ASD—2

1 QuickSort z "małym" stosem

Rozważmy raz jeszcze ogólny schemat algorytmu QuickSort.

```
void QuickSort(long *a, int l, int r)
/* sortujemy a[l..r]; l<r */
{
  int j;

  j = partition(l,r);
  if (j-1 > l)
    QuickSort(l,j-1);
  if (r > j+1)
    QuickSort(j+1,r);
}
```

Zauważmy, że:

1. oba wywołania rekurencyjne w treści funkcji QuickSort są niezależne, tzn. mogą być wykonywane w dowolnej kolejności;
2. wywołania te znajdują się na końcu funkcji, więc
   (a) jedno z nich może być zastąpione iteracją;
   (b) parametry drugiego wywołania można zapamiętać na stosie.

Na stosie znajdować się będą pary indeksów `[c,d]` określające parametry wywołań QuickSort, które należy jeszcze wykonać. Przy realizacji wywołania `QuickSort(l,r)` z dwóch wywołań rekurencyjnych, na stos będziemy wkładać końce dłuższego z podciągów podziału, powiedzmy $[c, d]$, końce krótszego pozostawiając dla iteracji. Zauważmy, że następny podciąg wkładany na stos po `a[c..d]` (a właściwie jego końce) ma rozmiar co najmniej dwukrotnie mniejszy niż `a[l..r]` (zakładamy, że parę `[c,d]` zdjeliśmy uprzednio z wierzchołka stosu). Przez indukcję pokazujemy, że k-ty podciąg na stosie ma rozmiar $\leq \frac{n}{2}$. Zatem na stosie może się znaleźć jednocześnie tylko co najwyżej $\log n$ par indeksów wyznaczających podciągi $a[0..n-1]$, które należy jeszcze posortować. Zatem $S(n) = O(\log n)$.

Oto zmieniona wersja algorytmu QUICKSORT. Stos reprezentujemy w tablicy par indeksów `St[MAX_STOS]`

```
void QuickSortMalyStos(long *a, int n)
{
   #define MAX_STOS 100

   /* stos */
   struct para { int c, d; } St[MAX_STOS];
   int ile_na_stosie = 0;

   int i,j,l,r;
   long v, x;

   if (n > 1){

     /* inicjalizacja --- QuickSort(a,0,n-1) */
     St[0].c = 0; St[0].d = n-1;
```

```
ile_na_stosie = 1;

do{
  /* [l,r] = Pop(St); */
  ile_na_stosie--;
  l = St[ile_na_stosie].c;
  r = St[ile_na_stosie].d;

  while (l < r){
  /* sortujemy a[l..r] */

    /* podzial */
    v = a[l];
    i = l;
    j = r+1;

    do{
      do
        i++;
      while ( (i <= r) && ( a[i] < v) );
      do
        j--;
      while ( a[j] > v );
      if (i < j){
        x = a[i]; a[i] = a[j]; a[j] = x;
      }
    }while (i < j);
    a[l] = a[j]; a[j] = v;
    /* koniec podzialu */
```

```
        if ( (j - l) < (r - j) ){
          /* Push(St,[j+1,r] */
          St[ile_na_stosie].c = j+1;
          St[ile_na_stosie].d = r;
          ile_na_stosie++;
          r = j-1;
        }
        else{
          /* Push(St,[l,j-1] */
          St[ile_na_stosie].c = l;
          St[ile_na_stosie].d = j-1;
          ile_na_stosie++;
          l = j+1;
        }
      }
    }while (ile_na_stosie > 0);
  }
}
```

Algorytm QuickSort był obiektem intensywnych badań w wyniku czego zaproponowano szereg jego modyfikacji:

1. Algorytm InsertionSort może być używany do sortowania danych o małych rozmiarach, np. ≤ 20.

2. Gdy nie ma pewności, że wejściowy ciąg jest losowy, to lepiej, albo używać generatora liczb losowych do określenia elementu dzielącego
```
/* generowanie losowego indeksu w [l,r] */
p = l + rand() % (r-l+1);
```

```
a[l]<->a[p]; /* zamiana */
```
albo jako element dzielący wybierać medianę z pewnej próbki elementów ciągu, np. pierwszy, środkowy i ostatni.

