Ćwiczenia z Metod Numerycznych II A
rok akademicki 2001/2002

seria 9

Jakub Narębski Tomasz Pawłowski

14 stycznia 2005 roku

1 Sortowanie

Sortowanie definiuje się następująco: dana jest lista (tablica) $q = [a_1, a_2, \ldots, a_n]$ elementów zbioru liniowo uporządkowanego; trzeba ustawić elementy w porządku niemalejącym $a_{i_1} \leqslant a_{i_2} \leqslant \cdots \leqslant a_{i_n}$. Czasami wymaga się dodatkowego warunku *stabilności*: elementy równe sobie na wyjściu znajdują się w tej samej kolejności, w jakiej znajdowały się na wejściu (np. sortowanie według imienia i nazwiska).

Często z sortowanymi elementami związane są dodatkowe dane. Dla uproszczenia w dalszych rozdziałach zakładać będziemy, że sortujemy liczby całkowite. Czas działania algorytmu będziemy oceniać na podstawie liczby *porównań*.

1.1 Sortowanie przez wstawianie

Sortowanie przez wstawianie, to algorytm stosowany przez większość graczy do układania kart. Trzymają w ręku uporządkowane do tej pory karty i gdy pojawia się następna, umieszczają ją na właściwym w danej chwili miejscu. Inaczej, dla każdego $i = 2, 3, \ldots, n$ trzeba wstawić $a[i]$ w już uporządkowaną część tablicy $a[1] \leqslant \cdots \leqslant a[i-1]$.

```
for (i = 1; i < N; i++) {
  /* a[0] ⩽ a[1] ⩽ ··· ⩽ a[i-1] */
  j = i;
  v = a[i];

  /* wstaw a[i] w odpowiednie miejsce w a[0...i] */
  while (j > 0 && a[j-1] > v) {
    a[j] = a[j-1]; /* a[j] - wolne miejsce */
    j--;
  }
  a[j] = v;
}
```

Algorytm ten działa w czasie pesymistycznym $\frac{1}{2}N^2 - \mathcal{O}(N)$, oczekiwanym $\frac{1}{4}N^2 + \mathcal{O}(N)$. Ilość wykonanych porównań jest równa liczbie *inwersji*, tzn. takich par że $a[i] > a[j]$ dla $i < j$, zatem działa bardzo szybko dla tablic niemal uporządkowanych. Algorytm ten jest stabilny, prosty i łatwo implementowalny.

1.2 Sortowanie przez selekcję (wybór)

Sortowanie przez selekcję polega na układaniu elementów od najmniejszego w ciągu. Dla każdego $i = 1, 2, \ldots, n$ trzeba wśród elementów $a[i], a[i+1], \ldots, a[n]$ znaleźć najmniejszy i wstawić go w miejsce elementu $a[i]$, czyli na koniec posortowanego kawałka tablicy.

```
for (i = 0; i < N-1; i++) {
  /* a[0] ⩽ a[1] ⩽ ··· ⩽ a[i-1] ⩽ a[i...n] */
  imin = i;
  for (j = i+1; j < N; j++) {
    /* znajdź najmniejszy element w a[i...n] */
    if (a[j] < a[imin])
      imin = j;
  }

  /* zamień miejscami a[imin] i a[i] */
  v = a[imin]; a[imin] = a[i]; a[i] = v;
}
```

Algorytm ten działa w czasie stałym $\frac{1}{2}N^2 - \mathcal{O}(N)$; zawsze wykonujemy ten sam ciąg porównań. Algorytm ten dokonuje tylko $N - 1$ przestawień. Algorytm ten nie jest stabilny.

1.3 Sortowanie szybkie (quicksort)

Algorytm ten jest jednym z najczęściej stosowanych algorytmów sortowania. Jest on uważany za najszybszy algorytm sortowania dla „losowych” danych wejściowych, bliski teoretycznej dolnej granicy szybkości algorytmów sortowania opartych o porównania elementów.

Jego konstrukcja oparta jest o zasadę „dziel–i–rządź”: aby posortować tablicę, dzielimy ją na dwie części (tak by wszystkie elementy lewej części były mniejsze od wszystkich elementów prawej części) i sortujemy każdą z tych części niezależnie.

