na strategii dziel i zwyciężaj, spory zysk można otrzymać, starannie dobierając próg na rozmiar danych, poniżej którego opłaca się zastosować prosty algorytm nierekurencyjny w miejsce rekurencyjnych wywołań procedury *quicksort*.
  - W wielu implementacjach *quicksortu* przeznaczonych do powszechnego użytku (np. w bibliotekach procedur) w ogóle wyeliminowano rekursję.

- Optymalizacja pętli wewnętrznej, aż do zapisania jej w języku wewnętrznym procesora.

- W zastosowaniach, w których krytycznym zasobem jest pamięć (np. w układach realizujących sortowanie hardware'owo), stosowana bywa nierekurencyjna wersja (rekursja wymaga pamięci na stos wywołań) działająca "w miejscu", a więc wykorzystująca co najwyżej $O(1)$ komórek pamięci poza tymi, które zajmuje sortowany ciąg.

Notatki z AiSD. Nr ??. 16 marca 2005

SORTOWANIE C.D.

IIUWr. II rok informatyki. Przygotował: Krzysztof Loryś

Na dzisiejszym wykładzie poznamy algorytmy, sortujące w czasie niższym niż wynika to z dolnego ograniczenia poznanego na poprzednim wykładzie. Jest to moźliwe z dwóch powodów. Po pierwsze algorytmy te zakładają pewne ograniczenia na postać danych, a po drugie wykonują one na sortowanych elementach operacje inne niż porównania.

5 Counting Sort

POSTAĆ DANYCH: ciąg $A[1..n]$ liczb całkowitych z przedziału $\langle 1, k\rangle$.

IDEA: $\forall_{x\in A[1..n]}$ obliczyć liczbę $c[x] = |\{y \ : \ y \in A[1..n] \ \& \ y \leq x\}|$.

```
procedure Counting − Sort(A[1..n], k, varB[1..n])
    for i ← 1 to k do c[i] ← 0
    for j ← 1 to n do c[A[j]] ← c[A[j]] + 1
    for i ← 2 to k do c[i] ← c[i] + c[i − 1]
    for j ← n downto 1 do  B[c[A[j]]] ← A[j]
                           c[A[j]] ← c[A[j]] − 1
```

UWAGA: W oczywisty sposób powyższa procedura może być zmodyfikowana do sortowania rekordów, w których klucz $A[j]$ jest jednym z wielu pól.

Definicja 10 *Metodę sortowania nazywamy* stabilną*, jeśli w ciągu wyjściowym elementy o tej samej wartości klucza pozostają w takim samym porządku względem siebie w jakim znajdowały się w ciągu wejściowym.*

Fakt 15 *Counting − sort jest metodą stabilną.*

KOSZT: $\Theta(n + k)$.

6 Sortowanie kubełkowe (bucket sort).

POSTAĆ DANYCH: Ciąg $A[1..n]$ liczb rzeczywistych z przedziału $\langle 0, 1)$ wygenerowany przez generator liczb losowych o rozkładzie jednostajnym.

IDEA: Podzielić przedział $\langle 0, 1)$ na n odcinków ("kubełków") jednakowej długości; umieścić liczby w odpowiadających im kubełkach; posortować poszczególne kubełki; połączyć kubełki.

```
procedure bucket − sort(A[1..n])
    for i ← 0 to n − 1 do B[i] ← ∅
    for i ← 1 to n do  dołącz A[i] do listy B[⌊nA[i]⌋]
    for i ← 0 to n − 1 do  posortuj procedurą insert − sort listę B[i]
    połącz listy B[0], B[1], ..., B[n − 1]
```

KOSZT: Oczekiwany czas działania: $\Theta(n)$.

7 Sortowanie leksykograficzne ciągów jednakowej długości (radix sort).

Definicja 11 *Niech S - zbiór uporządkowany liniowo oraz $s_1, \ldots, s_p, t_1, \ldots, t_q \in S$.*

$$(s_1, \ldots, s_p) \leq (t_1, \ldots, t_q) \stackrel{df}{\iff} \begin{array}{ll} (1) & \exists_{1 \leq j \leq \min(p,q)}\ s_j < t_j\ \&\ \forall_{i<j}\ s_i = t_i \\ \textit{albo} & \\ (2) & p \leq q\ \&\ \forall_{1 \leq i \leq p}\ s_i = t_i. \end{array}$$

POSTAĆ DANYCH: $A_1, \ldots, A_n$ - elementy $\{0, \ldots, k-1\}^d$.

```
procedure radix − sort(A_1, .., A_n)
    for i ← d downto 1 do
        metodą stabilną posortuj ciągi wg i−tego elementu
```

KOSZT: Jeśli w procedurze $Radix - sort$ zastosujemy $counting - sort$, to jej koszt wyniesie $O((n+k)d)$. Jest to koszt liniowy, gdy $k = O(n)$.

8 Sortowanie leksykograficzne ciągów niejednakowej długości.

POSTAĆ DANYCH: $A_1, \ldots, A_n$ ciągi liczb całkowitych $\in \langle 0, .., k-1 \rangle$.

Niech l_i-długość A_i, $l_{max} = \max\{l_i\ :\ i = 1, .., n\}$.

8.1 Pierwszy sposób

IDEA: Uzupełnić ciągi specjalnym elementem (mniejszym od każdego elementu z S), tak by miały jednakową długość i zastosować algorytm z poprzedniego punktu.

KOSZT: $\Theta((n+k) \cdot l_{max})$.

UWAGA: Jest to metoda nieefektywna, gdy ciągów długich jest niewiele.

8.2 Drugi sposób

IDEA:

for $i \leftarrow l_{max}$ **downto** 1 **do**
metodą stabilną posortuj ciągi o długości $\geq i$ wg i-tej składowej

ALGORYTM:

1. Utwórz listy $nonempty[l]$ $(l = 1, .., l_{max})$ takie, że
   - $x \in nonempty[l]$ iff x jest $l-$tą składową jakiegoś ciągu A_i.
   - $nonempty[l]$ jest uporządkowana niemalejąco.
2. Utwórz listy $length[l]$ $(l = 1, .., l_{max})$ takie, że $length[l]$ zawiera wszystkie ciągi A_i o długości l.
3. $queue \leftarrow \emptyset$
   for $j \leftarrow 0$ **to** $k-1$ **do** $q[j] \leftarrow \emptyset$
   for $l \leftarrow l_{max}$ **downto** 1 **do**
   $queue \leftarrow concat(length[l], queue)$
   while $queue \neq \emptyset$ **do**
   $Y \leftarrow$ pierwszy ciąg z $queue$
   $queue \leftarrow queue \setminus \{Y\}$
   $a \leftarrow l$-ta składowa ciągu Y
   $q[a] \leftarrow concat(q[a], \{Y\})$
   for each $j \leftarrow nonempty[l]$ **do**
   $queue \leftarrow concat(queue, q[j])$
   $q[j] \leftarrow \emptyset$

Operacja $concat(K_1, K_2)$ dołącza kolejkę K_2 do końca kolejki K_1.

Twierdzenie 5 *Powyższy algorytm można zaimplementować tak, by działał w czasie* $O(k + \sum_{i=1}^{n} l_i)$.

UZASADNIENIE: Niech $l_{total} = \sum_{i=1}^{n} l_i$.
Jedynym niezupełnie trywialnym krokiem jest tworzenie list *nonempty*:

- tworzymy w czasie $O(l_{total})$ ciąg S zawierający wszystkie pary $\langle l, a \rangle$, takie, że a jest l-tą składową jakiegoś $A[i]$;
- sortujemy leksykograficznie w czasie $O(k + l_{total})$ ciąg S;
- przeglądając S z lewa na prawo tworzymy $O(l_{total})$ listy *nonempty*.

Krok 2 wymaga czasu $O(l_{total})$.

Wewnętrzna pętla while działa w czasie proporcjonalnym do sumarycznej (po wszystkich iteracjach pętli zewnętrznej) długości kolejek *queue*. Ponieważ w l-tej iteracji *queue* ma długość równą liczbie ciągów co najmniej l-elementowych, więc koszt while jest $O(l_{total})$.

Wewnętrzna pętla for działa w czasie proporcjonalnym do sumarycznej (po wszystkich iteracjach pętli zewnętrznej) długości list *nonempty*. Ponieważ w każdej iteracji *nonempty* jest nie dłuższa od *queue*, czas pętli for jest również $O(l_{total})$. □

8.3 Przykład zastosowania

PROBLEM:
Dane: T_1, T_2 -drzewa o ustalonych korzeniach,
Zadanie: sprawdzić czy T_1 i T_2 są izomorficzne.

IDEA: Wędrując przez wszystkie poziomy (począwszy od najniższego) sprawdzamy czy na każdym poziomie obydwa drzewa zawierają taką samą liczbę wierzchołków tego samego typu. (Wierzchołki są tego samego typu, jeśli poddrzewa w nich zakorzenione są izomorficzne.)

ALGORYTM:
Bez zmniejszenia ogólności możemy założyć, że obydwa drzewa mają tę samą wysokość.

1. $\forall_{v \,-\, li\ w\ T_i}\ kod(v) \leftarrow 0$
2. **for** $j \leftarrow depth(T_1)$ **downto** 1 **do**
3. $S_i \leftarrow$ zbiór wierzchołków T_i z poziomu j nie będących liśćmi
4. $\forall_{v \in S_i}\ key(v) \leftarrow$ wektor $\langle i_1, \ldots, i_k \rangle$, taki że
   - $i_1 \leq i_2 \leq \ldots \leq i_k$
   - v ma k synów $u_1, \ldots, u_k$ i $i_l = kod(u_l)$
5. $L_i \leftarrow$ lista wierzchołków z S_i posortowana leksykograficznie według wartości key
6. $L_i^{'} \leftarrow$ otrzymany w ten sposób uporządkowany ciąg wektorów
7. **if** $L_1^{'} \neq L_2^{'}$ **then return** ("nieizomorficzne")
8. $\forall_{v \in L_i}\ kod(v) \leftarrow 1 + rank(key(v), \{key(u) \mid u \in L_i\})$
9. Na początek L_i dołącz wszystkie liście z poziomu j drzewa T_i
10. **return** ("izomorficzne")

Twierdzenie 6 *Izomorfizm dwóch drzew o n wierzchołkach może być sprawdzony w czasie* $O(n)$.

Notatki z AiSD. Nr 10. 16 marca 2005

Kopiec

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

9 Definicja

Definicja 12 *Niech T będzie drzewem binarnym o wysokości d, którego wierzchołki zawierają klucze z liniowo uporządkowanego zbioru. Drzewo T nazywamy* kopcem *iff T spełnia następujące warunki:*

- *Struktura drzewa*
  - *wszystkie jego liście znajdują się na głębokości d lub $d-1$;*
  - *wszystkie liście z poziomu $d-1$ leżą na prawo od wszystkich wierzchołków wewnętrznych z tego poziomu;*
  - *położony najbardziej na prawo wierzchołek wewnętrzny z poziomu $d-1$ jest jedynym wierzchołkiem wewnętrznym w T, który może mieć jednego syna (co implikuje, że pozostałe wierzchołki wewnętrzne mają po dwóch synów);*
- *Uporządkowanie*
  - *klucz w każdym wierzchołku wewnętrznym jest nie mniejszy od kluczy w jego potomkach.*

Warunki określające strukturę kopca mogą się wydać nieco skomplikowane. W rzeczywisctości mówią one, że dobrą strukturę mają drzewa binarne powstałe przez dopisywanie do początkowo pustego drzewa wierzchołków do kolejnych poziomów drzewa, zapełniając każdy poziom od lewej strony do prawej strony .

10 Implementacja kopców

Kopce w bardzo efektywny sposób mogą być pamiętane w tablicach. Do pamiętania kopca n-elementowego używamy n-elementowej tablicy K:

- korzeń kopca pamiętany jest w $K[1]$,
- lewy syn korzenia pamiętany jest w $K[2]$, prawy syn korzenia - w $K[3]$, itd ... .

Uogólniając: wierzchołki z poziomu k-tego pamiętane są kolejno od lewej do prawej w $K[2^k], K[2^k+1], \ldots, K[2^{k+1}-1]$.

Fakt 16 *Ojciec wierzchołka pamiętanego w $K[i]$ znajduje się w $K[i$ **div** $2]$ zaś jego dzieci (o ile istnieją) w $K[2i]$ i $K[2i+1]$.*

10.1 Ważniejsze procedury

10.1.1 Procedury przywracające tablicy K własności kopca

Zmiana klucza w wierzchołku kopca może spowodować zaburzenie własności (2). Jeśli nowy klucz jest większy od starego, należy sprawdzić, czy nie jest on większy także od klucza znajdującego się w ojcu. Jeśli tak jest, możemy zamienić miejscami te klucze i sprawdzić czy zaburzenie nie przeniosło się poziom wyżej. Jeśli nowy klucz jest mniejszy od starego, to możemy zamienić go z większym z kluczy znajdujących w jego synach, a następnie sprawdzić czy zaburzenie mnie przeniosło się na niższy poziom.

```
procedure zmień − element(K[1..n], i, u)
    x ← K[i]
    K[i] ← u
    if u < x   then przesuń − niżej(K, i)
               else przesuń − wyżej(K, i)


procedure przesuń − niżej(K[1..n], i)
    k ← i
    repeat
        j ← k
        if 2j ≤ n   and K[2j] > K[k]   then k ← 2j
        if 2j < n   and K[2j + 1] > K[k]   then k ← 2j + 1
        K[j] ↔ K[k]
    until j = k


procedure przesuń − wyżej(K[1..n], i)
    k ← i
    repeat
        j ← k
        if j > 1   and K[j div 2] < K[k]   then k ← j div 2
        K[j] ↔ K[k]
    until j = k
```

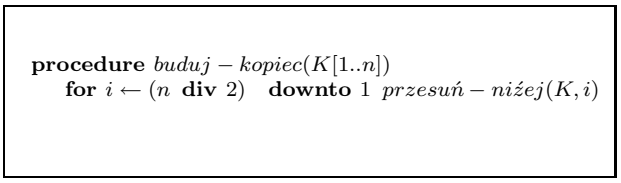
10.1.2 Procedura budująca kopiec

Kopiec można budować na wiele sposobów. Można np. zacząć od tablicy jednoelementowej, a następnie dodawać na jej koniec po jednym elemencie, za każdym razem używając procedury *przesuń-wyżej* do przywrócenia własności (2). Ten sposób realizuje poniższa procedura.

```
procedure wolna − budowa − kopca(K[1..n])
    for i ← 2   to n  przesuń − wyżej(K, i)
```

Łatwo sprawdzić, że ta procedura może wymagać czasu $n \log n$, a więc takiego samego jak sortowanie np. metodą *quicksort*, dając jednak znacznie mniej uporządkowaną strukturę.

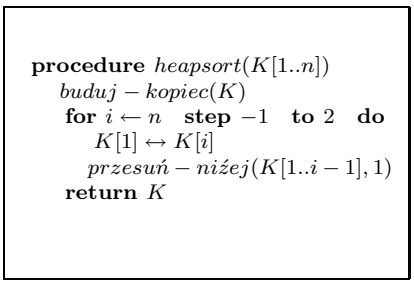
Inna metoda polega na budowaniu kopca od dołu. Startujemy od kopców 1-elementowych. Następnie używamy tych kopców oraz nowych elementów do utworzenia kopców 3-elementowych: nowy element umieszczamy w korzeniu takiego kopca, a jego synami czynimy korzenie kopców 1-elementowych; następnie używamy procedury *przesuń-niżej* do przywrócenia własności (2). W analogiczny sposób, dla dowolnego k, tworzymy z dwóch kopców $(2^k - 1)$-elementowych i jednego nowego elementu kopiec $(2^{k+1} - 1)$-elementowy.



Twierdzenie 7 *Procedura buduj − kopiec tworzy kopiec w czasie* $O(n)$.

11 Zastosowania kopców

11.1 HEAPSORT - sortowanie przy użyciu kopca



Twierdzenie 8 *Algorytm heapsort działa w czasie* $O(n \log n)$.

11.1.1 Przyspieszenie *heapsortu*

Po usunięciu maksimum na szczycie kopca powstaje dziura, w którą *heapsort* wstawia element z dołu kopca. Element taki jest, z dużym prawdopodobieństwem, mały i zostanie przez procedurę *przesuń-niżej* zsunięty z powrotem nisko. Przesuwając go o jeden poziom w dół *przesuń-niżej* wykonuje dwa porównania. Tak więc z dużym prawdopodobieństwem potrzeba będzie 2·`wysokość kopca` porównań na przywrócenie własności kopca.

Można postępować nieco oszczędniej. Otóż można najpierw przesunąć dziurę na dół kopca, następnie wstawić w nią ostatni element kopca i używając procedury *przesuń-wyżej* znale"xć dla niego odpowiednie miejsce w kopcu. Oszczędność wynika z tego, że na przesunięcie dziury o jedno miejsce w dół potrzeba tylko jednego porównania oraz z tego, że w średnim przypadku *przesuń-wyżej* będzie przesuwać element o nie więcej niż 2 poziomy w górę.

11.2 Kolejka priorytetowa

Kolejka priorytetowa jest strukturą danych przeznaczoną do pamiętania zbioru S (elementów z jakiegoś uporządkowanego uniwersum) i wykonywania operacji wstawiania elementów do S oraz znajdowania i usuwania największego elementu z S.

Wprost idealnie do implementacji kolejek priorytetowych nadają się kopce.

11.2.1 Procedury realizujące operacje kolejki priorytetowej

```
function find − max(K[1..n])
    return K[1]

procedure delete − max(K[1..n])
   K[1] ← K[n]
   przesuń − niżej(K[1..n − 1], 1)

procedure insert − node(K[1..n], v)
   K[n + 1] ← v
   przesuń − wyżej(K[1..n + 1], n + 1)
```

11.3 Podwójna kolejka priorytetowa

Podwójna kolejka priorytetowa umożliwia wykonywanie operacji znajdowania i usuwania zarówno maksymalnego jak i minimalnego elementu.

Prosta implementacja takiej kolejki mogłaby polegać na wykorzystaniu dwóch kopców: jeden z nich byłby uporządkowany malejąco, a drugi - rosnąco. Każdy element kolejki umieszczany byłby w obydwu kopcach. Na użytek operacji *deletemin* i *deletemax* należałoby powiązać ze sobą elementy kopców pamiętające ten sam element kolejki. Zasadniczym mankamentem takiego rozwiązania jest nieoszczędność pamięci. Poniżej przedstawiamy rozwiązanie wolne od tej wady.