2 Dolne ograniczenie na złożoność problemu sortowania

Zanim rozważymy następne algorytmy sortujące zastanowimy się nad tym, jaka jest minimalna liczba porównań potrzebnych do posortowania ciągu n różnych elementów $q = [a_0, a_1, \ldots, a_{n-1}]$. Jako model obliczeń przyjmiemy model drzew decyzyjnych.

Drzewem sortującym ciąg $q = [a_0, a_1, \ldots, a_{n-1}]$ nazywamy (binarne) drzewo decyzyjne, w którego węzłach wewnętrznych znajdują się porównania elementów ciągu, natomiast w liściach instrukcje wypisujące ciąg w postaci uporządkowanej.

Każdy algorytm sortujący przez porównania (np. SelectionSort, InsertionSort, Quicksort) daje się rozwinąć w drzewo sortujące przy założeniu ustalonego rozmiaru n.

Bez straty ogólności możemy założyć, że:

1. jest dokładnie $n!$ liści - tyle ile różnych permutacji ciągu $a_0, a_1, \ldots, a_{n-1}$;
2. każdy węzeł wewnętrzny ma dwa następniki, tzn. że każde porównanie jest istotne (obliczenie może pójść w lewo bądź w prawo od danego węzła wewnętrznego).

Klasę drzew binarnych spełniających warunki (1) i (2) oznaczmy przez $\mathcal{B}_n$.

Zauważmy, że długość ścieżki od korzenia do liścia jest równa liczbie porównań wykonywanych przez pewne obliczenie algorytmu.

Zadania znalezienia dolnych ograniczeń na liczbę porównań w przypadku pesymistycznym i oczekiwanym sprowadzają się zatem do wyznaczenia:

1. $h(T)$ - wysokości drzewa T;

2. $d(T) = \Sigma_{\text{X liść w T}} d_T(x)$ — całkowitej długości ścieżek

z klasy $\mathcal{B}_n$ drzew binarnych. Wówczas dla każdego algorytmu sortującego przez porównania, który rozwija się w drzewo sortujące zachodzą nierówności

$$W(n) \geq \ \min_{T\in\mathcal{B}_n} h(T)$$

$$A(n) \geq \ \min_{T\in\mathcal{B}_n} d(T)/n!$$

Twierdzenie 1
Dla każdego algorytmu sortującego przez porównania zachodzi:

- $W(n) \geq n\log n - 1.45n$
- $A(n) \geq n\log n - 1.45n$

3 Sortowanie pozycyjne

Wszystkie wielkości w komputerze są reprezentowane za pomocą słów binarnych. Ogólne algorytmy sortujące nie biorą tego pod uwagę.

W dalszej części używać będziemy funkcji wycinającej bity w słowie binarnym, którą w języku C można zdefiniować jak następuje:

```
unsigned long bits(unsigned long x,
                   unsigned int k, unsigned int j)
{
  return (x >> k) % (1 << j);
}
```

$$bits(x, k, j) = (x/2^k)\%2^j.$$

Przyjmować będziemy założenie, że b jest długością słowa binarnego. Wówczas bits(x,0,1) oraz bits(x,b-1,1) oznaczają odpowiednio pierwszy bit z prawej strony i pierwszy bit z lewej strony.

Najprostszą metodą sortowania pozycyjnego jest metoda liczników częstości. Załóżmy, że $m = 2^b$. Dla każdego $j = 0, 1, \ldots, m-1$ liczymy, ile razy j pojawia się w ciągu wejściowym $a_0, a_1, ..., a_{n-1}$. W oparciu o obliczone liczniki częstości umieszczamy każde a_i we właściwym miejscu w ciągu wynikowym.

```
void CountSort(unsigned long *a,
               unsigned int n, unsigned int m)
{
   unsigned long *t, p;
   unsigned int *count;

   int i, j;

   t = calloc(n, sizeof(unsigned long));
```

```
  count = calloc(m, sizeof(unsigned int));

  for (j = 0; j < m; j++)
    count[j] = 0;

  for (i = 0; i < n; i++)
    count[a[i]]++;
  /* count[j] = liczba wystapien j w a */

  for (j = 1; j < m; j++)
    count[j] += count[j-1];
  /* count[j] = liczba wystapien elementow <= j */

  for ( i = n-1; i >= 0; i-- ){
    p = a[i];
    count[p]--;
    t[count[p]] = p;
  }

  for (i = 0; i < n; i++)
    a[i] = t[i];

  free(t);
  free(count);
}
```

