wykład nr 1 (licencjat dzienny/wieczorowy) *11/12 października 2007 r.*

ALGORYTMY I STRUKTURY DANYCH

SORTOWANIE

Instytut Informatyki Uniwersytetu Wrocawskiego *Paweł Rzechonek*

1 Podstawowe pojęcia

Zadania porządkowe (ang. *ordering problems*) to klasa problemów, w których zbiór danych należy w pewien sposób uporządkować lub coś w tym zbiorze wyszukać. Do problemów porządkowych zalicza się przede wszystkim sortowanie, znajdowanie elementu minimalnego badź maksymalnego i wybór mediany.

Elementy zbioru danych pochodzą z określonego uniwersum z porządkiem liniowym i relacją $\leq$, aby każdą parę elementów można było ze sobą porównać. Zdarza się, że pojedynczy element jest całym rekodrm różnych informacji. Wtedy wyróżnia się jedno pole zwane ***kluczem*** (ang. *key*), na którym jest określona relacja $\leq$ i które jest reprezentantem rekordu w operacjach porównywania.

Elementy w zbiorze danych są ponumerowane kolejnymi liczbami naturalnymi. Liczby te są ***pozycjami*** (ang. *position*) elementów w zbiorze. Jest więc dość naturalne przechowywanie takich danych w tablicy; ale rozważa się też przypadki, gdy dane są zapisane w pliku badź są pamiętane w liście.

W problemach porządkowych nie zakłada się uporządkowania zbioru danych — przeważnie są to zbiory nieuporządkowane, albo częściowo uporządkowane. Spójrzmy na n–elementowy zbiór danych jako na ciąg elementów $A = (a_0, \ldots, a_{n-1})$. Miarą nieuporządkowania w takim ciągu jest liczba występujących w nim ***inwersji*** (ang. *inversion*), czyli takich par (i, j), że $0 \leq i < j < n$ oraz $a_i > a_j$:

$$|\{(i,j) \mid 0 \leq i < j < n \wedge a_i > a_j\}|$$

O ciągu powiemy, że jest ***uporządkowany*** (ang. *ordered*), gdy jest niemalejący (nie ma w nim żadnej inwersji):

$$\forall\, 0 < i < n \,:\, a_{i-1} \leq a_i$$

Spostrzeżenie 1 *Maksymalna liczba inwersji w ciągu n–elementowym wynosi $\frac{n(n-1)}{2}$.*

Wskazówka do dowodu
Maksymalna liczba inwersji występuje w ciągu odwrotnie uporządkowanym. Dowód nie wprost. ■

Mówimy, że algorytm porządkujący działa ***w miejscu*** (ang. *in–place*), jeśli podczas działania algorytm zużywa tylko stałą liczbę dodatkowych komórek pamięci. W skład dodatkowych komórek pamięci nie wliczamy danych.

Mówimy, że algorytm porządkujący jest ***stabilny*** (ang. *stable*), jeśli po zakończeniu działania algorytm zachowuje względne ustawienia elementów o równych kluczach. Algorytm taki może zmieniać pozycje elementów w zbiorze z danymi, ale elementy o takich samych wartościach zachowują początkowe ustawienia względem siebie.

O algorytmach porządkujących będziemy zakładali, że możemy mieć ***bezpośredni dostęp*** (ang. *random access*) do każdego elementu, że elementy mogą być jedynie ***porównywane*** (ang. *compare*), ***przenoszone*** (ang. *move*) bądź ***zamieniane*** (ang. *exchange*) miejscami. Zauważmy, że jedna zamiana miejscami dwóch elementów wymaga trzech przesunięć.

2 Problem sortowania

Sortowanie (ang. *sorting*) to problem bardzo często rozwiązywany na komputerze. Wiąże się to z faktem, że dużo szybciej można znajdować informacje w zbiorach uporządkowanych niż w nieuporządkowanych.

Problem sortowania tablic można zdefiniować następująco.

Dane: W n–elementowej tablicy $A[0 \ldots n-1]$ zapisany jest nieuporządkowany ciąg wartości $a_0, a_1, \ldots, a_{n-1}$, pochodzących z określonego uniwersum z porządkiem liniowym i relacją porządkująca $\leq$.

Zadanie: Dane w tej tablicy należy tak poprzestawiać, aby $a_{\sigma_0} \leq a_{\sigma_1} \leq \ldots \leq a_{\sigma_{n-1}}$, gdzie σ jest permutacją zbioru n–elementowego.

Ograniczenia: Porządkując dane możemy jedynie porównywać elementy i przepisywać je bądź zamieniać miejscami.

2.1 Wyszukiwanie

Najczęściej wykonywaną operacją na zbiorach z danymi jest ***wyszukiwanie*** (ang. *searching*) informacji względem zadanego klucza. Problem ten można zdefiniować następująco.