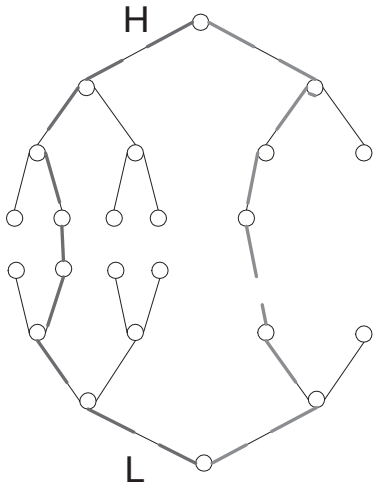
Idea rozwiązania:

1. Wykorzystujemy dwa kopce: L i H.
2. W kopcu H pamiętamy $\lceil n/2 \rceil$ elementów, a w kopcu L - $\lfloor n/2 \rfloor$ elementów (n oznacza liczbę elementów kolejki).
3. Kopiec L uporządkowany jest malejąco, a H - rosnąco.
4. Każdy element kopca L jest nie większy od odpowiadającego mu pozycją elementu z kopca H.

W kopcach H i L w naturalny sposób zdefiniowane są ścieżki biegnące od korzenia kopca L do korzenia kopca H (patrz Rysunek 9). O dwóch wierzchołkach, nazwijmy je u

i v, będziemy mówić, że u poprzedza v (jest poprzednikiem v), jeśli u leży bezpośrednio przed v na ścieżce od korzenia L do v. Łatwo sprawdzić, że spełniona jest następująca własność:

> Na każdej ścieżce obiegnącej od korzenia L do korzenia H klucze uporządkowane są niemalejąco.



Rysunek 9: *Dwie przykładowe ścieżki łaćzące korzenie kopców.*

11.3.1 Implementacja operacji *insert(x)*

Jeśli kolejka zawiera parzystą liczbę elementów, x wstawiamy do kopca H. W przeciwnym razie x wstawiamy do kopca L. Wstawienia dokonujemy w zwykły sposób, tj. wstawiając element do pierwszego wolnego liścia. Teraz jednak, zanim użyjemy procedury *przesuń-wyżej*, musimy sprawdzić, w którą stronę należy przesunąć wstawiony element: czy w stronę korzenia kopca H, czy też w stronę korzenia kopca L. Załóżmy, że x został wstawiony do H (przypadek wstawienia do L jest symetryczny) . Niech y będzie poprzednikiem x-a ze ścieżki prowadzącej do korzenia L (oczywiście y jest liściem kopca L). Porównujemy x i y. Jeśli $x < y$, przestawiamy te elementy a następnie procedurą *przesuń-wyżej* przesuwamy x w odpowiednie miejsce w kierunku korzenia L. Zauważmy, że przesunięcie y-ka z kopca L do H nie zaburzyło porządku w tym kopcu. Jeśli $x \geq y$, używamy procedury *przesuń-wyżej* do ewentualnego naprawienia porządku w kopcu H.

11.3.2 Implementacja operacji *deletemin*

Usuwamy element z korzenia kopca L. W jego miejsce wstawiamy y - ostatni element kopca H (jeśli przed operacją kopce L i H były równoliczne) albo ostatni element kopca L (jeśli przed operacją L zawierał o jeden element więcej niż H). Następnie procedurą *przesuń-niżej* przywracamy porządek w kopcu L. Jeśli procedura *przesuń-niżej* przesunęła y na sam dół kopca L, wówczas porównujemy y z odpowiednim (znajdującym się na tej samej pozycji)

elementem kopca H. Jeśli y jest od niego większy zamieniamy te elementy miejscami, a następnie procedurą *przesuń-wyżej* przesuwamy y na odpowiednie miejsce kopca H.

Pokażemy teraz, że po wykonaniu powyżej opisanej procedury, kopce L i H spełniają warunek 4 (pozostałe warunki spełnione są w oczywisty sposób).

Fakt 17 *Po wykonaniu operacji* deletemin *każdy element kopca L jest nie większy od znajdującego się na tej samej pozycji elementu kopca H.*

Dowód Załóżmy, że procedura *przesuń-wyżej* przesuwa y do liścia kopca L. Niech $l_1, \ldots, l_k$ będą elementami znajdującymi się na ścieżce od korzenia kopca L do tego liścia, a $h_1, \ldots, h_k$ niech będą odpowiadającymi im elementami kopca H. Z faktu, że L i H są kopcami wiemy, że $l_1 \leq l_2 \leq \ldots \leq l_k$ oraz $h_k \leq h_{k-1} \leq \ldots \leq h_1$. Ponadto z własności 4 mamy $l_k \leq h_k$. Tak więc elementy $l_1, \ldots, l_k, h_k, \ldots, h_1$ tworzą ciąg niemalejący. Łatwo sprawdzić, że procedura *delemin* usuwa l_1 z tego ciągu, a następnie wstawia do niego w odpowiednie miejsce element y tak, że ciąg pozostaje uporządkowany. Stąd w oczywisty sposób wynika, że własność 4 zostanie zachowana.

W ten sam sposób dowodzimy przypadku, gdy y nie zostanie przesunięty na dno kopca L. □

11.3.3 Implementacja operacji *deletemax*

Analogicznie do operacji *deletemin*.

11.3.4 Uwagi końcowe

W literaturze można znale"xć wiele implementacji kolejek podwójnych opartych na strukturze kopców. Implementacja podana przez nas oparta jest na symetrycznych kopcach minimaksowych przedstawionych w [1]. Inne metody można znale"xć np. w [2], [3] i [4].

Literatura

[1] A. Arvind, C. Pandu Rangan, Symmetric Min-Max heap: A simpler data structure for double-ended priority queue, *Inform. Process. Lett.*, 69(1999), 197-199.

[2] M.D. Atkinson, J.-R. Sack, T. Strothotte, Min-Max heaps and generalized priority queues, *Comm.ACM*, 29(1986), 996-1000.

[3] S. Carlsson, The Deap - a double-ended heap to implement double-ended priority queues, *Inform. Process. Lett.*, 26(1987), 33-36.

[4] S.C. Chang, M.W. Du, Diamond deque: A simple data structure for priority deques, *Inform. Process. Lett.*, 46(1993), 231-237.

Notatki z AiSD. Nr 18. 16 marca 2005

Wybór k-tego elementu

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

12 Definicja problemu

Dane: $T[1..n]$ - ciąg elementów należących do zbioru liniowo uporządkowanego
k - liczba z przedziału $\langle 1, n\rangle$.
Wynik: k-ty co do wielkości element ciągu T

Założenie (nie zmniejszające ogólności): wszystkie elementy w T są różne.

Model obliczeń: na elementach ciągu T dokonujemy jedynie porównań.

13 Szczególne przypadki

- $k = 1$.
  Konieczna i wystarczająca liczba porównań $= n - 1$.

- $k = 2$.

  Twierdzenie 1 *W tym przypadku potrzeba i wystarcza* $n - 2 + \lceil \log n \rceil$ *porównań.*

  Dowód(szkic)
  $\Rightarrow$
  Konstruujemy algorytm, działający w $\lceil \log n \rceil$ rundach: w 1-szej rundzie porównywane są pary elementów $\langle T[2k-1], T[2k]\rangle$ (dla $k = 1, \lfloor \frac{n}{2} \rfloor$). "Zwycięzcy" tych porównań (oraz element $T[n]$ - w przypadku nieparzystego n) przechodzą do następnej rundy. W kolejnych rundach postępujemy analogicznie.
  Oczywiście ostatnia runda wyznaczy element największy w T. Do tej pory algorytm wykona $n - 1$ porównań. Można to łatwo udowodnić, jeśli na działanie tego algorytmu popatrzeć jako na drzewo binarne o n liściach: liście tego drzewa etykietujemy elementami $T[i]$, a każdy wierzchołek wewnętrzny - większą spośród etykiet jego synów. Tak więc wierzchołki wewnętrzne odpowiadają porównaniom wykonanym przez algorytm.
  Znalezienie 2-ego co do wielkości elementu sprowadza się teraz do znalezienia największego elementu pośród tych, które były porównywane z elementem największym. Ponieważ elementów tych jest $\lceil \log n \rceil$, wystarczy teraz $\lceil \log n \rceil - 1$ porównań.
  $\Leftarrow$
  Rozważmy dowolny algorytm $\mathcal{A}$ wyznaczający 2-gi co wielkości element. Zauważmy, że $\mathcal{A}$ wyznacza jednocześnie element największy. Niech bowiem $X = T[j]$ będzie rozwiązaniem podanym przez $\mathcal{A}$. Elementem największym jest ten, z którym X "przegrał" porównanie. Element taki musi istnieć (w przeciwnym razie $\mathcal{A}$ podałby

$T[j]$ jako rozwiązanie także dla takich danych, w których $T[j]$ zwiększylibyśmy dowolnie, a pozostałe elementy pozostawilibyśmy bez zmian) i to dokładnie jeden (w przeciwnym razie X nie byłby drugim elementem). Tak więc $\mathcal{A}$ musi wykonać $n-1$ porównań, by wyznaczyć element największy. Porównania te nie wnoszą żadnej informacji o wzajemnej relacji pomiędzy elementami, które jedynie z nim przegrały porównanie (a X jest największym z tych elementów). Tak więc nasz dowód sprowadza się do pokazania, że w najgorszym przypadku element największy bierze udział w co najmniej $\lceil \log n \rceil$ porównaniach wykonywanych przez $\mathcal{A}$. To będzie treścią zadania na ćwiczenia (rozwiązanie możesz znale"xć w ([7], s.212).

14 Przypadek ogólny

14.1 Algorytm deterministyczny

IDEA Stosujemy metodę Dziel i Rząd"x. Rozdzielamy T na dwa podzbiory: U i $V = T \backslash U$, takie że wszystkie elementy U są mniejsze od wszystkich elementów V. Teraz porównanie k z mocą U pozwala określić, w którym ze zbiorów znajduje się szukany element. W ten sposób redukujemy problem do problemu szukania elementu w zbiorze mniejszym.

Procedure *SELECTION(k,T)*

1. **if** $|T|$ małe **then** $sort(T)$ & **return** $(T[k])$
2. $p \leftarrow$ jakiś element z T
3. $U \leftarrow$ elementy T mniejsze od p
4. **if** $k \leq |U|$ **then return** ($SELECTION(k, U)$)
   else return $(SELECTION(k - |U|, T \setminus U))$

UWAGA: W powyższej procedurze zbiory T i U wygodnie jest pamiętać w tablicach.

Aby algorytm był efektywny, musimy zagwarantować, że zbiory U i $T \setminus U$ są istotnie mniej liczne od zbioru T. W tym celu zbiór T dzielimy na 5-cioelementowe grupy. W każdej grupie wybieramy medianę (tj. środkowy element) a następnie rekurencyjnie wybieramy medianę tych median.

function *Pseudomed(T)*

1. Podziel T na rozłączne podzbiory 5-cioelementowe C_j $(j = 1, \ldots, \lceil |T|/5 \rceil)$
   {jeśli $|T|$ nie dzieli się przez 5, to $C_{\lceil |T|/5 \rceil}$ zawiera mniej niż 5 elementów }
2. **for** $i \leftarrow 1$ **to** $\lceil |T|/5 \rceil$ **do** $s_i \leftarrow adhocmed(C_i)$
3. $S \leftarrow \{s_i \mid i = 1, \ldots, \lceil |T|/5 \rceil\}$
4. **return** $(SELECTION(\lceil \frac{|S|}{2} \rceil, S))$

Twierdzenie 2 *Jeśli w procedurze SELECTION wybór elementu p dokonywany jest funkcją PSEUDOMED, to procedura SELECTION wyznacza k−ty element ciągu T w czasie $O(n)$.*

Pomijamy dowód poprawności procedury *Selection*. W oszacowaniu jej kosztu kluczowym punktem jest następujący lemat:

Lemat 1 *Jeśli element p został wybrany funkcją PSEUDOMED, to każdy ze zbiorów U i $T \setminus U$ zawiera nie mniej niż $\frac{3}{10}n - 4$ elementy.*

Dowód: Istnieje co najmniej (a przy założeniu, że wszystkie elementy w T są różne - dokładnie) $\frac{1}{2}\lceil n/5 \rceil$ grup, których mediana jest nie mniejsza od p. W każdej z tych grup, poza tą, która zawiera największą z median s_i znajdują się co najmniej 3 elementy nie mniejsze od p (takimi są na pewno elementy większe od mediany w danej grupie). Tak więc elementów nie mniejszych od p jest co najmniej $3(\frac{1}{2}\lceil n/5 \rceil - 1)$. W tym wliczony jest p. Ponieważ zakładamy, że wszystkie elementy są różne, więc elementów większych od p jest co najmniej $\frac{3}{10}n - 4$.
W podobny sposób pokazujemy, że co najmniej tyle samo jest elementów mniejszych od p. □

Koszt Procedury *Selection*
Niech T będzie funkcją ograniczającą z góry ten koszt. Bez zmniejszenia ogólności możemy założyć, że T jest monotoniczną funkcją niemalejącą. Ponieważ koszt wszystkich operacji poza rekurencyjnymi wywołaniami *Selection* można ograniczyć funkcją liniową otrzymujemy następującą nierówność rekurencyjną:

$$T(n) \leq T(\lceil n/5 \rceil) + T(7n/10 + 4) + O(n) \qquad \text{dla odpowiednio dużych } n.$$

Można łatwo sprawdzić, że $T(n)$ jest $O(n)$.

Uwagi:

- Stałą ukrytą w Twierdzeniu 2 pod "dużym O" można oszacować z góry przez 5.43.
- Obecnie najszybszy asymptotycznie algorytm wykonuje $2.9442n + o(n)$ porównań ([3]).
- Dolna granica - każdy algorytm deterministyczny wykonuje co najmniej $2n$ porównań.

14.2 Algorytmy zrandomizowane

14.2.1 Algorytm Hoare'a

W procedurze *Selection* element p wybieramy w sposób losowy. W ten sposób otrzymujemy algorytm o małej średniej liczbie porównań (choć oczywiście nadal liniowej) ([5],[2]).

Zauważ podobieństwo tego algorytmu do zrandomizowanego *Quicksortu*. Podobnie jak w tamtym algorytmie element rozdzielający można wybierać inaczej, np. jako medianę spośród losowej próbki kilku elementów zbioru T ([4]).

14.2.2 Algorytm *LazySelect*

IDEA Ze zbioru S wybieramy losowo próbkę R. Próbka powinna być niezbyt liczna, by można było szybko ją posortować. Znajdujemy w R dwa elementy L i H, takie że z dużym prawdopodobieństw em podzbiór $P \subseteq S$, elementów większych od L i mniejszych od H, jest nieduży oraz szukany element należy do P (możemy go wówczas łatwo znale"xć po posortowaniu P).

Algorytm *LazySelect*

1. Wybierz losowo, niezależnie, z powtórzeniami próbkę R złożoną z $n^{\frac{3}{4}}$ elementów zbioru S.
2. Posortuj R w czasie $O(n^{\frac{3}{4}} \log n)$.
3. $x \leftarrow kn^{-\frac{1}{4}}$; $L \leftarrow R[l]$; $H \leftarrow R[h]$
   gdzie $l = \max\{\lfloor x - \sqrt{n} \rfloor, 1\}$ oraz $h = \min\{\lfloor x + \sqrt{n} \rfloor, n^{\frac{3}{4}}\}$
4. Porównując L i H ze wszystkimi elementami z S oblicz:
   - $r_S(L) = \#\{y \in S \mid y < L\}$
   - $P \leftarrow \{y \in S \mid L \leq k \leq H\}$
5. Sprawd"x czy:
   - k-ty element zbioru S znajduje się w P, tj. czy $r_S(L) < k \leq r_S(L) + |P|$,
   - $|P| \leq 4n^{\frac{3}{4}} + 2$.

   Jeśli nie, to powtórz kroki 1-4.
6. Posortuj P.
   return $(P_{(k-r_S(L)+1)})$.

Twierdzenie 3 *Z prawdopodobieństw em* $= 1 - O\left(\frac{1}{\sqrt[4]{n}}\right)$ *LazySelect potrzebuje tylko jednej iteracji kroków* `1-4` *do znalezienia k-tego elementu zbioru S. Tak więc z prawdopodobieństw em* $1 - O\left(\frac{1}{\sqrt[4]{n}}\right)$ *LazySelect zatrzymuje się po wykonaniu 2n+o(n) porównań.*

Dowód tego twierdzenia nie jest trudny, ale wykracza poza zakres wykładu. Można go znale"xć w książce [8] (str. 47-50). Zainteresowani będą mogli go poznać na wykładzie z algorytmów zrandomizowanych (dla IV roku sekcji KiAA).
Teraz ograniczymy się do podania szkicu:

1. pokazujemy, że $kn^{-\frac{1}{4}}$ jest wartością oczekiwaną liczby elementów w R nie większych od szukanego,
2. pokazujemy, że wariancja tej liczby jest mniejsza od $\sqrt{n}$,
3. korzystamy z poniższej Nierówności Czebyszewa.

Twierdzenie 4 (nierówność Czebyszewa) *Jeśli X jest zmienną losową (o wartościach rzeczywistych) o wartości oczekiwanej* μ_X *i odchyleniu standardowym* σ_X. *Wówczas*

$$\forall_{t \in \mathcal{R}_+} \quad \mathbf{P}\Big[|X - \mu_X| \geq t\sigma_X\Big] \leq \frac{1}{t^2}$$

Literatura

[1] M.Blum, R.W.Floyd, V.Pratt, R.L.Rivest, R.E.Tarjan, Time bounds for selection, *Journal of Computer and System Sciences*, 7(1973), 448–461.

[2] T.Cormen, C.Leiserson, R.L.Rivest, *Introduction to Algorithms*, The MIT Press, 1990.

[3] D.Dor, U.Zwick, Median selection requires $O(2+\epsilon)n$ comparisions, Proceedings of the 37th FOCS, 1996, 125-134.

[4] R.W.Floyd, R.L.Rivest, Expected time bounds for selection, *Communication of the ACM*, 18(1975), 165–172.

[5] C.A.R.Hoare, Algorithm 63 (partition) and 65 (find), *Communication of the ACM*, 4(1961), 321–322.

[6] R.M.Karp, Probabilistic recurrence relations, w: *Proceedings of the 23rd STOC*, 1991, 190–197.