Analiza algorytmu CountSort jest bezpośrednia:

$$W(n, m) = A(n, m) = O(n + m)$$

$$\Delta(n, m) = \delta(n, m) = 0$$

$$S(n, m) = n + m + O(1)$$

Do zalet algorytmu CountSort należy przede wszystkim szybkość działania, gdy $m = O(n)$. Wadą jest dodatkowe obciążenie pamięci. Zauważmy, że w zapisanej postaci algorytm CountSort jest stabilny (dla stabilności istotna jest kolejność wpisywania od a[n-1] do a[0]).

Algorytm CountSort można stosować tylko dla niedużych wartości m (tak, aby można było użyć tablicy count rozmiaru m). Dla większych wartości m możemy stosować podobną metodę, ale z dzieleniem liczb do posortowania na części, na których może zadziałać CountSort.

Dzielimy słowa o b bitach na b/e grup e bitowych. Najpierw sortujemy ciąg liczb używając CountSort względem ostatniej (najmniej znaczącej) grupy bitów. Następnie sortujemy względem przedostatniej grupy bitów itd. aż w końcu względem pierwszej (najbardziej znaczącej) grupy bitów. Istotna jest przy tym stabilność algorytmu CountSort: o ostatecznej pozycji elementu decyduje pierwsze e bitów; jeśli pierwsze e bitów są takie same wtedy decydują następne grupy bitów.

```
void RadixSort(unsigned long *a, unsigned int n)
{
   #define Byte 8

   unsigned int b = Byte*sizeof(unsigned long);
   unsigned int e = 2,
                m = 1 << e; /* m = 2^e */

   unsigned long *t, p;
```

```
unsigned int count[m];

int i, j,
    pass, nofpasses, key;

t = calloc(n, sizeof(unsigned long));

nofpasses = b/e;
if ( b == (nofpasses*e) )
  nofpasses--;

for (pass = 0; pass <= nofpasses; pass++){
  for (j = 0; j < m; j++)
    count[j] = 0;

  for (i = 0; i < n; i++){
    key = bits(a[i],pass*e,e);
    count[key]++;
  }

  for (j = 1; j < m; j++)
    count[j] += count[j-1];


  for ( i = n-1; i >= 0; i-- ){
    key = bits(a[i],pass*e,e);
    count[key]--;
    t[count[key]] = a[i];
  }
```

```
    for (i = 0; i < n; i++)
      a[i] = t[i];
  }

  free(t);
}
```

Analiza złożoności obejmuje więcej parametrów:

$$W(n,m) = A(n,m) = O(\frac{b}{e}(n{+}m)) = O(n{+}m) \text{ gdy } \frac{b}{e} = O(1)$$

$$\Delta(n,m) = \delta(n,m) = 0$$

$$S(n,m) = n + m + O(1)$$

Algorytm RADIXSORT nie jest algorytmem sortującym przez porównania - wymaga zapisu elementów w układzie pozycyjnym. Do jego wad należą także:

1. spory współczynnik proporcjonalności, z czego wynika, że jest wolny dla małych rozmiarów n;
2. spore obciążenie pamięci, szczególnie, gdy n jest duże.

Główną zaletą RadixSort jest liniowość jego zlożoności czasowej (przy odpowiednich zalożeniach dotyczących związku n z długością słowa maszyny). Dla "średnich" wartości n może być lepszy niż algorytm QuickSort. Algorytm RadixSort (po odpowiednich modyfikacjach) może być użyty do sortowania słów nad skończonym alfabetem.