НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»
Кафедра информатики и процессов управления (№17)

Дисциплина «Информатика (основы программирования)»

Методические указания

Тематическое занятие 10

Усовершенствованные методы сортировки.

Содержание

Сложность алгоритмов и ее оценка

Понятие сложности алгоритма

Одной из самых важных характеристик алгоритма является оценка его *эффективности*, которая подразделяется на два вида: временну́ю и пространственную. ***Временна́я эффективность*** является индикатором скорости работы алгоритма. ***Пространственная эффективность*** показывает сколько дополнительной оперативной памяти нужно для работы алгоритма.

Для оценки качества алгоритма вводится понятие ***сложность алгоритма***. Чем большее время и объем памяти требуется для реализации алгоритма, тем ниже его эффективность и выше сложность.

С каждой задачей связывается некоторое число **n**, которое выражает меру количества входных данных и называется ***размером задачи***. Например, в случае одномерного массива n – это количество его элементов.

Время, затрачиваемое на решение задачи с помощью некоторого алгоритма, выраженное через n, называют ***временно́й сложностью алгоритма.*** Поведение функции временно́й сложности при увеличении размера задачи **n** называют ***асимптотической*** временно́й сложностью.

Далее будем рассматривать только временну́ю сложность алгоритма.

Предположим, что длительность выполнения любой операции над элементами данных не зависит от ее операндов и самого вида операции. Тогда время работы программы можно измерить *количеством действий*.

Порядок сложности

Главную роль в понятии сложности алгоритма играет *характер возрастания* сложности при увеличении размера задачи n. Лучший способ сравнения эффективности алгоритмов состоит в сопоставлении их *порядков сложности*. Порядок сложности алгоритма также выражается через количество обрабатываемых данных.

Порядок сложности алгоритма – это функция, доминирующая над точным выражением сложности. Функция $f(n)$ имеет порядок $O(g(n))$, если существует положительная константа C и натуральное n_0 такие, что $f(n) \le C \cdot g(n)$ при $n > n_0$.

Говорят, что $f(n)$ есть функция сложности порядка $g(n)$ для больших n. Например, обозначение $O(n^2)$ читается как функция сложности порядка n^2.

Существуют три важных правила для определения функции сложности:

Правило	Пример
$O(C \cdot f) = C \cdot O(f) = O(f), \quad C=const$	$O(3 \cdot n) = O(n)$
$O(f \cdot g) = O(f) \cdot O(g)$	$O(n^2) = O(n) \cdot O(n)$
$O(f+g)$ равна доминанте $O(f)$ и $O(g)$	$O(n^3+n^2) = O(n^3)$