[7] D.E.Knuth, *The Art of Computer Programming,* vol. 3, Addison-Weslay, 1973.

[8] R.Motwani, P.Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

Notatki z AiSD. Nr ?? 16 marca 2005

Drzewa zbalansowane:
Drzewa Czerwono-Czarne

IIUWr. II rok informatyki. Przygotował: Krzysztof Loryś

15 Drzewa zbalansowane -wstęp

Najpoważniejszą wadą binarnych drzew przeszukiwań jest brak zabezpieczenia przed nierównomiernym rozrastaniem się, przez co pesymistyczny czas wykonywania operacji na nich może być liniowy względem liczby wierzchołków. Proste sposoby zaradzenia temu zjawisku mogą być niepraktyczne. Idealnym rozwiązaniem byłoby utrzymywanie drzew w stanie dokładnego zbalansowania, tj. tak, by wszystkie liście leżały na co najwyżej dwóch poziomach. Niestety przywracanie struktury takiego drzewa po zaburzeniu jej operacjami insert czy delete jest niezwykle kosztowne.

Zadanie: Skonstruuj dokładnie zbalansowane drzewo T o n wierzchołkach i znajd"x element x taki, że po dopisaniu x do T i zbalansowaniu powstałego drzewa, $\Omega(n)$ elementów T zmieni swoje pozycje.

Innym, niestety także zbyt kosztownym, rozwiązaniem byłoby pamiętanie sumy długości wszystkich ścieżek od korzenia do wierzchołków drzewa i przeorganizowywanie drzewa dopiero gdy suma ta przekroczy jakąś graniczną wartość (np. $2n \log n$).
Znanych jest wiele różnych odmian drzew zbalansowanych. My poznamy drzewa czerwono-czarne, drzewa AVL oraz B-drzewa. Wspomnimy także o drzewach samoorganizujących się oraz tzw. treap'ach.

16 Drzewa czerwono-czarne

16.1 Definicja

Definicja 1 *Binarne drzewo przeszukiwań jest* drzewem czerwono-czarnym *jeśli spełnia następujące warunki:*

w1. Każdy wierzchołek jest czerwony lub czarny.

w2. Każdy liść jest czarny.

w3. Jeśli wierzchołek jest czerwony, to jego obaj synowie są czarni.

w4. Na każdej ścieżce prowadzącej z danego wierzchołka do liścia jest jednakowa liczba czarnych wierzchołków.

Konwencja: Dla wygody przyjmujemy, że liśćmi są wierzchołki zewnętrzne odpowiadające NIL. Nie zawierają one żadnych informacji poza tym, że są liśćmi (co implikuje, że są czarne).

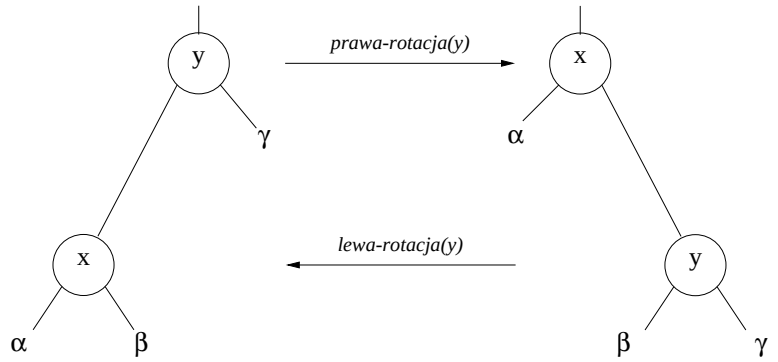
Definicja 2 *Liczbę czarnych wierzchołków na ścieżce z wierzchołka x (ale bez tego wierzchołka) do liścia nazywamy* czarną wysokością wierzchołka x *i oznaczamy* $bh(x)$.

Fakt 18 *Czerwono-czarne drzewo o n wierzchołkach wewnętrznych ma wysokość nie większą niż* $2\log(n+1)$.

Dowód. Przez pokazanie, że drzewo zakorzenione w dowolnym wierzchołku x zawiera co najmniej $2^{bh(x)} - 1$ wierzchołków wewnętrznych.

16.2 Operacje słownikowe na drzewach czerwono-czarnych

Operacje wstawiania i usuwania elementu mogą powodować zaburzenie własności *w1-w4*. W procedurach przywracających te własności podstawową rolę odgrywają rotacje.



Ważne własności:

1. Rotacje nie zmieniają porządku infiksowego elementów zapamiętanych w drzewie.
2. Pojedynczą rotację można wykonać w czasie stałym.

16.2.1 Wstawianie elementu

Wstawianie elementu wykonujemy jak w zwykłym binarnym drzewie przeszukiwań. Następnie nowemu wierzchołkowi nadajemy kolor czerwony i przywracamy drzewu własności drzewa czerwono-czarnego.
Łatwo widać, że jedyną własnością jaka mogła zostać zaburzona jest *w3*.

Przywracanie własności *w3*.

Idea:
Początkowo *w3* może być zaburzona jedynie w miejscu gdzie nowy wierzchołek x został wstawiony (wtedy gdy ojciec x-a jest czerwony).
Wędrujemy od wierzchołka x w górę. Stosujemy operację zmiany koloru wierzchołków, by przenieść zaburzenie na przodków x-a i operację rotacji do zlikwidowania zaburzenia (uwaga: operacja rotacji będzie wykonana co najwyżej dwa razy). Będziemy przy tym

dbać, by nie zaburzyć pozostałych własności drzewa czerwono-czarnego.

Procedura przywracająca własność *w3* musi rozważyć następujące przypadki (zakładamy, że ojciec x-a jest lewym synem swojego ojca; gdy jest prawym synem otrzymujemy symetryczne przypadki):
Przypadek 1. *Wujek x-a jest czerwony.*

- ◇ Zmieniamy kolory:
  - dziadka x-a malujemy na czerwono (dotąd był czarny, ponieważ ojciec x-a był czerwony i własność *w3* była zachowana),
  - ojca i wujka x-a malujemy na czarno.
- ◇ $x \leftarrow$ dziadek x-a.
- ◇ wywołujemy procedurę rekurencyjnie dla nowego x-a.

Przypadek 2. *Wujek x-a jest czarny i x jest prawym synem swojego ojca.*

- ◇ $x \leftarrow$ ojciec(x);
- ◇ *lewa-rotacja*(x)

W ten sposób otrzymujemy przypadek 3.

Przypadek 3. *Wujek x-a jest czarny i x jest lewym synem swojego ojca.*

- ◇ Zmieniamy kolory:
  - dziadka x-a malujemy na czerwono.
  - ojca x-a malujemy na czarno.
- ◇ *prawa-rotacja*(dziadek(x));

16.2.2 Usuwanie elementu

Wykonujemy jak w zwykłym binarnym drzewie przeszukiwań a następnie przywracamy własności *w1-w4*.

Przypomnienie: {Usuwanie wierzchołka w drzewie binarnym.}

Niech y będzie usuwanym wierzchołkiem.

- jeśli y jest liściem, to usuwamy y;
- jeśli y ma jednego syna x, to usuwamy y a x podczepiamy pod ojca y-a;
- jeśli y ma dwóch synów, to y zastępujemy przez x - następnik y-a (tj. najmniejszy element w prawym poddrzewie y-a), usuwamy x a syna x-a (x może mieć co najwyżej prawego syna), jeśli istnieje, poczepiamy pod ojca x-a.

□

Jeśli usunięty wierzchołek y miał kolor czarny, to zaburzona zostaje własność *w4* drzewa. Może też zostać zaburzona własność *w3*.

IDEA NAPRAWY: Czarny kolor z y-a (nazwijmy go extra czarnym kolorem) przesuwamy na jego syna x (syn ten był jedynakiem i został podczepiony pod ojca y-a lub jest liściem). W ten sposób wlasności *w3* i *w4* zostają przywrócone. Jedyny kłopot spowodowany jest tym, iź x mógł być czarny i teraz zawiera podwójny czarny kolor, a więc *w1* może być zaburzona.

Procedura przywracająca *w1* przesuwa ten extra kolor w odpowiedni sposób w górę drzewa, aź:

- napotka czerwony wierzchołek, którego może pomalować extra kolorem, lub
- przesunie go do korzenia i wówczas może go usunąć, lub
- dojdzie do miejsca, gdzie może wykonać odpowiednie rotacje i zmiany kolorów.

Zakładamy, źe wierzchołek x jest lewym synem swojego ojca (gdy x jest prawym synem, rozważania są symetryczne). Musimy rozważyć następujące przypadki:

Przypadek 1. *Brat x-a jest czerwony (to implikuje, źe ojciec x-a jest czerwony).*

◇ Zmieniamy kolory:

- brata x-a malujemy na czarno,
- ojca x-a malujemy na czerwono.

◇ *lewa-rotacja*(ojciec(x)).

Teraz x ma czarnego brata (jest nim były bratanek, który musiał być czarny, ponieważ miał czerwonego ojca) i otrzymujemy jeden z pozostalych przypadków.

Przypadek 2. *Brat x-a jest czarny i obydwaj jego bratankowie są czarni.*

◇ malujemy brata na czerwono,

◇ przesuwamy extra czarny kolor na ojca x-a (tj. $x \leftarrow$ojciec(x)).

Przypadek 3. *Brat x-a jest czarny, lewy bratanek - czerwony a prawy bratanek - czarny.*

◇ Zmieniamy kolory:

- brata x-a malujemy na czerwono,
- lewego bratanka malujemy na czarno.

◇ *prawa-rotacja*(brat(x)).

Teraz bratem x-a jest jego były lewy bratanek (który teraz ma czarny kolor), a prawym bratankiem jego były brat (który teraz ma kolor czerwony). Otrzymujemy przypadek 4.

Przypadek 4. *Brat x-a jest czarny a prawy bratanek czerwony.*

- ◇ Zmieniamy kolory:
  - brata x-a malujemy na kolor, którym pomalowany był ojciec,
  - ojca x-a malujemy na czarno,
  - prawego bratanka malujemy na czarno (extra kolorem z x-a).
- ◇ *lewa-rotacja*(ojciec(x)).

16.3 Koszt operacji

Koszt operacji wstawiania i usuwania elementu wynosi $O(\log n)$.

5 zmniejszyc rysunki

Notatki z AiSD. Nr 12. 16 marca 2005

Drzewa zbalansowane:
Drzewa AVL

IIUWr. II rok informatyki. Przygotował: Krzysztof Loryś

17 Definicja

Definicja 3 *Binarne drzewo przeszukiwań jest* drzewem AVL*, jeśli dla każdego wierzchołka wysokości jego lewego i prawego poddrzewa różnią się o co najwyżej* 1*.*

Uwaga: Skrót AVL pochodzi od pierwszych liter nazwisk autorów (Adelson-Velskij i Landis).

18 Zasadnicza cecha

Twierdzenie 5 *Wysokość drzewa AVL o n wierzchołkach jest mniejsza niż* $1.4405\log(n+2)$*.*

Fakt 19 *Liczba wierzchołków w dowolnym drzewie binarnym jest o* 1 *mniejsza od liczby pustych wska"zników (tj. równych* `NIL`*).*

Dowód (Twierdzenia 5)

Niech $\rho(i) =$ "liczba pustych wska"zników w minimalnym (tj. o najmniejszej 'liczbie wierzchołków) drzewie AVL o wysokości i".

Indukcyjnie po wysokości h drzewa dowodzimy, że $\rho(h) = (h+2)$-a liczba Fibonacciego.

Łatwo sprawdzić, że $\rho(1) = 2$ i $\rho(2) = 3$.

Niech T będzie minimalnym drzewem AVL o wysokości h ($h \geq 3$). Z minimalności T wiemy, że jedno z poddrzew podwieszonych pod jego korzeniem musi być minimalnym drzewem AVL o wysokości $h-1$, a drugie - minimalnym drzewem AVL o wysokości $h-2$. Ponieważ każdy pusty wska"znik T jest pustym wska"znikiem w jednym z tych poddrzew, otrzymujemy wzór $\rho(h) = \rho(h-1) + \rho(h-2)$.

Teraz niech N będzie liczbą wierzchołków w T. Z Faktu 19 i powyższych rozważań mamy

$$N + 1 > \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+2} - 1,$$

co po prostych przekształceniach daje tezę. □

19 Operacje słownikowe na drzewach AVL

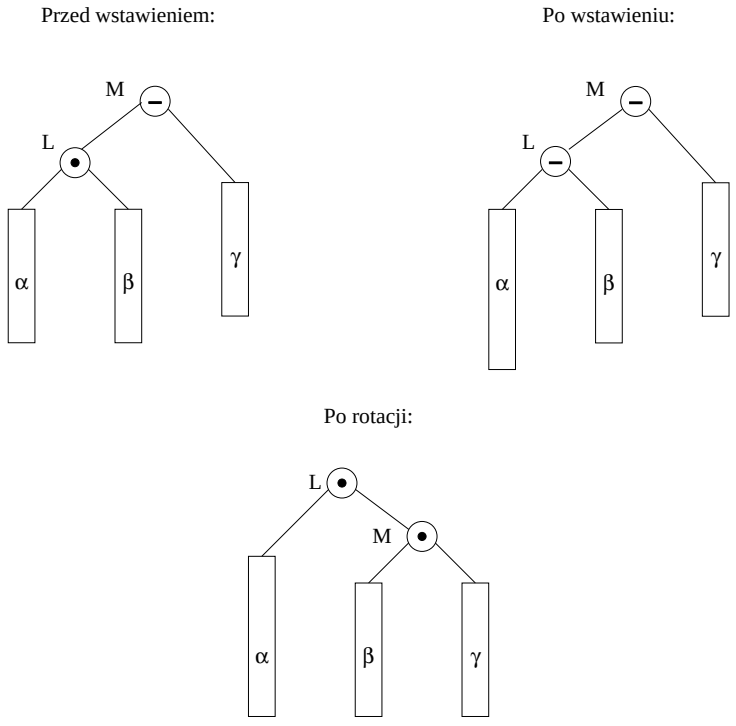
Wyszukiwanie elementu wykonuje się identycznie jak dla zwykłych binarnych drzew przeszukiwań. Pozostałe dwie operacje mogą zaburzyć strukturę drzewa AVL. Przywracanie tej struktury nazywamy *balansowaniem drzewa.*

19.1 Wstawianie elementu

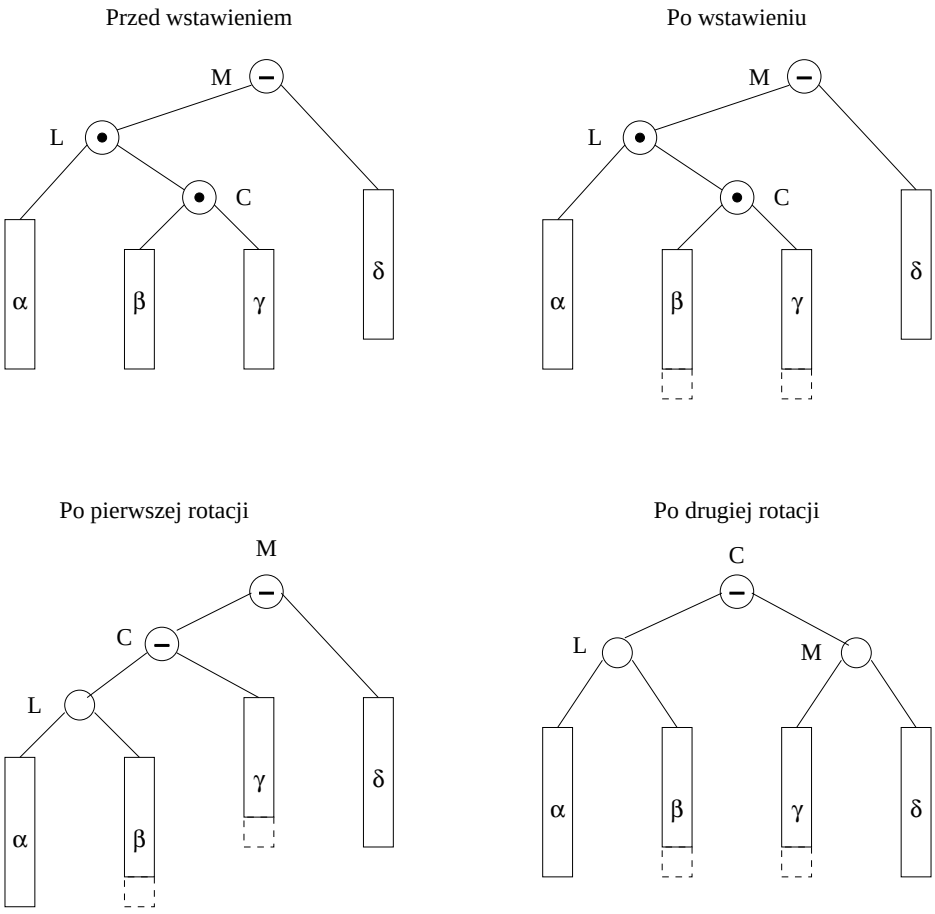
Niech M będzie pierwszym węzłem na drodze od wstawionego elementu do korzenia, w którym nastąpiło naruszenie równowagi drzewa AVL. Oznacza to, że przed operacją wstawienia poddrzewa zakorzenione w M były nierównej wysokości i wstawienie zwiększyło wysokość wyższego poddrzewa. Załóżmy, że tym poddrzewem jest lewe podrzewo i oznaczmy jego korzeń przez L (sytuacja, w której wyższym poddrzewem jest prawe poddrzewo jest symetryczna).

Procedura balansowania musi oddzielnie rozpatrywać dwa przypadki:

(A) W drzewie o korzeniu L zwiększyła się wysokość lewego poddrzewa.



(B) W drzewie o korzeniu L zwiększyła się wysokość prawego poddrzewa.



Uwaga: Po zbalansowaniu wysokość drzewa zakorzenionego w C jest równa wysokości drzewa zakorzenionego w M przed operacją wstawienia. Dlatego nie ma potrzeby przywracania zrównoważenia w innych węzłach poza M.

19.2 Usuwanie elementu

Operacja ta jest znacznie bardziej skomplikowana.
IDEA:

Algorytm *DeleteAVLnode*

1. Znale"xć wierzchołek zawierający element g, który chcemy usunąć.
2. Jeśli jest to wierzchołek wewnętrzny to wstawić do niego element g' z drzewa bezpośrednio następny (bąd"x bezpośrednio poprzedni) po g.
3. Powtarzać rekurencyjnie krok 2 dla g', tak długo, aź g' będzie elementem z liścia.
4. Usunąć ten liść. Przejść drogę od tego liścia do korzenia przywracając zrównoważenie wierzchołków na tej drodze przy pomocy rotacji.