Dane: Zadany jest klucz x oraz n–elementowa tablica $A[0 \ldots n-1]$ z elementami $a_0, a_1, \ldots, a_{n-1}$.

Zadanie: Należy znaleźć pozycję w tablicy, na której znajduje się element o wartości x.

Ograniczenia: Można jedynie porównywać klucz z elementami w tablicy.

Najprostszym rozwiązaniem tego zadania jest przeglądnięcie tablicy od początku do końca i porównywanie jej elementów z kluczem, czyli ***wyszukiwanie sekwencyjne*** (ang. *sequence searching*).

---

function SEQ–SEARCH (**keytype** $A[0 \ldots n-1]$, **key** x) $\mapsto$ **int**
{
    for $i = 0 \ldots n-1$ **do**
        if $A[i].key = x$ **then return** i;
    return null;
}

Działanie procedury SEQ–SEARCH wymaga wykonania 1 porównania w najlepszym przypadku, n porównań w najgorszym przypadku, oraz $\sim \frac{n}{2}$ porównań w przypadku średnim gdy szukany element znajduje się w tablicy.

Dużo szybszy algorytm można skonstruować, gdy dane w tablicy pochodzą z pewnego uniwersum z porządkiem liniowym i są uporządkowane. Oto zmienione założenie dotyczące postaci danych.

Dane: Zadany jest klucz x oraz n-elementowa tablica $A[0 \ldots n-1]$ z elementami $a_0, a_1, \ldots, a_{n-1}$ uporządkowanymi w kolejności niemalejącej $a_0 \leq a_1 \leq \ldots \leq a_{n-1}$.

Korzystając z informacji o uporządkowaniu danych można zastosować algorytm ***wyszukiwania binarnego*** (ang. *binary searching*).

---

function BIN–SEARCH (**ordtype** $A[0 \ldots n-1]$, **ord** x) $\mapsto$ **int**
{
    int $a \leftarrow 0$; *// początek obszaru poszukiwań*
    int $b \leftarrow n$; *// koniec obszaru poszukiwań*
    int $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$; *// środek obszaru poszukiwań*
    while $a < b$ **do**
    {
        compare $A[m].key ? x$
            case $<$ **then** $a \leftarrow m+1$;
            case $>$ **then** $b \leftarrow m$;
            case $=$ **then return** m;
        $m \leftarrow \lfloor \frac{a+b}{2} \rfloor$;
    }
    return null;
}

Procedura BIN–SEARCH wykona nie więcej niż $\lfloor 1 + \log n \rfloor$ porównań w najgorszym przypadku.

3 Elementarne metody sortowania

3.1 Sortowanie bąbelkowe

Algorytm ***sortowania bąbelkowego*** (ang. *bubble–sort*) jest najprostszym i najpopularniejszym algorytmem sortowania. Idea sortowania bąbelkowego polega na przesuwaniu coraz to większych elementów na koniec tablicy. Po dojściu do elementu maksymalnego podczas przeglądania tablicy, będzie on dalej przepychany o jedną pozycję aż do końca. W każdej fazie rozmiar problemu redukuje się więc o 1, bo element maksymalny znajdzie się na właściwej, ostatniej pozycji w tablicy.

---

procedure BUBBLE–SORT (**ordtype** $A[0 \ldots n-1]$)
{
  for $q = n-1 \ldots 1$ **do**
    for $i = 1 \ldots q$ **do**
      if $A_{i-1} > A_i$ **then** $A_{i-1} \longleftrightarrow A_i$;
}

Sortowanie bąbelkowe jest stabilne i działa w miejscu. W algorytmie tym wykonujemy $\frac{n(n-1)}{2}$ porównań w każdym przypadku i co najwyżej $\frac{n(n-1)}{2}$ zamian elementów. Jako operację dominującą przyjmujemy porównania. Czas działania algorytmu wynosi więc w każdym przypadku $\frac{n(n-1)}{2} \in \Theta(n^2)$.

3.2 Sortowanie przez zamianę

Idea algorytmu ***sortowania przez zamianę*** (ang. *exchange–sort*) polega na przenoszeniu coraz to mniejszych elementów na początek tablicy. Po dojściu do elementu minimalnego podczas przeglądania tablicy, zostanie on umieszczony na pierwszej pozycji, a w każdym następnym kroku będzie on tylko porównywany z następnymi elementami. W każdej fazie rozmiar problemu redukuje się więc o 1, bo element minimalny znajdzie się na właściwej, pierwszej pozycji w tablicy.