Виды функции сложности

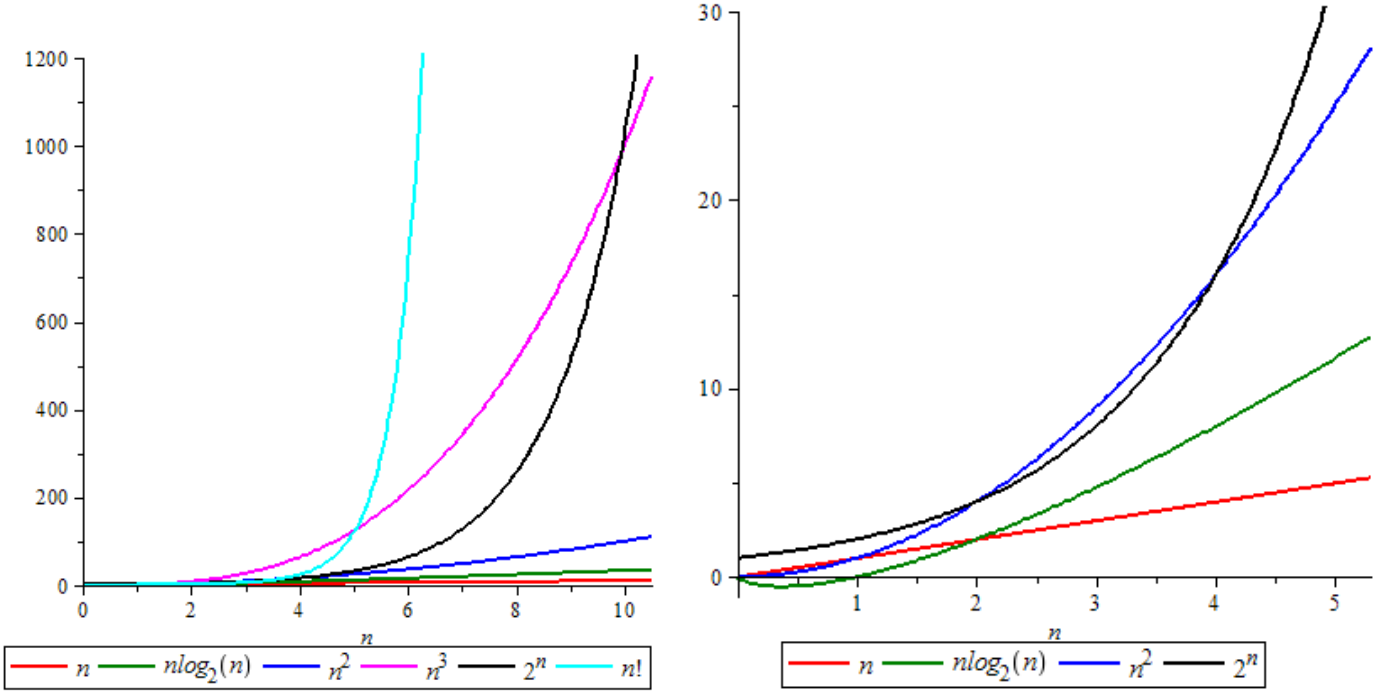
Для оценки порядка сложности реальных алгоритмов используют следующие виды функций сложности:

Функция сложности	Тип зависимости
$O(1)$	константная
$O(\log_2 n)$	логарифмическая
$O(n)$	линейная
$O(n \cdot \log_2 n)$	порядка $n \cdot \log_2 n$
$O(n^2), O(n^3), O(n^\alpha)$	полиномиальная
$O(2^n)$	экспоненциальная
$O(n!)$	факториальная

Приближенные значения данных функций при различных n:

$\log_2 n$	n	$n \cdot \log_2 n$	n^2	n^3	2^n	$n!$
3,3	10	$3{,}3 \cdot 10^1$	10^2	10^3	10^3	$3{,}6 \cdot 10^6$
6,6	10^2	$6{,}6 \cdot 10^2$	10^4	10^6	$1{,}3 \cdot 10^{30}$	$9{,}3 \cdot 10^{157}$
10,0	10^3	$1{,}0 \cdot 10^4$	10^6	10^9		
13,3	10^4	$1{,}3 \cdot 10^5$	10^8	10^{12}		
16,6	10^5	$1{,}7 \cdot 10^6$	10^{10}	10^{15}		
19,9	10^6	$2{,}0 \cdot 10^7$	10^{12}	10^{18}		

Графики данных функций (кроме функции $\log_2 n$, которая растет медленнее линейной):



Оценка сложности

Определение сложности алгоритма в основном сводится к анализу циклов и рекурсивных вызовов. Алгоритмы без циклов и рекурсивных вызовов имеют константную сложность.

Например, для алгоритм обработки элементов массива

```
for (i=1; i<=n; i++)
  a[i-1]=i;
```

имеет сложность порядок $O(n)$, так как тело цикла выполняется n раз, и сложность тела цикла равна $O(1)$.