Uwagi:

1. Tym razem może się zdarzyć, źe trzeba będzie dokonywać rotacji dla wszystkich wierzchołków na tej drodze.
2. Szczegółowy opis drzew AVL można znale"xć w ksiaźce [1].

19.3 Koszt

Wszystkie operacje słownikowe na drzewach AVL można wykonać w czasie ograniczonym funkcją liniową od wysokości drzewa, a więc w czasie $O(\log n)$.

20 Zastosowanie drzew AVL do implementacji list

Typowymi operacjami na listach są m.in.:

1. wstawianie elementu na wskazaną pozycję,
2. usuwanie elementu ze wskazanej pozycji,
3. konkatenacja list,
4. podział listy na dwie podlisty wg zadanej pozycji.

Przy tradycyjnych implementacjach list (tj. w tablicach lub przy pomocy zmiennych wska-"xnikowych) niektóre z tych operacji wymagają czasu liniowego. Drzewa AVL pozwalają na implementację list, która umoźliwia wykonanie powyższych operacji w czasie $O(\log n)$. Wystarczy w każdym wierzchołku pamiętać liczbę elementów w jego lewym poddrzewie (liczba ta wyznacza pozycję elementu w liście przechowywanej w drzewie zakorzenionym w tym wierzchołku).

Szczegóły pozostawiamy jako temat do samodzielnych studiów.

Literatura

[1] N.Wirth, *Algorytmy + Struktury Danych = Programy.*

Notatki z AiSD. Nr 13. 16 marca 2005

Drzewa zbalansowane:
B-drzewa

IIUWr. II rok informatyki. Przygotował: Krzysztof Loryś

21 Wstęp

Przypomnienie: *Słownikiem* nazywamy strukturę danych umożliwiającą pamiętanie zbioru pewnych elementów oraz wykonywanie na nim operacji wstawiania, wyszukiwania i usuwania elementu.

Gdy chcemy używać niewielkich słowników możemy przechowywać je w pamięci wewnętrznej. Strukturami danych nadającymi się do tego celu są przykładowo zbalansowane drzewa binarne (np. drzewa AVL, drzewa czerwono-czarne) i tablice hashujące.

Dla implementacji dużych słowników, nie mieszczących się w pamięci wewnętrznej idealnie nadają się B-drzewa. Są to zbalansowane drzewa przeszukiwań specjalnie zaprojektowane tak, by operacje na nich były efektywnie wykonywane wtedy gdy są one przechowywane w plikach dyskowych.
Cechy charakterystyczne B-drzew:

- Wszystkie liście B-drzewa leżą na tej samej głębokości.
- Każdy węzeł zawiera wiele elementów zbioru (są one uporządkowane).
- Nowe elementy zapamiętywane są w liściach.
- Drzewo rośnie od liści do korzenia: jeśli jakiś wezeł jest pełny to tworzony jest jego nowy brat, który przejmuje od niego połowę elementów a jeden z jego elementów (środkowy) wędruje wraz ze wskaźnikiem na nowego brata do ojca. Jeśli w ten sposób podzielony zostanie korzeń, to tworzony jest nowy korzeń, a stary będzie jednym z dwóch jego synów. Jest to jedyny moment, w którym może wzrosnąć wysokość B-drzewa.

22 Formalny opis

Definicja. *B-drzewo o minimalnym stopniu t* posiada następujące własności:

1. Każdy węzeł x ma następujące pola:
   a. $n[x]$ - liczba kluczy aktualnie pamiętanych w x,
   b. $2t-1$ pól $key_i[x]$ na klucze (pamiętane są one w porządku niemalejącym: $key_1[x] \leq key_2[x] \cdots key_{n[x]}[x]$),
   c. $leaf[x]$ - pole logiczne = TRUE iff x jest liściem.

2. Jeśli x jest węzłem wewnętrznym to posiada ponadto $2t$ pól $c_i[x]$ - na wskaźniki do swoich dzieci.

3. Klucze pamiętane w poddrzewie o korzeniu $c_i[x]$ są nie mniejsze od kluczy pamiętanych w poddrzewie o korzeniu $c_j[x]$ (dla każdego $j < i$) i nie większe od kluczy pamiętanych w poddrzewie o korzeniu $c_k[x]$ (dla każdego $i < k$).

4. Wszystkie liście mają tę samą głębokość (oznaczamy ją h).

5. $t \geq 2$ jest ustaloną liczbą całkowitą określającą dolną i górną granicę na liczbę kluczy pamiętanych w węzłach:

   a. Każdy węzeł różny od korzenia musi pamiętać co najmniej $t-1$ kluczy (a więc musi mieć co najmniej t dzieci). Jeśli drzewo jest niepuste, to korzeń musi pamiętać co najmniej jeden klucz.

   b. Każdy węzeł może pamiętać co najwyżej $2t-1$ kluczy (a więc może mieć co najwyżej $2t$ dzieci). Mówimy, że węzeł jest *pełny* jeśli zawiera dokładnie $2t-1$ kluczy.

23 Operacje na B-drzewach

Zakładamy, że B-drzewo pamiętane jest na dysku. Jego węzły sprowadzane są do pamięci wewnętrznej operacją *disc-read*. Każdorazowo w pamięci wewnętrznej znajduje się tylko niewielka liczba węzłów. Tylko te węzły mogą być modyfikowane przez program. Po każdorazowej modyfikacji węzeł zapisywany jest operacją *disc-write* na dysk. Przyjmujemy, że operacja *disc-read* nie powoduje żadnej akcji gdy wydana jest do węzła znajdującego się aktualnie w pamięci.

23.1 Przeszukiwanie

Wykonuje się w podobny sposób jak w binarnych drzewach przeszukiwań. Jedyna różnica polega na tym, źe przechodząc wierzchołki drzewa dokonujemy wyboru między wieloma synami.
W poniższej procedurze k jest poszukiwanym kluczem a x jest adresem węzła, od którego rozpoczynamy szukanie.

```
procedure B-Tree-Search(x, k)
    i ← 1
    while i ≤ n[x] and k > key_i[x]  do i ← i + 1
    if i ≤ n[x] and k = key_i[x]  then  return (x, i)
    if leaf[x]  then  return NIL
                 else  disc-read(c_i[x])
                       return B-Tree-Search(c_i[x], k)
```

W przypadku gdy $n[x]$ jest duże zamiast liniowego przeszukiwania kluczy w wierzchołku, może opłacić się zastosowanie przeszukiwania binarnego.

23.2 Tworzenie pustego B-drzewa

```
procedure B-Tree-Create(T)
    x ← Allocate-Node()
    leaf[x] ← TRUE
    n[x] ← 0
    Disc − Write(x)
    root[T] ← x
```

23.3 Rozdzielanie węzła w B-drzewie

Znaczenie parametrów:
y - pełny wierzchołek, tj. zawierający $2t-1$ kluczy, który należy rozdzielić;
x - ojciec y-ka, procedura *B-Tree-Split-Child* będzie wywoływana dla x-a, który jest niepełny;
i - określa, którym synem x-a jest y.

```
procedure B-Tree-Split-Child(x, i, y)
    z ← Allocate-Node()
    leaf[z] ← leaf[y]
    n[z] ← t − 1
    for j ← 1 to t − 1 do key_j[z] ← key_{j+t}[y]
    if not leaf[y] then for j ← 1 to t do c_j[z] ← c_{j+t}[y]
    n[y] ← t − 1
    for j ← n[x] + 1 downto i + 1 do c_{j+1}[x] ← c_j[x]
    c_{i+1}[x] ← z
    for j ← n[x] downto i do key_{j+1}[x] ← key_j[x]
    key_i[x] ← key_t[y]
    n[x] ← n[x] + 1
    Disc-Write(y); Disc-Write(z); Disc-Write(z)
```

23.4 Umieszczanie klucza w B-drzewie

Umieszczenie klucza k w drzewie dokonuje się w procedurze *B-Tree-Insert-Nonfull*. Procedura *B-Tree-Insert* sprawdza jedynie czy T nie ma pełnego korzenia i jeśli tak jest, to tworzy nowy korzeń, a stary rozdziela na dwa węzły, które stają się synami nowego korzenia.

```
procedure B-Tree-Insert(T, k)
    r ← root[T]
    if n[r] = 2t − 1
          then s ← Allocate-Node()
                root[T] ← s
                leaf[s] ← False
                n[s] ← 0
                c_1[s] ← r
                B-Tree-Split-Child(s, 1, r)
                B-Tree-Insert-Nonfull(s, k)
           else B-Tree-Insert-Nonfull(r, k)
```

Procedura *B-Tree-Insert-Nonfull* przechodzi ścieżkę od korzenia do odpowiedniego liścia, rozdzielając wszystkie pełne wierzchołki, które ma przejść. Chodzi o to, by w momencie wywołania tej procedury węzeł x był niepełny.

```
procedure B-Tree-Insert-Nonfull(x, k)
    i ← n[x]
    if leaf[x] then
                  while i ≥ 1 and k < key_i[x]
                      do key_{i+1}[x] ← key_i[x]
                          i ← i − 1
                  key_{i+1}[x] ← k
                  n[x] ← n[x] + 1
                  Disc-Write(x)
       else  while i ≥ 1 and k < key_i[x] do i ← i − 1
             i ← i + 1
             Disc-Read(c_i[x])
             if n[c_i[x]] = 2t − 1
                  then B-Tree-Split-Child(x, i, c_i[x])
                       if k > key_i[x] then i ← i + 1
             B-Tree-Insert-Nonfull(c_i[x], k)
```

23.5 Usuwanie klucza z B-drzewa

```
procedure B-Tree-Delete(x, k)
   (* zadanie domowe *)
```

24 Koszt operacji

Twierdzenie 1 *Jeśli $n \geq 1$, to dla każdego B-drzewa o wysokości h i stopniu minimalnym $t \geq 2$ pamiętającego n kluczy: $h \leq \log_t \frac{n+1}{2}$.*

Przykład Jeśli przyjmiemy $t = 100$, to wówczas B-drzewo zawierające do 2000000 elementów ma wysokość nie większą niż 3. Tak więc wszystkie omawiane operacje na takim

B-drzewie będą wymagały dostępu do co najwyżej trzech węzłów (a więc trzeba będzie wykonać co najwyżej sześć operacji dyskowych).

Niech n będzie liczbą węzłów w B-drzewie a $h = \Theta(\log_t n)$ - wysokością drzewa.

procedura	liczba operacji dyskowych	koszt pozostałych operacji
B-Tree-Search	$O(h)$	$O(th)$
B-Tree-Create	$O(1)$	$O(1)$
B-Tree-Split-Child	$O(1)$	$O(t)$
B-Tree-Insert	$O(h)$	$O(th)$
B-Tree-Delete	$O(h)$	$O(th)$

25 Rada praktyczna

Należy rozważnie dobierać wartość t. Trzeba pamiętać, że wraz ze wzrostem t rośnie liczba operacji wykonywanych w pamięci wewnętrznej i może zniweczyć korzyści wynikające ze zmniejszenia liczby operacji dyskowych.

Notatki z AiSD. Nr ?? 16 marca 2005

HASHOWANIE

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

26 Wstęp

Hashowanie jest jedną z metod realizacji słowników. Poznaliśmy już m.in. drzewa czerwono-czarne, drzewa AVL, czy B-drzewa, struktury, które umożliwiały wykonywanie operacji słownikowych w czasie proporcjonalnym z logarytmu z wielkości słownika.

Gdy uniwersum jest małe (powiedzmy n elementowe), możemy wykorzystać n elementowe tablice bitowe (i-ty element takiej tablicy jest równy 1 wtedy i tylko wtedy gdy i-ty element uniwersum należy do zbioru; zakładamy przy tym, że umiemy efektywnie numerować elementy uniwersum). Przy takim sposobie pamiętania słownika czas wykonania operacji słownikowych jest stały.

27 Metoda funkcji hashujących

Do pamiętania elementów podzbioru wykorzystywana jest tablica $T[0..m-1]$. Zwykle m jest proporcjonalne do maksymalnej liczności słownika; wielkość uniwersum nie ma tu większego znaczenia. Metoda wykorzystuje funkcję (tzw. *funkcję haszującą*) $h : U \to \{0, ..., m-1\}$, określającą miejsce pamiętania elementów U w T.

27.1 Funkcje haszujące

Dobra funkcja haszująca powinna spełniać następujący warunek:

$$\text{(dfh)} \qquad \forall_{j=0,\ldots,m-1} \sum_{k:h(k)=j} P(k) = \frac{1}{m},$$

gdzie $P(k) =$ jest prawdopodobieństwem tego, że $k \in U$ będzie parametrem którejś z operacji słownikowych. W praktyce warunek ten jest zwykle niesprawdzalny, gdyż nie znamy P. Ponadto, jeśli metoda ma być efektywna, funkcja hashująca powinna być szybkoobliczalna.

Nie polecam wymyślania własnych funkcji hashujących (przynajmniej na początku). Lepiej skorzystać z doświadczenia innych.

Przykłady funkcji haszujących.

- $h(k) = k \bmod m$

  UWAGA. Należy wykazać się ostrożnością w wyborze m. Nie zaleca się m postaci 2^p, gdyż wówczas $h(k)$ jest równe ostatnim p bitom klucza k, a te często mają nierównomierny rozkład. Z tego samego powodu nie zaleca się brać jako m potęg liczby 10. Zwykle dobrymi wartościami m są liczby pierwsze niezbyt bliskie potęgom liczby 2.

- $h(k) = \lfloor m(kA - \lfloor kA \rfloor \rfloor$, gdzie A jest ustaloną liczbą z przedziału $(0, 1)$.

  UWAGA. Teraz wartość m nie ma takiego znaczenia jak poprzednio i zwykle bierze się m równe potędze liczby 2 (ze względu na łatwość mnożenia). Wybór A jest bardziej zależny od cech danych, jednak zwykle A równe $(\sqrt{5} - 1)/2 \approx 0.6180339887$ jest dobre.

27.2 Metody pamiętania elementów

Ponieważ wielkość tablicy T jest z reguły znacznie mniejsza od wielkości uniwersum, dość często zetkniemy się z sytuacją, gdy chcemy w T zapamiętać y, taki że $h(y) = h(x)$ dla pewnego x aktualnie pamiętanego w T. Sytuację taką nazywamy *kolizją*. Sposoby rozwiązywania kolizji zależą istotnie od tego w jaki sposób pamiętamy elementy w tablicy.

Rozważymy dwa sposoby pamiętania elementów.

27.2.1 Listy elementów

i-ty element tablicy zawiera wskaźnik na początek listy tych elementów x słownika, dla których $h(x) = i$.

Jeśli założymy, że koszt obliczenia wartości funkcji haszującej jest stały, to koszt INSERT i DELETE też jest stały, a koszt SEARCH(K) jest zależny od długości listy $T[k]$.

Fakt 20 *Przy założeniu (dfh) średni koszt operacji* SEARCH *wynosi* $\Theta(1 + \frac{n}{m})$*, gdzie* n*-liczba elementów* U *pamiętanych w* T*.*

UWAGA. $\alpha = \frac{n}{m}$ nazywane jest *współczynnikiem wypełnienia* tablicy haszującej.

Wniosek 1 *Gdy* $n = O(m)$*, to średni koszt operacji* SEARCH *(a więc także wszystkich operacji słownikowych) jest* $\Theta(1)$*.*

27.2.2 Adresowanie otwarte

Teraz elementy słownika pamiętamy bezpośrednio w elementach tablicy T. Likwidujemy w ten spoób istotny mankament poprzedniej metody: nie tracimy miejsca na pamiętanie wska"xników. Powstaje jednak nowe niekorzystne zjawisko - przepełnienie się tablicy T. Często nie potrafimy z góry określić wielkości słownika i dlatego rozpoczynamy z tablicą umiarkowanych rozmiarów. Gdy okazuje się ona za mała (jest tak nie tylko wtedy gdy w słowniku chcemy umieścić $(m+1)$-szy element, lecz już wtedy gdy duży stopień wypełnienia tablicy powoduje, że operacje słownikowe są kosztowne), powiększamy ją, zmieniamy funkcję hashującą (tak by przyjmowała wartości z nowego zakresu) i na nowo obliczamy miejsca umieszczenia wszystkich elementów słownika.

Usuwanie kolizji

Używamy funkcji haszującej

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

Najpierw próbujemy umieścić element k na pozycji $h(k, 0)$. Jeśli pozycja ta jest zajęta, próbujemy $h(k, 1)$ jeśli ta jest zajęta sprawdzamy pozycję $h(k, 2)$, itd...

Funkcja h powinna spełniać następujący warunek:

(per) $\forall_{k \in U} \langle h(k, 0), \ldots, h(k, m-1) \rangle$ jest permutacją zbioru $\{0, 1, \ldots, m-1\}$.

To gwarantuje nam, że nie znajdziemy miejsca na umieszczenie danego elementu dopiero wtedy, gdy tablica jest całkowicie zapełniona.

Przykłady:

- *Metoda liniowa*:
$$h(k, i) = (h'(k) + i) \bmod m,$$
gdzie $h' : U \to \{0, .., m-1\}$ jest pomocniczą funkcją haszującą (np. takie jak opisano powyżej).

- *Metoda kwadratowa*:
$$h(k, i) = (h'(k) + c_1 i + c_2 * i^2) \bmod m,$$
gdzie h' - jak poprzednio.
Uwaga: $c_1, c_2 \neq 0$. Stałe c_1, c_2 oraz m powinny być tak dobrane by zachodził warunek (per).

- *Podwójne haszowanie*:
$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m,$$
gdzie h_1, h_2 - pomocnicze funkcje haszujące.
Uwaga: Dla każdego $k \in U$, $h_2(k)$ powinno być względnie pierwsze z m.

W praktyce najlepsze rezultaty daje podwójne haszowanie. W metodzie liniowej (i w mniejszym stopniu w metodzie kwadratowej) występuje negatywne zjawisko tworzenia się *zlepków* (tj. zwartych obszarów tablicy T, zajętych przez elementy U), które znacznie obniża efektywność metody.

Podczas wykonywania operacji Delete w miejscu usuwanego elementu w tablicy T należy wpisać znacznik świadczący o tym, że to miejsce było już kiedyś zajęte.