---

procedure EXCHANGE–SORT (**ordtype** $A[0 \ldots n-1]$)
{
  for $p = 0 \ldots n-2$ **do**
    for $i = p+1 \ldots n-1$ **do**
      if $A_p > A_i$ **then** $A_p \longleftrightarrow A_i$;
}

Sortowanie przez zamianę nie jest stabilne ale działa w miejscu. W algorytmie tym wykonujemy $\frac{n(n-1)}{2}$ porównań w każdym przypadku i co najwyżej $\frac{n(n-1)}{2}$ zamian elementów. Jako operację dominującą przyjmujemy porównania. Czas działania algorytmu wynosi więc w każdym przypadku $\frac{n(n-1)}{2} \in \Theta(n^2)$.

3.3 Sortowanie przez wstawianie

Idea algorytmu ***sortowania przez wstawianie*** (ang. *insertion–sort*) polega na wstawianiu kolejnych elementów do uporządkowanego początkowego fragmentu tablicy. W ten sposób w każdej fazie zwiększamy długość posortowanej części tablicy o 1.

---

procedure INSERTION–SORT (**ordtype** $A[0 \ldots n-1]$)
{
  for $q = 1 \ldots n-1$ **do**
    for $i = q \ldots 1$ **do**
      if $A_{i-1} > A_i$
        then $A_{i-1} \longleftrightarrow A_i$;
        else break;
}

Sortowanie przez wstawianie jest stabilne i działa w miejscu. W algorytmie tym wykonujemy $\frac{n(n-1)}{2}$ porównań w najgorszym przypadku oraz $\sim \frac{n^2}{4}$ w przypadku średnim. Jako operację dominującą przyjmujemy porównania. Czas działania algorytmu jest więc rzędu $\Theta(n^2)$ w przypadku pesymistycznym i średnim.

3.4 Sortowanie przez wybieranie

Algorytm ***sortowania przez wybieranie*** (ang. *selection–sort*) jest udoskonaloną wersją sortowania przez zamianę. Idea sortowania przez wybieranie polega na wyznaczeniu pozycji elementu minimalnego a następnie umieszczeniu go na początku tablicy za pomocą jednej zamiany. W każdej fazie rozmiar problemu redukuje się więc o 1, bo element minimalny znajdzie się na właściwej, pierwszej pozycji w tablicy.

---

procedure SELECTION–SORT (**ordtype** $A[0 \ldots n-1]$)
{
  for $p = 0 \ldots n-2$ **do**
  {
    $m \leftarrow p$; // *pozycja elementu minimalnego*
    for $i = p+1 \ldots n-1$ **do**
      if $A_m > A_i$ **then** $m \leftarrow i$;
    if $p < m$ **then** $A_p \longleftrightarrow A_m$;
  }
}

Sortowanie przez wybieranie nie jest stabilne ale działa w miejscu. W algorytmie sortowania przez wybieranie wykonujemy $\frac{n(n-1)}{2}$ porównań w każdym przypadku, ale tylko co najwyżej $n-1$ zamian elementów. Jako operację dominującą przyjmujemy porównania. Czas działania algorytmu wynosi więc w każdym przypadku $\frac{n(n-1)}{2} \in \Theta(n^2)$. Algorytm ten można wykorzystać do sortowania małych danych, w sytuacji gdy zamiana elementów miejscami jest operacją trudną i kosztowną.

4 Sortowanie Shella

Sortowanie Shella (D. L. Shell, 1959) jest prostym rozszerzeniem sortowania przez wstawianie, które przyspiesza proces sortowania dzięki przestawianiu odległych a nie tylko sąsiadujących ze sobą elementów.

Definicja 2 *d–pociąg ciągu $A = (a_0, a_1, a_2, \ldots)$ to podciąg złożony z elementów oddalonych od siebie o wielokrotność liczby d.*

Obserwacja 3 *Dla ustalonej wartości naturalnej d, ciąg n–elementowy zawiera co najwyżej d różnych d–podciągów: gdy $d > n$ to ciąg zawiera tylko n podciągów jednoelementowych, w przeciwnym przypadku jest ich dokładnie d.*

Definicja 4 *Ciąg $A = (a_0, a_1, a_2, \ldots)$ jest d-posortowany, gdy wszystkie jego d–podciągi są posortowane.*

Twierdzenie 5 *Przeprowadzenie p–sortowania w ciągu q-posortowanym nie zniszczy p–posortowania.*

---

procedure SHELL–SORT (**ordtype** $A[0 \ldots n-1]$)
{
  $D \leftarrow (d_0, d_1, d_2, \ldots)$; // *ciąg przyrostów*
  for $k = \max\{i \mid d_i < n\} \ldots 0$ **do**
  {
    $d \leftarrow d_k$;
    for $m = 0 \ldots d-1$ **do**
      $\langle$*posortuj metodą* INSERTION–SORT *d-podciąg rozpoczynający się od* $A[m]\rangle$;
    $k--$;
  }
}

Twierdzenie to jest podstawą sortowania Shella. Otwartym problemem pozostaje odpowiedni dobór przyrostów, czyli ciągu wartości $D = (d_0, d_1, d_2, \ldots)$, według którego należy przeprowadzać kolejne sortowania, przy czym $d_0 = 1$ jest ostatnim przyrostem w sortowaniu.