W bibliotece standardowej C znajduje się funkcja biblioteczna qsort której możemy użyć do sortowania. Jako jeden z argumentów wymaga ona funkcji porównującej elementy. Funkcja ta jako argumenty przyjmuje *adresy* obiektów do porównania, a zwraca wartość *całkowitą* większą od zera jeśli pierwszy obiekt jest większy od drugiego, zero jeśli obiekty są równe, mniejszą od zera jeśli pierwszy obiekt jest mniejszy od drugiego. $a > b \implies \mathrm{cmp}(a, b) > 0$. Poniżej znajduje się przykład jej wykorzystania

```
#include <stdlib.h>
...
int compare_double(const void *a, const void *b)
{
  /* rzutujemy na wskaźnik do właściwego typu typ */
  const double *ia = (const double *) a;
  const double *ib = (const double *) b;

  /* zwracamy >0 gdy *ia > *ib, 0 gdy *ia == *ib, <0 wpp */
  return (*ia > *ib) - (*ia < *ib);
}
...
```

```c
int a[N];
...
qsort(a, N, sizeof(a[0]), compare_double);
```

Sortowanie to ma koszt pesymistyczny $\mathcal{O}(N^2)$, oczekiwany $\mathcal{O}(N \log N)$ (z małym współczynnikiem proporcjonalności, rzędu 1.4), z silnym skupieniem wokół wartości oczekiwanej. Można go ulepszyć by był częściej $\mathcal{O}(N \log N)$ (m. in. dla prawie posortowanej lu o większości elementów równych sobie). Liczba zamian jest podobna jak liczba porównań. Algorytm ten zazwyczaj wymaga dodatkowo $\mathcal{O}(\log N)$ ($\mathcal{O}(N^2)$ w częsci implementacji) dodatkowej pamięci, nie jest też stabilny (ale można go uczynić stabilnym za pomocą porównywania adresów; patrz `info libc`).

1.4 Sortowanie stogowe (kopcowe)

Sortowanie stogowe (heapsort) korzysta z właściwości stogu, struktury danych która umożliwia dodanie nowego elementu oraz pobranie i usunięcie największego elementu w czasie $\mathcal{O}(\log N)$. Konstruujemy kopiec z elementów tablicy, a następnie N–krotnie pobieramy i usuwamy największy element ze stogu, uzyskując uporządkowaną tablicę (podobnie jak w sortowaniu przez selekcję (wybieranie)). Kopiec można umieścić w sortowanej tablicy, uzyskując sortowanie za pomocą tylko stałej dodatkowej pamięci.

1.4.1 Definicja i właściwości stogu

Przez *stóg* (kopiec) rozumiemy drzewo binarne (każdy węzeł ma co najwyżej dwu potomków) o dwu własnościach. Pierwszą z nich to *porządek stogowy*: wartość każdego węzła jest nie mniejsza niż wartości jego potomków. Drugą własnością stogu jest jego *kształt*: wszystkie poziomy są wypełnione całkowicie, z wyjątkiem co najwyżej ostatniego — spójnie wypełnionego od strony lewej.

Dzięki właściwości kształtu można reprezentować kopiec w tablicy (kolejnymi poziomami, począwszy od wierzchołka) bez dodatkowych dowiązań. W tej reprezentacji potomkami węzła o indeksie k są węzły $2k$ i $2k+1$, zaś przodkiem węzła l jest węzeł $\lfloor l/2 \rfloor$. Warunek uporządkowania przyjmuje postać:

$$\bigwedge_{2 \leqslant i \leqslant N} a[\lfloor i/2 \rfloor] \geqslant a[i].$$

1.4.2 Operacje na stogu

Dwie podstawowe operacje na stogu to operacja przywracania uporządkowania dla stogu do którego wstawiono nowy element do korzenia (operacja downheap), oraz stogu do którego wstawiono nowy element jako jego ostatni element (operacja upheap). Poniżej znajdują się implementacje tych dwu procedur ($1 \leqslant k \leqslant N$).

```c
void upheap(int heap[], int k)
{
  int v; /* wartość którą wstawiamy */
  int i; /* indeks ojca węzła k */

  v = heap[k-1];
```

```
    i = k/2;
    while (k > 1 && heap[i-1] < v) {
        heap[k-1] = heap[i-1];
        k = l; i = i/2;
    }
    heap[k-1] = v;
}

void downheap(int heap[], int k, int N)
{
    int v; /* wartość którą wstawiamy */
    int j; /* indeks większego z synów węzła k */

    v = heap[k-1];
    while (k <= N/2) {
        j = 2*k;
        if (j < N && heap[j-1] < heap[j+1-1])
            j++;
        if (v >= heap[j-1])
            break;
        heap[k-1] = heap[j-1];
        k = j;
    }
    heap[k-1] = v;
}
```

Obydwie procedury wykonują się w czasie $\mathcal{O}(\log N)$.