Analiza kosztów

Dla uproszczenia analizy stosujemy poniższe (nieco wyidealizowane) założenie:

(dper) ciąg $\langle h(k, 0), \ldots, h(k, m-1) \rangle$ jest z równym prawdopodobieństwem dowolną permutacją zbioru $\{0, \ldots, m-1\}$.

Twierdzenie 9 *Przy założeniu (dper) i* $\alpha = \frac{n}{m} < 1$ *oczekiwana liczba prób w poszukiwaniu zakończonym fiaskiem jest* $\leq \frac{1}{1-\alpha}$.

PRZYKŁAD. Załóżmy, że utworzyliśmy słownik i teraz wykonujemy wiele operacji SEARCH. Jeśli tablica jest zajęta w 50%, to średnia liczba prób przy poszukiwaniu zakończonym niepowodzeniem jest nie większa od 2; gdy tablica zajęta jest w 90%, to liczba ta jest nie większa od 10.

Wniosek 2 *Przy powyższych założeniach umieszczenie elementu w tablicy haszującej wymaga średnio* $\leq \frac{1}{1-\alpha}$ *prób.*

Twierdzenie 10 *Przy założeniu (dper) i* $\alpha = \frac{n}{m} < 1$ *oczekiwana liczba prób w poszukiwaniu zakończonym sukcesem jest* $\leq \frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha}$.

PRZYKŁAD. Gdy tablica wypełniona jest w 90%, poszukiwanie zakończone sukcesem wymaga średnio nie więcej niż 3.67 prób.

UWAGA. W praktyce, pomimo niespełnienia założenia (dper), koszt operacji słownikowych jest zbliżony do kosztu wynikającego z powyższych twierdzeń.

28 Hashowanie uniwersalne

Oczywiście dla każdej funkcji hashującej istnieją dane, które powodują, że czas wykonywania operacji słownikowych jest duży (np. może się zdarzyć, że dla wszystkich elementów umieszczanych w słowniku wartość funkcji hashującej będzie ta sama). Aby uniezależnić się od takich danych wprowadzamy, podobnie jak w algorytmie *Quicksort*, randomizację: zamiast korzystać z ustalonej funkcji hashującej, losujemy ją na początku działania programu z pewnej rodziny funkcji.

Definicja 13 *Niech* H *będzie rodziną funkcji hashujących z* U *w* $\{0, \ldots, m-1\}$. *Rodzinę* H *nazywamy* uniwersalną, *jeśli* $\forall_{x,y \in U;\, x \neq y}$:

$$|\{h \in H : \ h(x) = h(y)\}| = \frac{|H|}{m}$$

Twierdzenie 11 *Niech* H *będzie uniwersalną rodziną funkcji hashujących. Dla dowolnego zbioru* $n \leq m$ *kluczy, liczba kolizji w jakich bierze udział ustalony (ale dowolny) klucz* x *jest w średnim przypadku mniejsza od* 1.

28.1 Przykład rodziny uniwersalnej

Niech m będzie liczbą pierwszą oraz $|U| < m^{r+1}$. Dla każdego $0 \leq a < m^{r+1}$ definiujemy funkcję h_a:

$$h_a(x) = \sum_{i=0}^{r} a_i x_i,$$

gdzie $\langle a_0, a_1, \ldots, a_r \rangle$ i $\langle x_0, x_1, \ldots, x_r \rangle$ są reprezentacjami odpowiednio liczb a i x w systemie m-arnym.

Twierdzenie 12 *Rodzina* $H = \{h_a : 0 \leq a < m^{r+1}\}$ *jest rodziną uniwersalną.*

UWAGA: Dowody wszystkich twierdzeń można znale"xć w książce Cormena.

Notatki z AiSD. Nr ?? 16 marca 2005

Kopce dwumianowe

IIUWr. II rok informatyki. Przygotował: Krzysztof Loryś

29 Definicja

Kopce dwumianowe są strukturą danych umożliwiającą łatwe wykonywanie zwykłych operacji kopcowych (insert, makeheap, findmin i deletemin) a ponadto operacji *meld* łączenia kopców.

Definicja 14 Drzewa dwumianowe *zdefiniowane są indukcyjnie: i-te drzewo dwumianowe B_i składa się z korzenia oraz i poddrzew: B_0, B_1, ..., B_{i-1}.*

Definicja 15 Kopiec dwumianowy *to zbiór drzew dwumianowych, które pamiętają elementy z uporządkowanego uniwersum zgodnie z porządkiem kopcowym.*

Definicja 16 $\forall_{x-wierzchoka}$ *rząd(x)=liczba dzieci x-a.*
$\forall_{T-drzewa}$ *rząd*(T)=*rząd*(*korzeń*(T)).

Szczegół implementacyjny: Aby umożliwić szybką realizację operacji na kopcu dwumianowym, będziemy zakładać, że dzieci każdego wierzchołka zorganizowane są w cykliczną listę dwukierunkową, a ojciec pamięta wska"xnik do jednego z nich (np. do dziecka o najmniejszym rzędzie).

30 Operacje na kopcach dwumianowych

30.1 Łączenie drzew dwumianowych - operacja *join*

Dwa drzewa B_i łączymy ze sobą tak, że korzeń jednego drzewa staje się synem korzenia drugiego drzewa. W ten sposób otrzymujemy drzewo B_{i+1}.

Uwagi:

(a) Nigdy nie będziemy łączyć drzew o różnych rzędach.

(b) Zawsze podłączamy to drzewo, którego korzeń pamięta mniejszą wartość do tego, którego korzeń pamięta większą wartość.

RYSUNEK - będzie pó"xniej

Koszt: $O(1)$.

30.2 Operacja *makeheap(i)*

Bez komentarza. Koszt - $O(1)$.

30.3 Operacja *findmin*

Z każdym kopcem dwumianowym wiążemy wska"xnik *MIN* wskazujący na minimalny element. Operacja findmin polega na odczytaniu tego elementu. Stąd jej koszt wynosi $O(1)$.

30.4 Operacja *insert(i,h)*

Wykonujemy $meld(h, makeheap(i))$.
Koszt tej operacji zależny jest od kosztu *meld*. Podamy go dalej.

30.5 Operacja *deletemin(h)*

Sposób jej wykonania zależy od realizacji *meld*. Omówimy go dalej.

30.6 Operacja *meld*

Rozważymy dwie metody realizacji operacji *meld*:

(a) wersja "*eager*" - w tej wersji kopiec przybiera docelowy kształt przed wykonaniem następnej po meld operacji;

(b) wersja "*lazy*" - w tej wersji pozwalamy, by kopiec utracił strukturę kopca dwumianowego; zostanie ona mu przywrócona dopiero podczas wykonania operacji deletemin.

30.6.1 Eager meld(h,h')

W tej wersji drzewa kopca dostępne są poprzez tablicę wska"xników (będziemy ją oznaczać tą samą nazwą co kopiec). Każdy kopiec zawiera co najwyżej jedno drzewo każdego rzędu. i-ty wska"xnik jest albo pusty albo wskazuje na drzewo i-tego rzędu.
$Meld(h, h')$ tworzy nowy kopiec H; stare kopce ulegają likwidacji.

Procedure Eagermeld(h,h')
  if $key(MIN_h) < key(MIN_{h'})$ **then** $MIN_H \leftarrow MIN_h$ **else** $MIN_H \leftarrow MIN_{h'}$
  $carry \leftarrow nil$;
  for $i \leftarrow 0$ **to** $maxheapsize$ **do**
    $k \leftarrow$ # wskaźników $\neq nil$ pośród $\{carry, h[i], h'[i]\}$
    case k **of**
      0: $H[i] \leftarrow nil$
      1: $H[i] \leftarrow$ jedyny niepusty wskaźnik pośród $\{carry, h[i], h'[i]\}$
      2: $H[i] \leftarrow nil; carry \leftarrow join(B^1, B^2)$
         gdzie B^1 i B^2 są drzewami wskazywanymi przez
         dwa niepuste wskaźniki spośród $\{carry, h[i], h'[i]\}$)
      3: $H[i] \leftarrow h[i]; carry \leftarrow join(h'[i], carry)$

Koszt: $O(\log n)$. Korzystamy tu z prostego faktu:

Fakt 21 *Kopiec zawierający n elementów składa się z co najwyżej $\log n$ różnych drzew dwumianowych.*

Operacja *deletemin(h).*

Wska"xnik MIN wskazuje na drzewo dwumianowe B, którego korzeń zawiera najmniejszy element. W stałym czasie usuwamy B z h. Następnie usuwamy korzeń z drzewa B otrzymując rodzinę drzew $B_0, B_1, \ldots, B_{\text{rząd}(B)-1}$. Z drzew tych tworzymy kopiec dwumianowy h' i wykonujemy $meld(h, h')$.

Koszt: $O(\log n)$.

Operacja *Insert(i,h)*

Pojedyncza operacja insert może kosztować $\Omega(\log n)$, np. gdy h zawiera drzewa każdego rzędu. Można jednak pokazać, że czas zamortyzowany można ograniczyć do $O(1)$ (ćwiczenie).

30.6.2 Lazy meld

Chcemy, by wszystkie operacje oprócz *deletemin* kosztowały nas $O(1)$ czasu zamortyzowanego.

Zmieniamy reprezentację kopca: zamiast tablicy wska"xników, drzewa dwumianowe danego kopca łączymy w cykliczną listę dwukierunkową.

Procedura $lazymeld(h, h')$ polega na połączeniu list i aktualizacji wska"xnika MIN. Można tego dokonać w czasie $O(1)$. Teraz jednak kopiec może zawierać wiele drzew tego samego rzędu. Dopiero operacja *deletemin* redukuje liczbę drzew.

Operacja *deletemin(h)*

Usuwamy korzeń x drzewa wskazywanego przez *MIN*, dołączamy poddrzewa wierzchołka x do listy drzew kopca, uaktualniamy wska"xnik *MIN*, a następnie redukujemy liczbę drzew w kopcu. W tym celu wystarczy raz przeglądnąć listę drzew kopca (możemy roboczo wykorzystać tablicę wska"xników na wzór tablicy z wersji eager operacji *meld*).

Pojedyncza operacja *deletemin* może być bardzo kosztowna (nawet $O(n)$ - np. wtedy, gdy kopiec składa się z n drzew jednoelementowych). Pokażemy jednak, że czas zamortyzowany można ograniczyć przez $O(\log n)$.

Utrzymujemy następujący niezmiennik kredytowy:

Każde drzewo kopca ma 1 jednostkę kredytu na swoim koncie.

Operacjom przydzielamy następujące kredyty:

makeheap	-	2
insert	-	2
meld	-	1
findmin	-	1
deletemin	-	$2 \log n$

Kredyty te wystarczają na wykonanie instrukcji niskiego poziomu, związanych z realizacją operacji oraz na utrzymanie niezmiennika kredytowego:

- Operacje *meld* i *findmin* nie zmieniają liczby drzew w kopcach i wykonują się w stałym czasie.

- Operacje *insert* i *makeheap* także wykonują się w stałym czasie, ale tworzą nowe drzewo. Jedna jednostka kredytu przydzielonego tym operacjom zostaje odłożona na koncie tego drzewa.

- Operacja *deletemin* może dodać co najwyżej $\log n$ drzew do kopca. Z kredytu operacji *deletemin* przekazujemy po jednej jednostce na konta tych drzew. Obliczenie nowej wartości wska"xnika *MIN* można wykonać w czasie $O(\log n)$. Podczas redukcji liczby drzew musimy przeglądnąć listę wszystkich drzew kopca i dokonać pewnej liczby operacji *join*. Koszt operacji *join* możemy pominąć, ponieważ każda taka operacja może być opłacona jednostką kredytu znajdującą się na koncie przyłączanego drzewa. Jednostką tą możemy opłacić także koszt odwiedzenia tego drzewa na liście. Odwiedzenie pozostałych drzew musimy opłacić kredytem przydzielonym operacji *deletemin*. Możemy to zrobić, ponieważ takich drzew (tj. tych, które w czasie *deletemin* nie będą podłączone do innego drzewa) jest nie więcej niż różnych rzędów, a więc $O(\log n)$.

Notatki z AiSD. Nr 17 16 marca 2005

Kopce Fibonacciego

IIUWr. II rok informatyki. Przygotował: Krzysztof Loryś

31 Wstęp

Operacją kopcową, której do tej pory nie rozważaliśmy, a która jest ważna w wielu zastosowaniach jest operacja *decrement*(h, p, Δ), polegająca na zmniejszeniu o Δ klucza w elemencie wskazywanym przez p. Wykonywana na kopcach dwumianowych może wymagać czasu $\log n$ (np. zmniejszenie wartości klucza znajdującego się w liściu może spowodować konieczność przesunięcia go aź do korzenia). Taki czas jest nieakceptowalny, gdy liczba operacji *decrement* jest duźa.

Pokaźemy jak w prosty sposób zmodyfikować kopce dwumianowe, by operacja *deletemin* wykonywała się w stałym czasie zamortyzowanym. Otrzymana struktura danych nosi nazwę *kopców Fibonacciego*.

32 Przykład zastosowania - algorytm Dijkstry

Algorytm Dijkstry oblicza najkrótsze odległości wszystkich wierzchołków grafu $G = (V, E)$ od ustalonego wierzchołka s (*"xródła*). Algorytm jest zachłanny. Buduje zbiór X, wierzchołków, których najkrótsza odległość od s jest juź ustalona: rozpoczyna od jednoelementowego zbioru $\{s\}$ i na każdym kroku dokłada wierzchołek spoza X leźący najbliźej s. Do wyznaczenia takiego wierzchołka służą wartości $D(u)$, które w każdej fazie algorytmu równe są długości najkrótszej ścieźki od u do s prowadzącej jedynie przez wierzchołki z X. Do pamiętania tych wartości możemy używać kopca, ponieważ na każdym kroku szukamy wierzchołka o minimalnej wartości D. Zwykłe kopce nie są tu jednak odpowiednie, ponieważ dołączenie nowego wierzchołka u do X może powodować konieczność uaktualnienia (zmniejszenia) wartości pozostających w kopcu dla wszystkich wierzchołków incydentnych z u. W rezultacie na elementach kopca wykonujemy $|E|$ operacji *decrement* i $|V|$ operacji *deletemin*. Zaimplementowanie algorytmu Dijkstry przy zastosowaniu kopców Fibonacciego da w efekcie jego złożoność $O(m + n \log n)$.

```
procedure Dijkstra
    X ← {s}
    D(s) ← ∅
    for each u ∈ V \ {s} do D(u) ← l(s, u)
    while X ≠ V do
        Niech u ∈ V \ X o minimalnej wartości D(u)
        X ← X ∪ {u}
        for each ⟨u, v⟩ ∈ E takiej, że v ∈ V \ X do
                        D(v) ← min(D(v), D(u) + l(u, v))
```

33 Struktura kopców Fibonacciego

Podobnie jak kopce dwumianowe, kopce Fibonacciego są zbiorami drzew, których wierzchołki pamiętają elementy zgodnie z porządkiem kopcowym. Teraz jednak drzewa nie muszą być drzewami dwumianowymi.

Przyjmujemy taki sam sposób pamiętania drzew i elementu minimalnego, jak w przypadku kopców dwumianowych (wersja lazy). Ponadto w każdym wewnętrznym wierzchołku kopca pamiętamy wartość logiczną, mówiacą czy wierzchołek ten utracił jednego ze swoich synów w wyniku operacji *cut* - patrz niżej.

34 Operacje

Operacje *makeheap, insert, findmin* i *meld* wykonujemy w taki sam sposób jak na kopcach dwumianowych.

34.1 Operacja $cut(h, p)$

Operacja ta zastosowana do wierzchołka wewnętrznego wskazywanego przez p, odcina go od swojego ojca p' i dołącza (operacją *meld* poddrzewo zakorzenione w p do listy drzew kopca. Jeśli p jest pierwszym synem jakiego utracił p', to fakt ten jest zapamiętywany w p'. Jeśli p' wcześniej utracił juź jakiegoś syna, to wykonujemy operację $cut(h, p')$. W ten sposób będziemy wędrować w górę drzewa odcinając odpowiednie poddrzewa tak długo, aź napotkamy korzeń lub wierzchołek, który dotąd nie utracił źadnego syna.

34.2 Operacja *decrement*(h, p, Δ)

Zmniejszamy wartość klucza w wierzchołku wskazywanym przez p. Jeśli nowa wartość klucza zakłóca porządek kopcowy (tzn. jest mniejsza od klucza ojca wierzchołka p), wykonujemy $cut(h, p)$.

34.2.1 Zamortyzowany koszt

Teraz każdy wierzchołek ma swoje konto. Będzie ono niepuste tylko u wierzchołków, które utraciły jednego syna.
Operacji *decrement*(h, p, Δ) przydzielamy 4 jednostki kredytu. Jedną jednostką opłacamy koszt instrukcji niskiego poziomu i operację *meld* przyłączenia drzewa o korzeniu w p do kopca. Drugą umieszczamy na koncie tego drzewa (obowiązuje nas w dalszym ciągu niezmiennik kredytowy, mówiący, iź na koncie każdego drzewa kopca znajduje się jedna jednostka). Dwie pozostałe jednostki wykorzystujemy tylko wtedy, gdy wykonujemy $cut(h, p)$ i p jest pierwszym synem odciętym od swojego ojca. Umieszczamy je wówczas na koncie ojca p. Jednostki te są wykorzystywane do opłacenia operacji *cut* wykonanej wskutek tego, źe ojciec p straci drugiego syna.

34.3 Operacja *deletemin*(h)

Deletemin wykonujemy w sposób analogiczny jak w przypadku kopców dwumianowych. W szczególności podczas redukcji łączymy drzewa o jednakowym rzędzie (zdefiniowanym jako liczba synów korzenia), otrzymując drzewo o stopniu o jeden wyższym. Jedyna różnica wynika z tego, źe teraz drzewa nie są dwumianowe i nie można oczekiwać, źe łączone drzewa będą identyczne.

Aby wykazać, źe $O(\log n)$ nadal ogranicza czas wykonywania tej operacji musimy dowieść, źe stopień wierzchołków drzew występujących w kopcach Fibonacciego jest ograniczony przez $O(\log n)$. Oczywiście będzie to także ograniczeniem na liczbę różnych rzędów drzew.