Twierdzenie 6 *Sortowanie n–elementowego ciągu metodą Shella, w którym zastosuje się dwuelementowy ciąg przyrostów $D = (1, \lfloor \sqrt[3]{n} \rfloor)$ będzie działać w czasie $O(n^{5/3})$.*

Stosując sortowanie Shella z tylko dwoma skokami można istotnie przyspieszyć proces sortowania w stosunku do sortowania przez proste wstawianie.

Jeszcze lepsze wyniki można uzyskać wprowadzając większą ilość skoków. Bardzo dobrze zachowuje się sekwencja skoków 1, 3, 7, 15, 31... (T. N. Hibbard, 1963), którą można zapisać następującym wzorem:

$$h_i \quad = \quad 2_{i+1} - 1 \quad \text{dla} \quad i \geq 0$$

Twierdzenie 7 *Sortowanie n–elementowego ciągu metodą Shella, w którym zasosuje się ciąg przyrostów Hibbarda (1, 3, 7, 15, 31...) będzie działać w czasie $O(n^{3/2})$.*

Inny dobry ciąg to sekwencja skoków 1, 4, 13, 40, 121... (D. E. Knuth, 1969), którą można zdefiniować rekurencyjnie:

$$\begin{array}{lcl} h_0 & = & 1 \\ h_i & = & 3\,h_{i-1} + 1 \quad \text{dla} \quad i \geq 1 \end{array}$$

Twierdzenie 8 *Sortowanie n–elementowego ciągu metodą Shella, w którym zasosuje się ciąg przyrostów Knutha (1, 4, 13, 40, 121...) będzie działać w czasie $O(n^{3/2})$.*

Działanie algorytmu Shella można poprawić stosując sekwencję skoków 1, 5, 19, 41, 109... (R. Sedgewick, 1986), która wyraża się wzorem:

$$h_i \; = \; \begin{cases} 9 \cdot 2^i - 9 \cdot 2^{i/2} + 1 & \text{dla parzystych } i \\ 8 \cdot 2^i - 6 \cdot 2^{(i+1)/2} + 1 & \text{dla nieparzystych } i \end{cases}$$

Twierdzenie 9 *Sortowanie n–elementowego ciągu metodą Shella, w którym zasosuje się ciąg przyrostów Sedgewicka (1, 5, 19, 41, 109...) będzie działać w czasie $O(n^{4/3})$.*

Dużo lepszy wynik można uzyskać, gdy wykorzysta się fakt, że sortowanie przez wstawianie zastosowane do ciągu 2-uporządkowanego i 3-uporządkowanego będzie działało w liniowym czasie.

Fakt 10 *Jeśli zastosujemy metodę sortowania przez proste wstawianie do ciągu, który jest jednocześnie 2-posortowany i 3-posortowany, to każdy element przesunie się co najwyżej o jedną pozycję.*

Jeśli ciąg jest 4- i 6-uporządkowany, to 2-sortowanie wykona się na tym ciągu w liniowym czasie; a jeśli ciąg jest 6- i 9-uporządkowany, to również w liniowym czasie wykona się na tym ciągu 3-sortowanie. Kontynuując to rozumowanie można skonstruować następującą sekwencję przyrostów (V. R. Pratt, 1971): 1, 2, 3, 4, 6, 9, 8, 12, 18, 27...

$$\begin{array}{ccccccccccc} & & & & 1 & & & & & & \\ & & & 2 & & & 3 & & & & \\ & & 4 & & & 6 & & & 9 & & \\ & 8 & & & 12 & & & 18 & & & 27 \\ 16 & & & 24 & & & 36 & & & 54 & & & 81 \end{array}$$

Wartości ciągu przyrostów Pratta można odczytać z powyższego trójkąta, a ogólna postać każdego wyrazu to:

$$h_{(i^2+i)/2+j} \; = \; 2^{i-j} \cdot 3^j \quad \text{dla } i = 0, 1 \ldots \text{ oraz } j = 0 \ldots i$$

Twierdzenie 11 *Sortowanie n-elementowego ciągu metodą Shella, w którym zasosuje się ciąg przyrostów Pratta (1, 2, 3, 4, 6, 9, 8, 12, 18, 27...) będzie działać w czasie $O(n \log^2 n)$.*