Для вложенного цикла:

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    a[i-1,j-1]=i*j;
```

порядок сложности равен $O(n^2)$, поскольку оба цикла зависят от размера одной и той же переменной.

Методы сортировки со сложностью порядка $n \cdot log_2 n$

Принципы и подходы

Все рассмотренные простые методы сортировки (методы вставки, выбора и обмена) имеют сложность порядка $O(n^2)$.

Далее рассмотрим три метода сортировки, порядок сложности каждого из которых в среднем равен $O(n \cdot log_2 n)$, (в скобках указаны фамилии разработчиков и год):

- ***быстрая*** сортировка (Хоар, 1962) – ***Quicksort*** ;
- сортировка ***слиянием*** (фон Нейман, 1945) – ***Mergesort*** ;
- ***пирамидальная*** сортировка (Уильямс и Флойд, 1964) – ***Heapsort, «сортировка кучей»***.

Общие описания

Вначале зададим количество элементов массива в виде символической константы `n` :

```
#define n 20
```

Пусть исходный массив `A` из `n` элементов объявлен как *глобальная* переменная:

```
int A[n];
```

Условимся, что далее везде сортировка проводится *по возрастанию* значений элементов массива.

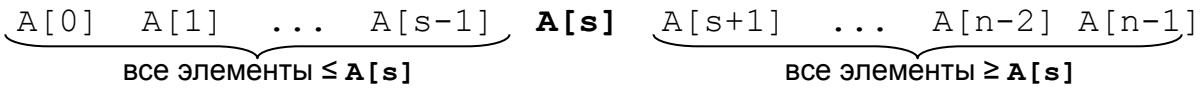
Чтобы сделать реализацию методов более краткой и наглядной, опишем функцию `swap`, которая меняет местами два элемента массива.

```
void swap(int *x, int *y)
{
  int temp; /* temp - вспомогательная переменная */
  temp=*x;
  *x=*y;
  *y=temp;
}
```

Быстрая сортировка (метод Хоара)

Принцип метода

Фиксируется какой-либо ***опорный*** элемент (ключ), относительно которого все меньшие элементы перемещаются влево, а бóльшие – вправо:



Очевидно, что после этого опорный элемент `A[s]` находится в своей окончательной позиции, а все остальные элементы массива разделены на две независимые части.

Затем процесс повторяется для каждой части отдельно. Повторение удобно реализовать с помощью рекурсии.

Реализация алгоритма

Пусть `l` и `r` – левый и правый счетчики (типа `int`). Перед началом работы `l` и `r` принимают значения индексов крайнего левого и крайнего правого элементов сортируемой части массива `A[0]...A[n-1]`, соответственно.

Выберем в качестве опорного элемента `pivot` элемент из середины массива – элемент с индексом, равным `(l+r)/2` (здесь операция «`/`» – частное от целочисленного деления):

```
pivot=A[(l+r)/2];
```

Теперь в цикле будем искать слева от `A[s]` бóльшие элементы, а справа – меньшие, и менять их местами, пока счетчики ни встретятся (или пересекутся):

```
while (l<=r) {
  while (A[l]<pivot) l++;
  while (A[r]>pivot) r--;
  if (l<=r) {
      swap(&A[l],&A[r]); /*перестановка двух элементов*/
      l++;
      r--;
  }
}
```

В результате – массив разделен на две части, а опорный элемент занял свое окончательное положение с индексом `s`.

Теперь необходимо повторить данную процедуру для каждой из частей массива `A[0]...A[s-1]` и `A[s+1]...A[n-1]` по отдельности. И так далее.

Пример реализации

Пример реализации алгоритма быстрой сортировки.

Реализация быстрой сортировки включает рекурсивную функцию `QuickSort`, в которую передаются значения индексов крайнего левого (`first`) и крайнего правого (`last`) элементов сортируемой части массива.

```
void QuickSort(int first, int last)
  {
      int pivot; /*опорное значение*/
      int l,r; /*левый и правый счетчики*/
      l=first; r=last;
      pivot=A[(l+r)/2]; /*определение опорного значения*/
      while (l<=r) {
        while (A[l]<pivot) l++;
        while (A[r]>pivot) r--;
        if (l<=r) {
          swap(&A[l],&A[r]); /*перестановка двух элементов*/
          l++;
          r--;
        }
      }                              /*Рекурсивная сортировка:*/
      if (first<r) QuickSort(first,r); /*- левого участка, */
      if (l<last) QuickSort(l,last);   /*- правого участка.*/
  }
```

В программе достаточно вызвать функцию `QuickSort` с начальными параметрами:

```
QuickSort(0,n-1);
```

Сортировка слиянием

Принцип метода

Метод основан на идее слияния двух отсортированных частей массива. Поэтому первоначально массив разделяется на две части, которые сортируются независимо друг от друга. Затем результаты объединяются («сливаются») в единый упорядоченный массив. С каждой из частей выполняются аналогичные действия, до тех пор, пока количество элементов в сортируемой части не станет равно двум. В этом случае выполняется простое сравнение элементов и их перестановка, если она необходима.

Реализация алгоритма

Вначале рассмотрим промежуточную задачу. Допустим, имеется два фрагмента массива, каждый из которых отсортирован. Необходимо объединить («слить») их в один, тоже упорядоченный фрагмент.

Идея решения этой задачи состоит в сравнении очередных элементов каждого фрагмента и переносе меньшего из них во *вспомогательный массив*. Дальнейшее продвижение происходит по тому фрагменту, из которого взят элемент. Процесс продолжается до исчерпания элементов одного из массивов, после чего остаток другого массива дописывается во вспомогательный массив.

При рекурсивной сортировке со слиянием исходный массив делится на две части, длина которых отличается не более, чем на единицу. Эти части рекурсивно сортируются и затем сливаются с помощью вспомогательного массива. Возврат из рекурсии происходит, если длина части уменьшается до одного элемента.

Пример реализации

Пример реализации рекурсивного алгоритма сортировки со слиянием. Вначале составим функцию `Merge()` для слияния двух упорядоченных по возрастанию фрагментов массива, в которую передаются значения индексов первого элемента левого фрагмента (`first`) и последнего элемента правого фрагмена (`last`) массива.

```
void Merge(int first, int last)
  {
    int B[n]; /*вспомогательный массив*/
    int k; /*индекс во вспомогательном массиве*/
    int l,r; /*левый и правый счетчики*/
    int m; /*индекс среднего элемента*/
```

```
    m=(first+last)/2;
    l=first; r=m+1;
    k=0;
    while (l<=m && r<=last) { /* Пока не закончился   */
      if (A[l]<=A[r]) {       /*хотя бы один фрагмент.*/
        B[k]=A[l]; l++; }
      else {
        B[k]=A[r]; r++; }
      k++; } /*Один из фрагментов закончился.*/
    /*Переносим остаток другого фрагмента во вспомогательный
      массив.*/
    while (l<=m) {
      B[k]=A[l]; l++; k++; }
    while (r<=last) {
      B[k]=A[r]; r++; k++; }
    for (l=0; l<k; l++)
      A[first+l]=B[l];
  }
```

Результат работы – отсортированный массив.

Составим рекурсивную функцию сортировки слиянием `MergeSort()`, которая вначале разбивает исходный массив на пары упорядоченных фрагментов, а затем попарно объединяет их с помощью функции `Merge()`.