Lemat 1 *Dla każdego wierzchołka x kopca Fibonacciego o rzędzie k, drzewo zakorzenione w x ma rozmiar wykładniczy względem k.*

Dowód: Niech x będzie dowolnym wierzchołkiem kopca i niech $y_1, \ldots, y_k$ będą jego synami uporządkowanymi w kolejności przyłączania ich do x. W momencie przyłączania y_i do x-a, x miał co najmniej $i-1$ synów. Stąd y_i też miał wówczas co najmniej $i-1$ synów, ponieważ przyłączane są tylko drzewa o jednakowym rzędzie. Od tego momentu y_i mógł stracić co najwyźej jednego syna, ponieważ w przeciwnym razie zostałby odcięty od x-a. Tak więc w każdym momencie i-ty syn każdego wierzchołka ma rząd co najmniej $i-2$.
Oznaczmy przez F_i najmniejsze drzewo o rzędzie i, spełniające powyższą zależność. Łatwo sprawdzić, źe F_0 jest drzewem jednowierzchołkowym, a F_i składa się z korzenia oraz i poddrzew: $F_0, F_0, F_1, F_2, \ldots, F_{i-2}$. Tak więc liczba $|F_i|$ wierzchołków takiego drzewa jest nie mniejsza niź $1 + \sum_{j=0}^{i-2} |F_j|$, co, jak łatwo pokazać indukcyjnie, jest równe i-tej liczbie Fibonacciego. Stąd liczba wierzchołków w drzewie o rzędzie k jest nie mniejsza niź ϕ^k, gdzie $\phi = (1+\sqrt{5})/2$. □

Wniosek 2 *Każdy wierzchołek w n-elementowym kopcu Fibonacciego ma stopień ograniczony przez $O(\log n)$.*

34.3.1 Operacja *delete*(h, p)

Operację *delete*(h, p) można wykonać najpierw ustanawiając w p minimum kopca (poprzez operację *decrement*$(h, p, -\infty)$) a następnie usuwając minimum. Zamortyzowany koszt wynosi $O(\log n)$.
Uwaga: W ten sam sposób możemy wykonywać *delete* na kopcach dwumianowych. Oczywiście wówczas *decrement* musi polegać na przesunięciu zmniejszonego elementu do korzenia drzewa.

Notatki z AiSD. Nr ?. 16 marca 2005

Drzewa samoorganizujące się

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

35 Wprowadzenie

Drzewa samoorganizujące się są kolejnym przykładem struktury danych opartej na binarnych drzewach przeszukiwań. Przymiotnik "samoorganizujące" oznacza, źe drzewa te w trakcie wykonywania na nich operacji zmieniają swoją strukturę automatycznie, stosując pewną prostą heurystykę. W przeciwieństwie do drzew zbalansowanych (AVL, czerwono-czarnych) heurystyka ta nie korzysta z źadnych dodatkowych informacji pamiętanych w wierzchołkach. Druga istotna róznica polega na tym, źe teraz pojedyncze operacje słownikowe mogą być kosztowne. Jak jednak pokaźemy, zamortyzowany koszt ciągu operacji jest niski.

36 Operacje na drzewach samoorganizujących się

Oprócz operacji słownikowych ($find(i, S)$, $insert(i, S)$, $delete(i, S)$, odpowiednio odszukiwania, wstawiania i usuwania klucza i w (do, z) drzewie S) rozwaźymy realizację następujących operacji:

- $join(S_1, S_2)$ - połącz drzewa S_1 i S_2 w jedno drzewo (przy założeniu, źe każdy klucz w drzewie S_1 jest nie większy od każdego klucza z drzewa S_2),
- $split(i, S)$ - rozdziel S na dwa drzewa S_1 i S_2 takie, źe każdy klucz w S_1 jest nie większy od i, a każdy klucz w S_2 jest nie mniejszy od i.

37 Implementacja operacji

Podstawową idea drzew samoorganizujących się polega na tym, by wierzchołki drzewa zawierające klucz i (parametr operacji *insert, delete, find, split*) przesuwać serią rotacji do korzenia. Umiejętnie wykonywane rotacje będą powodować "spłaszczenie" drzewa.

Wygodnie jest nam wprowadzić operację *splay*, w terminach której wyrazimy wszystkie interesujące nas operacje.

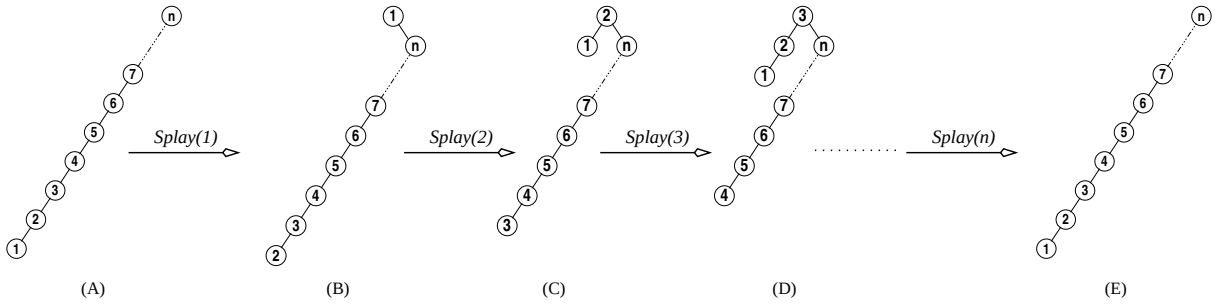
Definicja 17 *$splay(j, S)$ - przeorganizuj S tak, by jego korzeniem stał się wierzchołek zawierający k takie, źe w S nie ma elementu leźącego między k i j.*

Tak więc jeśli j znajduje się w S to operacja $splay(j, S)$ przesunie j do korzenia. W przeciwnym razie w korzeniu znajdzie się $k = \min\{x \in S | x > j\}$ lub $k = \max\{x \in S | x < j\}$.

38 Implementacja Splay(x)

Splay łatwo jest zaimplementować przy pomocy rotacji. Jedną z moźliwości jest stosowanie rotacji do elementu x tak długo, aź znajdzie się on w korzeniu. Jak jednak pokazuje poniższy przykład, taka implementacja powoduje, źe niektóre ciągi operacji słownikowych byłyby wykonywane w czasie kwadratowym od długości ciągu.

Przykład 1



Drzewo (A) moźe powstać na skutek wykonania ciągu instrukcji: $insert(1)$, $insert(2)$, ..., $insert(n)$. Kolejne operacje: $splay(1), splay(2), \ldots splay(n-1)$ wykonują odpowiednio: $n-1$, $n-1$, $n-2$, $n-3, \ldots, 1$ rotacji. Po wykonaniu $Splay(n)$ otrzymujemy z powrotem drzewo (A). □

Wobec tego musimy zaproponować inny sposób implementacji. Rozważamy 3 przypadki:

(a) x ma ojca, ale nie ma dziadka → $rotate(x)$,

(b) x ma ojca $p(x)$ i ma dziadka; x i $p(x)$ są obydwaj lewymi bąd"x obydwaj prawymi synami swoich ojców → $rotate(y)$; $rotate(x)$,

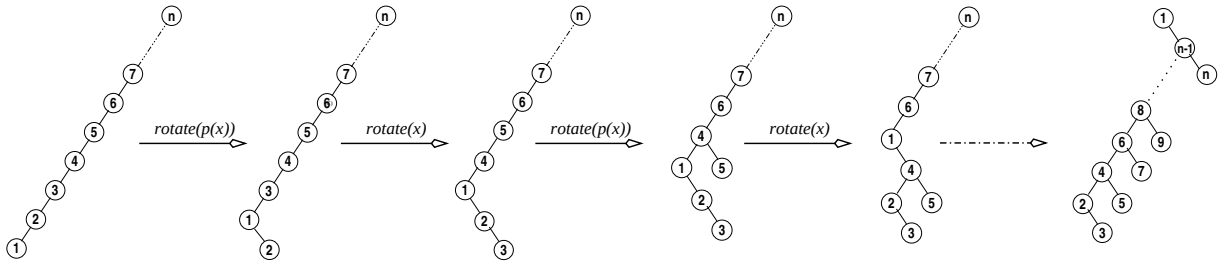
(c) x ma ojca $p(x)$ i ma dziadka; x jest lewym a $p(x)$ prawym synem, bąd"x na odwrót → $rotate(x)$; $rotate(x)$.

Przykład 2

39 Analiza

Stosujemy analizę zamortyzowaną. Każdy wierzchołek drzewa przechowuje pewien depozyt. Operacja wykonywana na drzewie moźe zwiększać depozyty, bad"x teź moźe być opłacana przez kwoty z depozytów.

Oznaczenia



$S(x)$ - poddrzewo o korzeniu w x,

$|S|$ - liczba wierzchołków w drzewie S,

$\mu(S) = \lfloor \log(|S|) \rfloor$,

$\mu(x) = \mu(S(x))$.

Będziemy utrzymywać następujący niezmiennik:

Wierzchołek x ma zawsze co najmniej $\mu(x)$ jednostek na swoim koncie.

Insert daje wierzchołkowi pewien początkoy depozyt.

Lemat 2 *Każda operacja* Splay(x,S) *wymaga nie więcej niż* $3(\mu(S) - \mu(x)) + 1$ *jednostek do wykonania operacji i zachowania niezmiennika kredytowego.*

Dowód: Niech y będzie ojcem x-a, a z - ojcem y-ka (o ile on istnieje). Niech ponadto μ oraz μ' oznaczają odpowiednio depozyty przed i po wykonaniu operacji *splay*.

(a) z nie istnieje. W tym przypadku wykonujemy pojedynczą rotację *rotate(x)*:

rysunek

Jak łatwo widać: $\mu'(x) = \mu(y)$, $\mu'(x) \geq \mu(x)$ oraz $\mu'(y) \leq \mu'(x)$.

Aby utrzymać niezmiennik musimy zapłacić:

$$\mu'(x) + \mu'(y) - \mu(x) - \mu(y) = \mu'(y) - \mu(x) \leq \mu'(x) - \mu(x) \leq 3(\mu'(x) - \mu(x)).$$

Mając do dyspozycji $3(\mu'(x) - \mu(x)) + 1$ jednostek jesteśmy w stanie utrzymać niezmiennik i pozostanie nam jeszcze jedna jednostka na opłacenie operacji niskiego poziomu związanych z wykonaniem *splay* (manipulacje wska"xnikami, porównania,...).

(b) Mamy

rysunek

Pokażemy, że *rotate*(y)*; rotate*(x) oraz utrzymanie niezmiennika kosztują nie więcej niż $3(\mu'(x) - \mu(x)$.

Aby utrzymać niezmiennik potrzebujemy:

$$(*) = \mu'(x) + \mu'(y) + \mu'(z) - \mu(x) - \mu(y) - \mu(z)$$

jednostek. Ponieważ

rysunek

mamy $\mu'(x) = \mu(z)$. Stąd

$$(*) = \mu'(y) + \mu'(z) - \mu(x) - \mu(y) = [\mu'(y) - \mu(x)] + [\mu'(z) - \mu(y)] \leq$$

$$\leq [\mu'(x) - \mu(x)] + [\mu'(x) - \mu(y)] \leq 2[\mu'(x) - \mu(x)].$$

Mając $3[\mu'(x)-\mu(x)]$ jednostek do dyspozycji, na opłacenie operacji niskiego poziomu wykonywanych przy tych dwóch rotacjach pozostaje nam $\mu'(x) - \mu(x)$ jednostek. Może się jednak okazać, że $\mu'(x) = \mu(x)$. Pokażemy, że wówczas (*) jest ujemna i dlatego w tym przypadku niezmiennik mamy utrzymany bez ponoszenia kosztów a nawet możemy uszczuplić

(c) Podobnie jak (b).

W trakcie operacji *Splay(x,S)* x zajmuje coraz wyższe pozycje. Niech $S_1, S_2, \ldots, S_k$ będą drzewami zakorzenionymi w x w momencie gdy x zajmuje te pozycje. Wówczas całkowity koszt *Splay(x,S)* wynosi

$$3(\mu(S_1) - \mu(x)) + 3(\mu(S_2) - \mu(S_1)) + \ldots + 3(\mu(S_k) - \mu(S_{k-1}) + 1 =$$

$$3(\mu(S_k) - \mu(x)) + 1 = 3(\mu(S) - \mu(x)) + 1$$

□

Literatura

[1] D.Sleator, R.E.Tarjan, *Self-adjusting binary trees,* JACM, 32(1985), s. 652-686.

[2] R.E.Tarjan, *Data Structures and Network Algorithms,* SIAM, 1983.

Notatki z AiSD. Nr 18 16 marca 2005

UNION-FIND

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

40 Definicja problemu

Dany jest skończony zbiór U oraz ciąg σ instrukcji UNION i FIND:

- UNION(A, B, C); gdzie A, B - rozłączne podzbiory U;
  wynikiem instrukcji jest utworzenie zbioru C takiego, że $C \leftarrow A \cup B$, oraz usunięcie zbiorów A i B;
- FIND(i); gdzie $i \in U$;
  wynikiem instrukcji jest nazwa podzbioru, do którego aktualnie należy i.

Problem polega na zaprojektowaniu struktury danych umożliwiającej szybkie wykonywanie ciągów σ. Początkowo każdy element U tworzy jednoelementowy podzbiór.

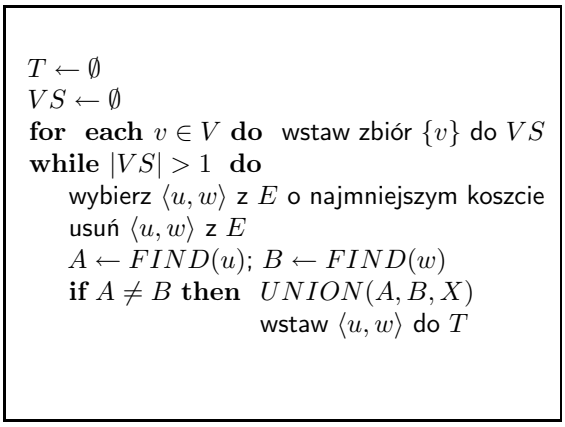
40.1 Uwagi i założenia

- Zbiór U jest mały ($|U| \ll$ pojemność pamięci wewnętrznej). Zwykle przyjmuje się, że $U = \{1, \ldots, n\}$.
- Bardzo często σ zawiera cn instrukcji (c-stała).
- Rozważa się dwa sposoby wykonywania ciągów σ:
  - *on-line* - wynik każdej instrukcji musi zostać obliczony przed wczytaniem kolejnej instrukcji;
  - *off-line* - ciąg σ może być wczytany całkowicie zanim zostanie obliczony wynik którejkolwiek instrukcji.

  Nas interesować będzie sposób *on-line*.
- Często nazwy podzbiorów są nieistotne, a instrukcja FIND służy jedynie do stwierdzenia czy dane elementy należą do tego samego podzbioru.

41 Przykład zastosowania

41.1 Konstrukcja minimalnego drzewa rozpinającego grafu



42 Rozwiązania

42.1 Proste rozwiązanie

Do reprezentowania rodziny zbiorów używamy tablicy $R[1..n]$ takiej, że

$$\forall_i \quad R[i] \text{ jest nazwą zbioru zawierającego } i.$$

Koszt: FIND - $\Theta(1)$; UNION -$\Theta(n^2)$.

42.2 Modyfikacja prostego rozwiązania

42.2.1 Idea

Oparta na dwóch trickach:

- Wprowadzamy nazwy wewnętrzne zbiorów (niewidoczne dla użytkownika).
- Podczas wykonywania UNION(A, B, C) zbiór mniejszy przyłączany jest do większego.

42.2.2 Realizacja

Używamy tablic: $R, ExtName, IntName, List, Next$ i $Size$ takich, że:

$R[i]$	=	nazwa wewnętrzna zbioru zawierającego i,
$ExtName[j]$	=	nazwa zewnętrzna zbioru o nazwie wewnętrznej j,
$IntName[k]$	=	nazwa wewnętrzna zbioru o nazwie zewnętrznej j,
$List[j]$	=	wska"xnik na pierwszy element w liście elementów zbioru o nazwie wewnętrznej j,
$Next[i]$	=	następny po i element w liście elementów zbioru $R[i]$,
$Size[j]$	=	liczba elementów w zbiorze o nazwie wewnętrznej j.

```
procedure Find(i)
    return (ExtName(R[i]))


procedure UNION(I, J, K)
    A ← IntName[I]
    B ← IntName[J]
    Niech Size[A] ≤ Size[B]; w p.p. zamień A i B rolami
    el ← List[A]
    while el ≠ 0 do  R[el] ← B
                     last ← el
                     el ← Next[el]
    Next[last] ← List[B]
    List[B] ← List[A]
    Size[B] ← Size[A] + Size[B]
    IntName[K] ← B
    ExtName[B] ← K
```

Twierdzenie 13 *Używając powyższego algorytmu można wykonać dowolny ciąg σ o długości $O(n)$ w czasie $O(n \log n)$.*

43 Struktury drzewiaste dla problemu Union-Find

43.1 Elementy składowe struktury danych

- Las drzew.
  Każdy podzbiór reprezentowany jest przez drzewo z wyróżnionym korzeniem. Wierzchołki wewnętrzne zawierają wska"xnik na ojca (nie ma wska"xników na dzieci!).

- Tablica $Element[1..n]$:

$$Element[i] = \text{ wska"xnik na wierzchołek zawierający } i.$$

- Tablica *Root*:

$$Root[I] = \text{ wska"xnik na korzeń drzewa odpowiadającego zbiorowi } I$$

  (nazwy zbiorów są dla nas nieistotne; będą one liczbami z $[1, .., n]$).

43.2 Realizacja instrukcji

$Union(A, B, C)$ polega na połączeniu drzew odpowiadających zbiorom A i B w jedno drzewo i umieszczeniu w jego korzeniu nazwy C.

$Find(i)$ polega na przejściu ścieżki od wierzchołka wskazywanego przez $Element(i)$ do korzenia drzewa i odczytaniu pamiętanej tam nazwy drzewa. Przy wykonywaniu tych instrukcji stosujemy następującą strategię:

1. instrukcję $Union$ wykonujemy w sposób zbalansowany - korzeń mniejszego (w sensie liczby wierzchołków) drzewa podwieszamy do korzenia drzewa większego (a dokładniej drzewa nie większego do korzenia drzewa nie mniejszego),

2. podczas instrukcji $Find(i)$ wykonujemy *kompresję ścieżki* prowadzącej od i do korzenia - wszystkie wierzchołki leżące na tej ścieźce podwieszamy bezpośrednio pod korzeń.