Za ich pomocą łatwo jest zaimplementować funkcje insert wstawiającą element do kopca w czasie $\mathcal{O}(\log N)$, deletemax usuwająca (i zwracająca) największy element ze stogu w czasie $\mathcal{O}(\log N)$ oraz construct budującą stóg z tablicy w czasie $\mathcal{O}(N)$.

```
void insert(int heap[], int *N, int value)
{
    heap[(*N)-1] = value;
    upheap(heap, *N);
    (*N)++;
}

int deletemax(int heap[], int *N)
{
    int max = heap[0];

    a[0] = a[(*N)-1];
    (*N)--;
    downheap(heap, 1, *N);
}

void construct(int heap[], int N)
{
    int i;

    for (i = N/2; i >= 1; i--)
        downheap(heap, i, N);
}
```

1.4.3 Implementacja sortowania stogowego (heapsort)

W sortowaniu stogowym najpierw budujemy stóg z tablicy, korzystając N–krotnie z funkcji insert lub z funkcji construct. Następnie budujemy uporządkowana tablicę od końca, zdejmując kolejne największe elementy ze stogu przy pomocy funkcji deletemax.

```
void heapsort(int a[], int N)
{
  int M;

  construct(a, N);
  do {
    M = N;
    a[M] = deletemax(a, &N);
  } while (N > 1);
}
```

Bez utraty prostoty można by napisać sortowanie stogowe używając wyłącznie funkcji downheap:

```
void heapsort(int a[], int N)
{
  int v;
  int i;

  for (i = N/2; i >= 1; i--)
    downheap(a, i, N);
  for (i = N; i >= 2; i--) {
    v = a[i-1]; a[i-1] = a[0]; a[0] = v;
    downheap(a, 1, i-1);
  }
}
```

Sortowanie stogowe ma koszt pesymistyczny $2N \log N + \mathcal{O}(N)$. Wyniki badań pokazują, że oczekiwany współczynnik przy $N \log N$ jest bliski 2. Sortowanie to działa „w miejscu", t.j. wymaga tylko stałej ilości dodatkowej pamięci (w przeciwieństwie do sortowania szybkiego). Sortowanie stogowe nie jest stabilne.

1.5 Sortowanie przez scalanie (łączenie)

Sortowanie przez scalanie to algorytm sortowania używany najczęściej do sortowania plików. Algorytm ten działa w czasie pesymistycznym $\mathcal{O}(N \log N)$, i jest on stabilny (w klasycznej implementacji). Algorytm ten wymaga dodatkowej pamięci, tablicy (lub pliku) pomocniczego, $\mathcal{O}(N)$ — można tego uniknąć ale kosztem dużej komplikacji (i spowolnienia) algorytmu. Co czyni ten algorytm szczególnie przydatnym do sortowania plików jest fakt, że jest on lokalny, i nie trzeba trzymać w pamięci całego pliku (potrzebny jest tylko sekwencyjny dostęp do danych).

1.5.1 Zapis w postacie rekurencyjnej

Najprościej zrozumieć sortowanie przez scalanie w wersji rekurencyjnej. Dzielimy tablicę (plik) na połowy, sortujemy te połówki rekurencyjnie (tablica/plik długości 1 jest już posortowana), a następnie łączymy (scalamy) posortowane kawałki — stąd nazwa algorytmu.

```
template <typename T>
void mergeSort(T a[], int left, int right, T tmp[])
{
  if (left >= right)
    return;

  int middle = (left+right)/2;

  mergeSort(a,      left, middle, tmp);
  mergeSort(a, middle+1,  right, tmp);
  merge(a, left, middle, right, tmp);
}
```

Scalanie dwu posortowanych kawałków polega na tym, że do tablicy (pliku) wynikowego przenosimy zawsze mniejszy z początkowych elementów obu części; jeśli jedna z części zostanie w całości przeniesiona do tablicy (pliku) wynikowego to przenosimy tam pozostały kawałek drugiej części.

1.5.2 Łączenie proste

Algorytm sortowania przez scalanie (łączenie) można także zapisać bez pomocy rekurencji. Odpowiednikiem wyżej przedstawione wersji rekurencyjnej jest sortowanie przez tzw. *łączenie proste*:

(a) Dzielimy ciąg (tablicę, plik) a na dwie połowy nazwane b i c.
(b) Łączymy ciągi b i c składając pojedyncze ich elementy w pary uporządkowane.
(c) Nazywamy połącząony ciąg a i powtarzamy kroki 1 i 2, łącząc tym razem pary uporządkowane w czwórki uporządkowane.
(d) Powtarzamy poprzednie kroki łąćząc czwórki w ósemki itd., podwajając za każdym razem długość łączonych podciągw dopóty, dopóki cały ciąg nie zostanie uporządkowany.

1.5.3 Łączenie naturalne

W metodzie łączenia prostego nie korzysta się z ewentualnego, częsciowego uporządkowania danych źródłowych. Sortowanie w którym zawsze scala się dwa możliwie najdłuższe podciągi nazywamy *sortowaniem przez łączenie naturalne*.

Metoda ta daje znaczące przyspieszenie dla prawie posortowanych ciągów, kosztem dodatkowych operacji wymaganych do sprawdzania czy podciag posortowany już się skończył. Dodatkowo może obecnie zajść sytuacja, gdzie dwie części nie będa równej długości, i/lub nie będą zawierały identycznej ilości podciągów.

Przygotował: Jakub Narębski