```
void MergeSort(int fst, int lst)
  {
    int m; /*индекс среднего элемента*/
    if (fst<lst)
      if (lst-fst==1) {
        if (A[lst]<A[fst])
          swap(&A[fst],&A[lst]);
      }
      else {
        m=(fst+lst)/2;
        MergeSort(fst,m);
        MergeSort(m+1,lst);
        Merge(fst,lst);
      }
  }
```

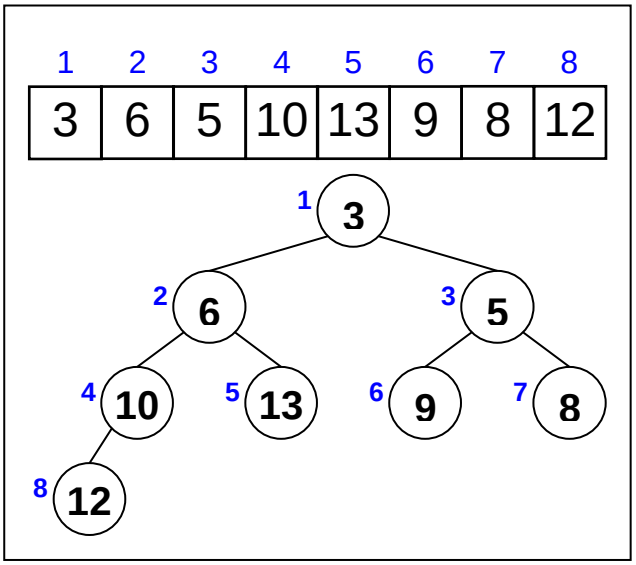
Теперь для запуска сортировки слиянием достаточно вызвать функцию `MergeSort(0,n-1);`

Пирамидальная сортировка

Принцип метода

Элементы массива $A[1],\ldots,A[n]$ образуют ***пирамиду***, если **для всех** значений i выполняются условия:

$$A[i] \le A[2 \cdot i] \text{ и } A[i] \le A[2 \cdot i+1].$$



Например, элементы массива **8,10,3,6,13,9,5,12** не образуют пирамиду (т.к. $A[1] > A[3]$, $A[2] > A[4]$ и т.д.), а элементы **3,6,5,10,13,9,8,12** – образуют.

Часть элементов массива $A[n/2+1],\ldots,A[n]$ независимо от их значений образуют пирамиду.

Данные хранятся в массиве, а пирамида – это абстракция, она заполняется сверху вниз, слева направо.

В ***«вершине»*** пирамиды находится минимальный элемент. Поместим его в результат, а на его место поставим максимально допустимое в данном диапазоне значение и протолкнем его по пирамиде. После этого в вершине пирамиды опять появится очередной минимальный элемент. Поместим его в результат. Процесс продолжается до тех пор, пока в пирамиде не останется элементов (все элементы «протолкнули» через пирамиду). В результате получим отсортированный массив.

С учетом того, что в языке С индексы элементов массива нумеруются с нуля, перепишем приведенные выше правила. Тогда элементы массива $A[0],\ldots,A[n-1]$ образуют пирамиду, если для всех значений i ($0 \le i \le n-1$) выполняются условия:

$$A[i] \le A[2 \cdot i+1] \text{ и } A[i] \le A[2 \cdot i+2].$$

А часть элементов массива $A[n/2],\ldots,A[n-1]$ независимо от их значений образуют пирамиду.

Пример реализации алгоритма

Пример реализации пирамидальной сортировки *по убыванию* значений элементов массива.

Выделим «проталкивание» одного элемента через пирамиду в отдельную функцию `TreeRebuild()`, в которую передаются параметры `r` – номер проталкиваемого элемента и `q` – верхнее значение индекса.

```
void TreeRebuild(int r, int q)
  {
    int v;
    int i,j;
    int pp;
```

```
    i=r; /*индекс рассматриваемого элемента*/
    v=A[i]; /*рассматриваемый элемент*/
    j=2*i+1;/*индекс элемента, с которым проводится сравнение*/
    pp=0; /*предположение, что не найдено место в пирамиде*/
    while (j<=q && !pp) {
      if (j<q)
        if (A[j]>A[j+1]) /*сравнение с меньшим элементом*/
          j++;
      if (v<=A[j])
        pp=1; /*элемент стоит на своем месте*/
      else {
        A[i]=A[j]; /*перестановка элемента*/
        i=j;
        j=2*i+1; /*прохождение дальше по пирамиде*/
      }
    }
    A[i]=v;
  }
```

Функция пирамидальной сортировки `TreeSort()`.

```
void TreeSort()
  {
    int k,i;
    k=n/2; /*эта часть массива является пирамидой*/
    for (i=k-1; i>=0; i--)
      TreeRebuild(i,n-1); /*построение пирамиды (только один раз)*/
    for (i=n-1; i>=1; i--) {
      swap(&A[0],&A[i]); /*перестановка 0-го и i-го элементов*/
      TreeRebuild(0,i-1); /* «проталкивание» i-го элемента*/
    }
}
```

Для запуска сортировки достаточно вызвать функцию

```
TreeSort();
```