43.3 Implementacja

Każdy wierzchołek v zawiera pola:

- $Father[v]$ - wska"xnik na ojca (równy NIL, gdy v jest korzeniem),
- $Size[v]$ - liczba wierzchołków w drzewie o korzeniu v,
- $Name[v]$ - nazwa drzewa o korzeniu v

Zawartość pól $Size[v]$ i $Name[v]$ ma znaczenie tylko wówczas, gdy v jest korzeniem.

```
procedure InitForest
    for i ← 1 to n do v ← Allocate − Node()
                       Size[v] ← 1
                       Name[v] ← i
                       Father[v] ← NIL
                       Element[i] ← v
                       Root[i] ← v
```

```
procedure Union(i, j, k)
    Niech Size[Root[i]] ≤ Size[Root[j]]; w p.p. zamień i oraz j rolami
    large ← Root[j]
    small ← Root[i]
    Father[small] ← large
    Size[large] ← Size[large] + Size[small]
    Name[large] ← k
    Root[k] ← large
```

```
procedure Find(i)
    list ← NIL
    v ← Element[i]
    while Father[v] ≠ NIL do wstaw v na list
                          v ← Father[v]
    for each w na list do Father[w] ← v
    return Name[v]
```

43.4 Analiza algorytmu

Lemat 3 *Jeśli instrukcje Union wykonujemy w sposób zbalansowany, to każde powstające drzewo o wysokości h ma co najmniej 2^h wierzchołków.*

Definicja 18 *Niech $\tilde{\sigma}$ będzie ciągiem instrukcji Union powstałym po usunięciu wszystkich instrukcji Find z ciągu σ.* Rzędem *wierzchołka v względem σ nazywamy jego wysokość w lesie powstałym po wykonaniu ciągu $\tilde{\sigma}$.*

Lemat 4 *Jest co najwyżej $\frac{n}{2^r}$ wierzchołków rzędu r.*

Wniosek 3 *Każdy wierzchołek ma rząd co najwyżej r.*

Lemat 5 *Jeśli w trakcie wykonywania ciągu σ wierzchołek w staje się potomkiem wierzchołka v, to rząd w jest mniejszy niż rząd v.*

Definicja 19

$$\log^*(n) \stackrel{df}{=} \min\{k \mid F(k) \geq n\},$$

gdzie $F(0) = 1$ i $F(i) = 2^{F(i-1)}$ dla $i > 0$.

Twierdzenie 14 *Niech c będzie dowolną stałą. Wówczas istnieje inna stała c' (zależna od c) taka, że powyższe procedury wykonują dowolny ciąg σ złożony z cn instrukcji Union i Find w czasie $c'n \log^* n$.*

Twierdzenie 15 *Algorytm realizujący ciągi instrukcji Union i Find przy użyciu powyższych procedur ma złożoność większą niż cn dla dowolnej stałej c.*

UWAGA: na ćwiczeniach pokażemy, że przy pomocy struktur drzewiastych można w czasie $O(n \log^* n)$ realizować ciągi σ, które oprócz instrukcji *Union* i *Find* zawierają także instrukcje *Insert* i *Delete*.

Notatki z AiSD. Nr 19. 16 marca 2005

Mnożenie macierzy

IIUWr. II rok informatyki. Przygotował: Krzysztof Loryś

44 Metoda Strassena

Problem:

- Dane są dwie macierze A i B o rozmiarach $n \times n$, elementów z pierścienia R.
- Chcemy obliczyć $C = A \cdot B$.

Idea.
Stosujemy strategię Dziel i Rząd"x:
Dzielimy macierze A, B i C na cztery podmacierze o rozmiarze $\frac{n}{2} \times \frac{n}{2}$ każda (dla prostoty zakładamy, że n jest potęgą liczby 2):

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \qquad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix},$$

a następnie obliczamy niezależnie podmacierze C_{ij}.

Podmacierze te możemy obliczyć korzystając z oczywistych wzorów: $\forall_{i=1,2;\, j=1,2}\ C_{ij} = A_{i1} \cdot B_{1j} + A_{i2} \cdot B_{2j}$. Czyli jedno mnożenie macierzy $x \times n$ zastępujemy ośmioma mnożeniami macierzy $\frac{n}{2} \times \frac{n}{2}$. Tak otrzymany algorytm ma niestety złożoność $\Omega(n^3)$, czyli taką samą jak tradycyjny algorytm.

Kluczem do przyśpieszenia algorytmu jest zredukowanie liczby mnożeń macierzy $\frac{n}{2} \times \frac{n}{2}$ z ośmiu do siedmiu.

Algorytm *Metoda Strassena*

1. Jeśli n jest małe oblicz iloczyn $A \cdot B$ metodą tradycyjną.
   W przeciwnym przypadku:
2. Oblicz 7 pomocniczych macierzy m_i o rozmiarze $\frac{n}{2} \times \frac{n}{2}$.

$$
\begin{aligned}
m_1 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\
m_2 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
m_3 &= (A_{11} - A_{21}) \cdot (B_{11} + B_{12}) \\
m_4 &= (A_{11} + A_{12}) \cdot B_{22} \\
m_5 &= A_{11} \cdot (B_{12} - B_{22}) \\
m_6 &= A_{22} \cdot (B_{21} - B_{11}) \\
m_7 &= (A_{21} + A_{22}) \cdot B_{11}
\end{aligned}
$$

3. Oblicz składowe podmacierze C_{ij} macierzy wynikowej C.

$$
\begin{aligned}
C_{11} &= m_1 + m_2 - m_4 + m_6 \\
C_{12} &= m_4 + m_5 \\
C_{21} &= m_6 + m_7 \\
C_{22} &= m_2 - m_3 + m_5 - m_7
\end{aligned}
$$

Twierdzenie 6 *Powyższy algorytm oblicza iloczyn dwóch dowolnych macierzy $n \times n$ o elementach z dowolnego pierścienia za pomocą $O(n^{\log_2 7})$ operacji arytmetycznych na elementach tego pierścienia.*

UWAGI:

- Stała skryta pod "dużym O" czyni algorytm Strassena niepraktycznym dla macierzy małych rozmiarów oraz dla macierzy specjalnych postaci, dla których opracowano szybkie algorytmy mnożenia (np. dla macierzy rzadkich).
- Nie można zmniejszyć liczby mnożeń macierzy 2×2 do 6-iu.
- Obecnie najszybszy asymptotycznie algorytm mnożenia macierzy ([1]) działa w czasie $O(n^{2.376})$.
- Nieznane jest obecnie ograniczenie dolne na złożoność problemu mnożenia macierzy lepsze niż trywialne ograniczenie $\Omega(n^2)$.

45 Mnożenie macierzy logicznych

45.1 Metoda wykorzystująca metodę Strassena

Metody Strassena nie można wykorzystać bezpośrednio do mnożenia macierzy logicznych, ponieważ $\langle \{\mathbf{0}, \mathbf{1}\}; \vee, \wedge \rangle$ nie stanowi pierścienia. Istnieje jednak prosty sposób obejścia tego problemu:
Traktujemy macierze logiczne A i B jako macierze nad $\mathcal{Z}_{n+1}$ ($\mathbf{0}$ i $\mathbf{1}$ traktujemy odpowiednio jako 0 i 1 z $\mathcal{Z}_{n+1}$). Mnożymy A i B w $\mathcal{Z}_{n+1}$ (tj. zastępując $\vee$ przez sumę modulo $(n+1)$ a

$\wedge$ przez iloczyn modulo $(n+1)$. Jeśli tak otrzymany iloczyn oznaczymy przez C a iloczyn logiczny przez D to mamy:

Fakt 22 $\forall_{1\leq i,j\leq n}\ D[i,j] = \mathbf{1}$ *iff* $C[i,j] \neq 0$.

Koszt:

- $O(n^{2.81})$ operacji arytmetycznych w $\mathcal{Z}_{n+1}$.
- $O(n^{2.81}\log(n)\log\log(n)\log\log\log(n))$ operacji na bitach.
  Uzasadnienie: Dodawanie i odejmowanie liczb k-bitowych wymaga $O(k)$ operacji bitowych, zaś mnożenie metodą Schönchagego-Strassena - $O(k\log k\log\log k)$ operacji bitowych. My wszystkie operacje wykonujemy na elementach z $\mathcal{Z}_n$, a więc na liczbach $\log n$ bitowych.

45.2 Metoda czterech Rosjan

Metoda Czterech Rosjan daje algorytm o nieco gorszej złożoności od wyżej opisanej metody. Jest ona jednak atrakcyjna ze względu na moźliwość wykorzystania operacji na wektorach bitów (realizowalnych hardware'owo), co wydatnie zwiększa jej praktyczną szybkość.

Idea:
Macierz A dzielimy na $\frac{n}{\log n}$ podmacierzy A_i o rozmiarze $n \times \log n$ każda, a macierz B na $\frac{n}{\log n}$ podmacierzy B_i o rozmiarze $\log n \times n$ każda (zakładamy dla prostoty opisu, źe $\log n$ jest liczbą całkowitą dzielącą n). Podmacierz A_i (B_i) składa się z kolejnych kolumn (wierszy) macierzy A (macierzy B) o numerach od $(\log n)(i-1)+1$ do $(\log n)i$. Łatwo sprawdzić,źe $\forall_{i=1,\ldots,n/\log n}\ A_i \cdot B_i$ jest macierzą $n \times n$ oraz źe

$$A \cdot B = \sum_{i=1}^{n/\log n} A_i \cdot B_i.$$

Kluczowym trickiem jest metoda obliczania iloczynów $A_i \cdot B_i$ w czasie $O(n^2)$.

Spostrzeźenie:

1. Jeśli j-ty wiersz macierzy A_i składa się z samych zer, to j-ty wiersz macierzy C_i również składa się z samych zer.
2. Jeśli wiersze j_1 i j_2 macierzy A_i różnią się tylko na pozycji k-tej, przy czym wiersz j_1 ma na tej pozycji zero, to wiersz j_2 macierzy C_i jest równy sumie logicznej wiersza j_1 macierzy C_i oraz wiersza k macierzy B_i.

Wiersze macierzy A_i są wektorami z $\{\mathbf{0},\mathbf{1}\}^{\log n}$. Różnych takich wektorów jest n. Na podstawie Spostrzeźenia, wszystkie iloczyny postaci $\mathbf{x} \cdot B_i$, gdzie $\mathbf{x} \in \{\mathbf{0},\mathbf{1}\}^{\log n}$, moźna obliczyć w czasie $O(n^2)$.

45.2.1 Algorytm

Oznaczenia:

- $\mathbf{b}_s^i$ - s-ty wiersz macierzy B_i,
- $\mathbf{a}_j$ - j-ty wiersz macierzy A,
- jeśli $\mathbf{v} \in \{\mathbf{0},\mathbf{1}\}^n$, to $\tilde{\mathbf{v}}$ - odpowiadający mu wektor w $\{0,1\}^n$, a $num(\mathbf{v}) = \sum_{i=1}^{n} \tilde{\mathbf{v}}_i 2^{n-i}$.

Algorytm *CzterechRosjan*

procedure $LogMatMult(A, B : array[1..n, 1..n]\ of\ boolean)$
  for $i \leftarrow 1$ **to** $\frac{n}{\log n}$ **do**
    $Rowsum[0] \leftarrow [\underbrace{0, 0, \ldots, 0}_{n \text{ razy}}]$
    for $j \leftarrow 1$ **to** n **do**
      $k \leftarrow \max\{l \mid j \geq 2^l\}$
      $Rowsum[j] \leftarrow Rowsum[j - 2^k] + \mathbf{b}_{k+1}^i$ {sumowanie po współrzędnych}
    niech C_i będzie macierzą, której j-ty wiersz
    jest równy $Rowsum[num(\mathbf{a_j})]$ $(j = 1, \ldots, n)$
  return $C = \sum_{i=1}^{\frac{n}{\log n}} C_i$

Komentarz: Wektory z $\{\mathbf{0},\mathbf{1}\}^{\log n}$ utożsamiamy z liczbami $\log n$-bitowymi. Dzięki temu łatwo możemy takimi wektorami indeksować tablicę $Rowsum$. Dla ustalonego i, $Rowsum[j]$ będzie pamiętał iloczyn wektora odpowiadającego liczbie j oraz macierzy B_i. Jak łatwo sprawdzić, wektory odpowiadające j oraz $j - 2^k$ różnią się tylko na jednej pozycji i $Rowsum[j]$ możemy obliczyć przez dodanie k-ego wiersza macierzy B_i $Rowsum[j - 2^k]$.

Fakt 23 *Powyższy algorytm oblicza $C = A \cdot B$ w czasie $O(n^3/\log n)$. Ponadto można go zaimplementować tak, by wymagał $O(n^2/\log n)$ operacji na wektorach bitów.*

Literatura

[1] D.Coppersmith, S.Winograd, Matrix multiplication via arithmetic progressions, w: *Proceedings of 19th STOC*, 1987, s. 1–6.

Notatki z AiSD. Nr ?? 16 marca 2005

Szybka transformacja Fouriera (FFT)

IIUWr. II rok informatyki. *Opracował: Krzysztof Loryś*

46 Reprezentacje wielomianów

Dwie reprezentacje wielomianu A stopnia $n-1$:

`[Wsp]` jako n-elementowy *wektor współczynników* $\langle a_0, a_1, \ldots, a_{n-1}\rangle$.

`[War]` jako *zbiór wartości* w n różnych punktach $\{(x_i, y_i) : i = 0, \ldots, n-1$ i $\forall_{0\le i\neq j\le n-1}\ x_i \neq x_j$ i $y_i = A(x_i)\}$.

47 Podstawowe operacje na wielomianach

- dodawanie - wykonalne w czasie $O(n)$ przy obydwu reprezentacjach wielomianów,
- mnożenie - łatwe przy reprezentacji `[War]` (w czasie $O(n)$); trudne przy reprezentacji `[Wsp]` (prosta implementacja wymaga czasu $\Omega(n^2)$.
- obliczanie wartości w punkcie - łatwe przy reprezentacji `[Wsp]` (np. schemat Hornera - w czasie $O(n)$); trudne przy reprezentacji `[War]`

48 Zmiana reprezentacji wielomianu stopnia $n-1$

`[Wsp]` $\rightarrow$`[War]`

Reprezentacja `[War]` może być wybrana na wiele róznych sposobów. Korzystając ze schematu Hornera można ją obliczyć w czasie $\Theta(n^2)$.

`[War]` $\rightarrow$`[Wsp]`

Twierdzenie 16 *Dla każdego zbioru* $\{\langle x_i, y_i\rangle \mid i = 0, \ldots, n-1$ *oraz* $\forall_{0\le i\neq j\le n-1}\ x_i \neq x_j\}$ *istnieje jednoznacznie wyznaczony wielomian* A *stopnia* $n-1$ *taki, że* $\forall_{0\le i\le n-1}\ y_i = A(x_i)$.

Współczynniki tego wielomianu można obliczyć w czasie $\Theta(n^2)$ ze wzoru Lagrange'a:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j\neq k}(x - x_j)}{\prod_{j\neq k}(x_k - x_j)}.$$

Jak pó"xniej pokaźemy przejścia `[Wsp]` $\rightarrow$`[War]` i `[War]` $\rightarrow$ `[Wsp]` , można obliczyć w czasie $O(n \log n)$.

49 Pomysł na szybkie mnożenie wielomianów w postaci [Wsp]

Niech $A(x)$ i $B(x)$ będą wielomianami stopnia $\leq n-1$.

1. Utworzyć reprezentacje [Wsp] wielomianów A i B jako wielomianów stopnia $2n-1$ (przez dodanie n współczynników równych 0).
2. Stosując FFT obliczyć dla tych wielomianów reprezentacje [War] o długości $2n$.
3. Obliczyć reprezentację [War] wielomianu $C(x) = A(x) \cdot B(x)$.
4. Stosując FFT obliczyć reprezentację [Wsp] wielomianu $C(x)$.

Kroki 1 i 3 można wykonać w czasie $O(n)$, a kroki 2 i 4 w czasie $O(n \log n)$.

50 Pierwiastki z jedności w ciele liczb zespolonych

Definicja 20 *n-tym* pierwiastkiem z jedności *nazywamy liczbę ω taką, że $\omega^n = 1$.*

Fakt 24 *W ciele liczb zespolonych istnieje dokładnie n n-tych pierwiastków z jedności. Są nimi liczby $e^{2\pi ik/n}$ dla $i = 0, \ldots, n-1$.*

Definicja 21 *n-ty pierwiastek z jedności, którego potęgi generują zbiór wszystkich n-tych pierwiastków nazywamy* n-tym pierwotnym pierwiastkiem z jedności.

Fakt 25 *Liczba $\omega_n = e^{2\pi i/n}$ jest n-tym pierwotnym pierwiastkiem z jedności.*

Fakt 26 *Zbiór $\{\omega_n^j \mid j = 0, \ldots, n-1\}$ z mnożeniem tworzy grupę izomorficzną z grupą $(\mathcal{Z}_n, +_{mod\ n})$.*

Lemat 6 *(a)* $\forall_{n\geq 0, k\geq 0, d>0}\ \omega_{dn}^{dk} = \omega_n^k$.

(b) $\forall_{parzystego\ n>0}\ \omega_n^{n/2} = \omega_2 = -1$.

(c) $\forall_{parzystego\ n>0}\ \{(\omega_n^j)^2 \mid j = 0, \ldots, n-1\} = \{\omega_{n/2}^l \mid l = 0, \ldots, \frac{n}{2} - 1\}$.

(d) $\forall_{n\geq 1, k\geq 0 takiego, że\ n \not| k}\ \sum_{j=0}^{n-1} (\omega_n^k)^j = 0$.

51 Dyskretna Transformacja Fouriera (DFT).

Definicja 22 *Niech $\mathbf{a} = a_0, \ldots, a_{n-1}$. Wektor $\mathbf{y} = y_0, \ldots, y_{n-1}$ taki, że $y_k = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$ (dla $k = 0, \ldots, n-1$) nazywamy Dyskretną Transformacją Fouriera wektora $\mathbf{a}$.*

Jeśli $\mathbf{a}$ jest wektorem współczynników wielomianu $A(x)$, to $\mathbf{y}$ jest wektorem wartości tego wielomianu w punktach $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$.

52 FFT - szybki algorytm obliczania DFT

- Idea algorytmu.

  Niech
  $A^{[0]}(z) = a_0 + a_2 z + a_4 z^2 \ldots + a_{n-2} z^{n/2-1}$ i $A^{[1]}(z) = a_1 + a_3 z + a_5 z^2 \ldots + a_{n-1} z^{n/2-1}$.
  Wówczas $A(x) = A^{[0]}(x^2) + xA^{[0]}(x^2)$.

  Tak więc problem obliczenia wartości wielomianu A stopnia $n-1$ w n punktach: $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$, redukuje się do problemu obliczenia wartości dwóch wielomianów $A^{[0]}$ i $A^{[1]}$ stopnia $\frac{n}{2} - 1$ w $\frac{n}{2}$ punktach: $\omega_{n/2}^0, \omega_{n/2}^1, \ldots, \omega_{n/2}^{(n/2)-1}$.

-
```
procedure Recursive − FFT(a)
    n ← length(a)
    if n = 1 then return (a)
    ω_n ← e^{2πi/n}
    ω ← 1
    a^[0] ← ⟨a_0, a_2, ..., a_{n-2}⟩
    a^[1] ← ⟨a_1, a_3, ..., a_{n-1}⟩
    y^[0] ← Recursive − FFT(a^[0])
    y^[1] ← Recursive − FFT(a^[1])
    for k ← 0 to n/2 − 1 do
        y_k ← y_k^[0] + ω y_k^[1]
        y_{k+(n/2)} ← y_k^[0] − ω y_k^[1]
        ω ← ω ω_n
    return y
```

- Złożoność algorytmu: $T(n) = 2T(\frac{n}{2}) + \Theta(n) = \Theta(n \log n)$.

Definicja 23 Splotem *wektorów* $\mathbf{a} = \langle a_0, \ldots, a_{n-1} \rangle$ *i* $\mathbf{b} = \langle b_0, \ldots, b_{n-1} \rangle$ *nazywamy wektor* $\mathbf{c} = \langle c_0, \ldots, c_{2n-1} \rangle$ *taki, że* $\forall_{0 \le i \le 2n-1}\ c_i = \sum_{j=0}^{i} a_j b_{i-j}$ *i oznaczamy go* $\mathbf{c} = \mathbf{a} \otimes \mathbf{b}$.

Tak więc splot $\mathbf{a} \otimes \mathbf{b}$ jest reprezentacją `[Wsp]` iloczynu wielomianów o reprezentacjach `[Wsp]` **a** i **b**.

53 Interpolacja w n-tych pierwiastkach z jedności

Jeśli $\mathbf{y} = DFT(\mathbf{a})$, to $\mathbf{y} = V_n \cdot \mathbf{a}$, gdzie V_n jest macierzą $n \times n$, której wyraz (j,k)-ty równa się ω_n^{jk}.

Fakt 27 *Dla* $j, k = 0, \ldots, n-1$ *wyraz* (j,k)*-ty macierzy* V_n^{-1} *równa się* ω_n^{-jk}/n.

Powyższy fakt pozwala na obliczenie **a** z danego **y** przez zastosowanie FFT (należy ω_n zastąpić przez ω_n^{-1})

54 Efektywna implementacja FFT

```
procedure Iterative − FFT(a)
   Bit − Reverse − Copy(a, A)
   n ← length(a)                          { n jest potęgą 2-ki }
   for s ← 1 to log n do
         m ← 2^s
         ω_m ← e^{2πi/m}
         ω ← 1
         for j ← 0 to m/2 − 1 do
               for k ← j to n − 1 step m do
                     t ← ωA[k + m/2]
                     u ← A[k]
                     A[k] ← u + t
                     A[k + m/2] ← u − t
               ω ← ωω_m
   return A

procedure Bit − Reverse − Copy(a, A)
   n ← length(a)
   for k ← 0 to n − 1 do A[rev(k)] ← a_k
```

$rev(k)$ oznacza tutaj n-bitową liczbę powstałą przez zapisanie n-bitowego rozwinięcia binarnego liczby k od prawej do lewej strony.

Definicja 24 *(a)* Splotem *wektorów* $\mathbf{a} = \langle a_0, \ldots, a_{n-1} \rangle$ *i* $\mathbf{b} = \langle b_0, \ldots, b_{n-1} \rangle$ *nazywamy wektor* $\mathbf{c} = \langle c_0, \ldots, c_{2n-1} \rangle$ *taki, że* $\forall_{0 \leq i \leq 2n-1}\ c_i = \sum_{j=0}^{i} a_j b_{i-j}$ *i oznaczamy go* $\mathbf{c} = \mathbf{a} \otimes \mathbf{b}$.

(b) Negatywnym splotem zwiniętym *wektorów* $\mathbf{a}$ *i* $\mathbf{b}$ *nazywamy wektor* $\mathbf{d} = \langle d_0, \ldots, d_{n-1} \rangle$, *taki że* $d_i = \sum_{j=0}^{i} a_j b_{i-j} - \sum_{j=i+1}^{n-1} a_j b_{n+i-j}$.

Tak więc splot $\mathbf{a} \otimes \mathbf{b}$ jest reprezentacją `[Wsp]` iloczynu wielomianów o reprezentacjach `[Wsp]` **a** i **b** i, jak pokazaliśmy, może być obliczony przy użyciu transformacji Fouriera.

Fakt 28 *Niech* $\mathbf{a}$, $\mathbf{b}$ *i* $\mathbf{d}$ *jak w powyższej definicji. Niech* ψ *będzie pierwiastkiem z jedności stopnia* $2n$. *Oznaczmy przez* $\hat{\mathbf{a}}$, $\hat{\mathbf{b}}$ *i* $\hat{\mathbf{d}}$ *wektory* $\langle a_0, \psi a_1, \ldots, \psi^{n-1} a_{n-1} \rangle$, $\langle b_0, \psi b_1, \ldots, \psi^{n-1} b_{n-1} \rangle$ *i* $\langle d_0, \psi d_1, \ldots, \psi^{n-1} d_{n-1} \rangle$. *Wówczas* $DFT(\hat{\mathbf{d}}) = DFT(\hat{\mathbf{a}}) \cdot DFT(\hat{\mathbf{b}})$.

Aby uniknąć kłopotów związanych z niedokładną reprezentacją zespolonych pierwiastków z jedności, można transformację Fouriera wykonywać nad jakimś ciałem skończonym lub pierścieniem R_m liczb całkowitych modulo m posiadającym n-ty pierwotny pierwiastek z jedności.

Fakt 29 *Niech* n *i* ω *będą potęgami liczby* 2 *(różnymi od* 1*) oraz niech* $m = \omega^{n/2} + 1$. *Wówczas* n *i* ω *są odwracalne w* R_m *oraz* ω *jestn-tym pierwotnym pierwiastkiem z jedności.*

Notatki z AiSD. Nr ??. 16 marca 2005

Wyszukiwanie wzorców

IIUWr. II rok informatyki.

55 Notacja

- Σ - ustalony alfabet
- $T[1..n]$ i $P[1..m]$ - ciągi symboli z Σ
- T nazywamy *tekstem* a P - *wzorcem*
- Mówimy, że P *występuje z przesunięciem* s w tekście T jeśli $0 \leq s \leq n - m$ oraz $T[s + 1..s + m] = P[1..m]$.
- $w \sqsubset x$ - w jest *prefiksem* x-a (tzn. $\exists_{y \in \Sigma^*}\ wy = x$)
- $w \sqsupset x$ - w jest *sufiksem* x-a (tzn. $\exists_{y \in \Sigma^*}\ yw = x$)
- P_k - k-elementowy prefiks $P[1..k]$ wzorca $P[1..m]$
- T_k - k-elementowy prefiks $T[1..k]$ tekstu $T[1..m]$

Fakt 30 *Niech x, y i z będą takie, że $x \sqsupset z$ i $y \sqsupset z$. Wówczas*

- $|x| \leq |y| \Rightarrow x \sqsupset y$,
- $|x| \geq |y| \Rightarrow y \sqsupset x$,
- $|x| = |y| \Rightarrow x = y$,

56 Definicja problemu

Dane: wzorzec $P[1..m]$ oraz tekst $T[1..n]$
Zadanie: znaleźć wszystkie wystąpienia P w T (tj. znaleźć wszystkie s z przedziału $\langle 0, n - m\rangle$ takie, że $P \sqsupset T_{s+m}$).

57 Algorytmy

57.1 Algorytm naiwny

```
procedure Naive − string − matcher(T, P)
    n ← length(T)
    m ← length(P)
    for s ← 0 to n − m do
        if P[1..m] = T[s + 1..s + m]  then write("wzorzec występuje z przesunięciem", s)
```

Koszt: $\Theta((n - m + 1)m)$ w najgorszym przypadku.

57.2 Algorytm Karpa-Rabina

IDEA:
Słowa nad d-literowym alfabetem Σ traktujemy jako liczby d-arne. Jeśli p oznacza liczbę odpowiadajcacą wzorcowi P, a t_s - liczbę odpowiadającą $T[s+1..s+m+1]$ $(s = 0, ..n-m)$, to wzorzec występuje z przesunięciem s iff $p = t_s$. Gdy m jest duże, to p oraz t_i są duże i ich porównywanie jest kosztowne. Dlatego wybieramy liczbę q (zwykle jest to liczba pierwsza) taką, że dq mieści się w słowie maszynowym i liczby p oraz t_i obliczamy modulo q. Wówczas

(1) $p \neq t_s \Rightarrow P$ nie występuje w T z przesunięciem s,
(2) $p = t_s \Rightarrow P$ może występować w T z przesunięciem s.

```
procedure Karp − Rabin − matcher(T, P, d, q)
    n ← length(T)
    m ← length(P)
    h ← d^{m−1} mod q
    p ← 0; t_0 ← 0
    for i ← 1 to m do
        p ← (dp + P[i]) mod q
        t_0 ← (dt_0 + T[i]) mod q
    for s ← 0 to n − m do
        if p = t_s then
            if P[1..m] = T[s + 1..s + m]  then write("wzorzec występuje z przesunięciem", s)
        if s < n − m then t_{s+1} ← (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
```

KOSZT: $\Theta((n - m + 1)m)$ w najgorszym przypadku. Gdy wzorzec występuje w tekście niewiele razy oraz gdy t_i przyjmują wartości $\{0, .., q-1\}$ z równym prawdopodobieństwem, to wybierając q większe od m koszt powyższej procedury można oszacować przez $O(m+n)$.
UWAGA: Algorytm ten łatwo uogólnia się na problem szukania wzorców dwuwymiarowych.

57.3 Wyszukiwanie wzorców automatami skończonymi.

57.3.1 Konstrukcja automatu

IDEA:
Dla danego wzorca P skonstruujemy automat skończony M_P o stanach ze zbioru $\{0, ..m\}$. Automat, czytając tekst T, będzie znajdować się w stanie d, jeśli ostatnich d liter tekstu może rozpoczynać wzorzec i dla żadnego $e > d$, e ostatnio wczytanych liter nie może rozpoczynać wzorca. W szczególności dojście do stanu m będzie oznaczać, że m ostatnio wczytanych liter tekstu tworzy wzorzec.

Definicja 4 *Dla automatu skończonego $M = (Q, q_0, A, \Sigma, \delta)$, określamy funkcję $\phi : \Sigma^* \to Q$:*

$$\begin{aligned} \phi(\varepsilon) &= q_0 \\ \phi(wa) &= \delta(\phi(w), a), \end{aligned}$$

Innymi słowy $\phi(w)$ ="stan, w którym znajdzie się M po przeczytaniu w".

Definicja 5 *Dla wzorca P definiujemy funkcję $\sigma : \Sigma^* \to \{0, \ldots, m\}$:*

$$\sigma(x) \quad = \quad \max\{k \mid P_k \sqsupset x\}$$

Czyli $\sigma(x)$ ="długość najdłuższego prefiksu P, który jest sufiksem x-a".

Fakt 31 *(Własności funkcji σ)*

(a) $\sigma(x) = |P| \quad iff \quad P \sqsupset x$

(b) $x \sqsupset y \quad \Rightarrow \quad \sigma(x) \leq \sigma(y)$

Definicja 6 *(Automatu skończonego M_P dla wzorca P)*

- *zbiór stanów:* $Q = \{0, 1, \ldots, m\}$,
- *stan początkowy:* $q_0 = 0$,
- *zbiór stanów końcowych:* $A = \{m\}$,
- *funkcja przejścia:* $\forall_{q \in Q, a \in \Sigma} \ \delta(q, a) = \sigma(P_q a)$.

57.3.2 Program symulujący automat M_P.

```
procedure Finite − automaton − matcher(T, δ, m)
    n ← length(T)
    q ← 0
    for i ← 1 to n do
        q ← δ(q, T[i])
        if q = m then write( "wzorzec występuje z przesunięciem" , i − m)
```

Koszt procedury: $O(n)$ (koszt ten nie obejmuje kosztu obliczenia funkcji δ).

57.3.3 Analiza poprawności

Poniższe lematy i twierdzenie pokazują, że jeśli po wczytaniu i-tej litery tekstu M_P jest w stanie q $(=\phi(T_i))$, to q jest długością najdłuższego sufiksu T_i, który jest prefiksem P $(= \sigma(T_i))$. Ponieważ $\sigma(T_i) = m \quad$ iff $\quad P \sqsupset T_i$, więc stan akceptujący będzie osiągany wtedy i tylko wtedy, gdy m ostatnio przeczytanych znaków tworzy wzorzec.

- **Lemat 2** $\forall_{x \in \Sigma^*} \forall_{a \in \Sigma} \ \ \sigma(xa) \leq \sigma(x) + 1$,
- **Lemat 3** $\forall_{x \in \Sigma^*} \forall_{a \in \Sigma} \ \ q = \sigma(x) \ \ \Rightarrow \ \ \sigma(xa) = \sigma(P_q a)$
- **Twierdzenie 7** $\forall_{i=0,1,\ldots,n} \ \ \phi(T_i) = \sigma(T_i)$.

57.3.4 Obliczanie funkcji δ

- Sposób naiwny.

```
procedure Compute − Transition − Function(P, Σ)
    m ← length(P)
    for q ← 0 to m do
        for each a ∈ Σ do
            k ← min(m + 1, q + 2)
            repeat k ← k − 1 until P_k ⊐ P_q a
            δ(q, a) ← k
    return δ
```

  KOSZT: $O(m^3 \mid \Sigma \mid)$

- Sposób zdecydowanie mniej naiwny (będzie przedmiotem ćwiczeń).

  Wykorzystuje funkcję prefiksową, którą zdefiniujemy opisując algorytm Knutha-Morrisa-Pratta. Czas jego działania wynosi $O(m|\Sigma|)$.

58 Algorytm Knutha-Morrisa-Pratta.

58.1 Idea

Zasada podobna jak poprzednio: po przeczytaniu T_i chcemy wiedzieć jak długi prefiks P jest sufiksem T_i. Załóżmy, że długość tego prefiksu wynosi k. Jeśli $T[i+1] = P[k+1]$, to wiemy, że teraz ta długość wynosi $k+1$. Gorzej jeśli $T[i+1] \neq P[k+1]$. Funkcja δ pozwalała nam tę dlugość określić w jednym kroku. Pociągało to jednak za sobą konieczność wstępnego obliczenia wartości δ dla wszystkich par (k, a). To jest kosztowne! Teraz unikamy tego, pozwalając, by algorytm poświęcił więcej czasu na określenie długości prefiksu w trakcie czytania tekstu. Algorytm korzysta przy tym z pomocniczej funkcji π, którą oblicza wstępnie na podstawie wzorca w czasie $O(m)$.

Definicja 7 *Dla wzorca P definiujemy funkcję prefiksową $\pi : \{1, .., m\} \to \{0, .., m-1\}$*

$$\pi(q) \quad = \quad \max\{k \mid k < q \ i \ P_k \sqsupset P_q\}$$

KOMENTARZ: W sytuacji gdy k ostatnich znaków tekstu tworzy prefiks P, a kolejny znak tekstu jest niezgodny z $k+1$-szym znakiem P, algorytm może sprawdzać czy znak ten jest zgodny z krótszymi prefiksami P, będącymi jednocześnie sufiksami wczytanego tekstu. Jako kandydatów na te prefiksy algorytm próbuje te prefiksy wzorca, które są sufiksami P_k. O tym, które są to prefiksy mówi funkcja π.

58.2 Algorytm

```
procedure KMP − Matcher(T, P)
    n ← length(T);  m ← length(P)
    π ← Compute − Prefix − Function(P)
    q ← 0
    for i ← 1 to n do
        while q > 0 and P[q + 1] ≠ T[i] do q ← π(q)
        if P[q + 1] = T[i] then q ← q + 1
        if q = m then write( "wzorzec występuje z przesunięciem", i − m)
                      q ← π(q)
```

58.3 Obliczanie funkcji prefiksowej

```
procedure Compute − Prefix − Function(P)
    m ← length(P)
    π(1) ← 0;  k ← 0
    for q ← 2 to m do
        while k > 0 and P[k + 1] ≠ P[q] do k ← π(k)
        if P[k + 1] = P[q] then k ← k + 1
            π(q) ← k
    return π
```

KOSZT: Procedura $Compute - Prefix - Function$ działa w czasie $O(m)$, a procedura $KMP - Matcher$ w czasie $O(n + m)$.

59 Algorytm Boyera-Moore'a

IDEA:

Metoda podobna do metody naiwnej: sprawdzamy kolejne przesunięcia s, ale dla danego s tekst sprawdzamy począwszy od końca wzorca. Gdy napotkamy niezgodność korzystamy z dwóch heurystyk do zwiększenia s (stosujemy tę, która proponuje większe przesunięcie):

- heurystyka "zły znak",
- heurystyka "dobry sufiks".

59.1 Heurystyka "zły znak"

Jeśli niezgodność wystąpiła dla $P[j] \neq T[s + j]$ $(1 \leq j \leq m)$, to niech

$$k = \begin{cases} \max \{z \mid P[z] = T[s + j]\} & \text{jeśli takie } z \text{ istnieje,} \\ 0 & \text{w p.p.} \end{cases}$$

Jeśli $k = 0$ lub $k < j$, to ta heurystyka proponuje przesunąć s o $j - k$ znaków. Gdy $k > j$, to heurystyka nic nie proponuje.

59.2 Heurystyka "dobry sufiks"

Definicja 8 *Mówimy, że Q jest* podobne do R *(i piszemy $Q \sim R$) iff $Q \sqsupset R$ lub $R \sqsupset Q$.*

Heurystyka "dobry sufiks" mówi, że gdy napotkamy niezgodność $P[j] \neq T[s+j]$ $(1 \leq j \leq m)$, to s możemy zwiększyć o $m - \max\{k \mid 0 \leq k < m \;\&\; P[j+1..m] \sim P_k\}$.