Лекция 11 Сортировка

11.1. ***Сортировка. Постановка задачи***.

11.1.1. Сортировка – это упорядочение наборов однотипных данных, для которых определено отношение линейного порядка (например, <) по возрастанию или по убыванию. Здесь будут рассматриваться целочисленные данные и отношение порядка "<".

11.1.2. В стандартную библиотеку **stdlib** входит функция **qsort**:

```
void qsort (void *buf, size_t num, size_t size,
                        int(*compare)());
```

Функция **qsort** сортирует (по возрастанию) массив с указателем **buf**, используя алгоритм быстрой сортировки *Ч.Э.Р. Хоара*, который считается одним из лучших алгоритмов сортировки общего назначения. Параметр **num** задает количество элементов массива **buf**, параметр **size** – размер (в байтах) каждого элемента массива **buf**.

Функция, указатель на которую передается в **qsort** в качестве аргумента, соответствующего параметру **int(*compare)()**, должна иметь описание:

int *имя функции* (const void *arg1, const void *arg2)

Эта функция должна возвращать целое < 0, если ***arg1*** < ***arg2***,
целое = 0, если ***arg1*** = ***arg2***,
целое > 0, если ***arg1*** > ***arg2***,

11.2. ***Указатель на функцию***.

11.2.1. Каждая функция располагается в памяти по определенному адресу, который можно присвоить указателю в качестве его значения. Адресом функции является ее точка входа (при вызове функции управление передается именно на эту точку). Указатель функции можно использовать вместо ее имени при вызове этой функции. Указатель «лучше» имени тем, что его можно передавать другим функциям в качестве их аргумента. Имя функции **f()** без скобок и аргументов (**f**) по определению является указателем на функции **f()** (аналогия с массивом).

11.2.2. ***Пример***. Сравнение двух строк символов, введенных пользователем (функция **check()**).

```
#include <stdio.h>
#include <string.h>
void check(char *a, char *b, int (*cmp)(const char*, const
     char*));
int main() {
     char s1[80], s2[80];
       /*объявление указателя на функцию */
     int (*p)(const char *, const char *);
       /* указателю p присваивается адрес функции strcmp() */
       /* из стандартной библиотеки string */
     p = strcmp;
     printf("Введите две строки \n");
     gets(s1);
     gets(s2);
     check(s1, s2, p);
     return 0;
}
```

```
void check(char *a, char *b, int (*cmp)(const char*, const
     char*)){
    printf("Проверка на совпадение \n");
    if(!(*cmp)(a, b))) printf("Равны");;
    else printf("Не равны");
}
```

11.2.3. Пояснения к примеру.

(1) Объявление **`int (*p)(const char *, const char *);`** сообщает компилятору, что **`p`** – указатель на функцию, имеющую два параметра типа **`const char *`** и возвращающую значение типа **`int`**. Скобки вокруг **`*p`** нужны, так как операция **`*`** имеет более низкий приоритет, чем **`()`**. Если написать **`int *p(...)`** получится, что объявлен не указатель функции, а функция **`p`**, которая возвращает указатель на целое.

(2) **`(*cmp)(a, b)`** эквивалентно **`cmp(a, b)`** (имя функции имеет те же свойства, что и имя массива – 11.1.3) и при создании языка Си вариант **`cmp(a, b)`** очень нравился его авторам, но потом выяснилось, что использование этого варианта во многих случаях мешает правильно понять программу при ее чтении.

(3) У функции **`check`** три параметра: два указателя на тип **`char`** и указатель на функцию **`cmp`**. Указатели **`p`** и **`cmp`** имеют одинаковый формат, что позволяет использовать **`p`** в качестве аргумента, соответствующего параметру **`p`**.

(4) В данном случае использование указателя функции позволяет менять программу сравнения и тем самым получается более общий алгоритм. Например, для сравнения строк можно использовать не библиотечную функцию **`strcmp()`**, а следующую функцию **`compvalues()`**, которая, используя функцию **`atoi()`** из стандартной библиотеки **`stdlib`**, сравнивает числа, записанные во входных строках.

```
int compvalues(const char *a, const char *b) {
     if(atoi(a) == atoi(b)) return 0;
     else return 1;
}
```

11.3. ***Алгоритмы сортировки.***

11.3.1. Почему **`qsort`** считается лучшим алгоритмом сортировки общего назначения? Для ответа на этот вопрос рассмотрим *проблему сортировки* более подробно.

11.3.2. Даже если считать алгоритм Хоара лучшим алгоритмом сортировки общего назначения библиотечной функции **`qsort`** недостаточно, чтобы покрыть все потребности.

(1) **`qsort`** сортирует только массивы (не может сортировать, например, связанные списки).

(2) **`qsort`**, будучи ориентированной на широкий набор типов обрабатываемых данных, работает медленнее, чем специализированная функция сортировки (например, за счет вызова функции для сравнения).

11.3.3. В некоторых ситуациях алгоритм быстрой сортировки работает плохо (это будет показано ниже).

11.3.4. ***Простейший алгоритм сортировки***: сведение сортировки к задаче нахождения максимального (минимального) из *n* чисел.

11.3.4.1. Нахождение максимума *n* чисел (*n* сравнений):

Числа содержатся в массиве **`int a[n]`**

```
max = a[0];
```

```
for(i = 1; i < n; i++) if(a[i] > max) max = a[i];
```

11.3.4.2. ***Алгоритм сортировки***: находим максимальное из n чисел, получаем последний элемент отсортированного массива (n сравнений); находим максимальное из n – 1 оставшихся чисел, получаем предпоследний элемент отсортированного массива (еще n – 1 сравнений); и так далее.

11.3.4.3. Общее количество сравнений: $1 + 2 + \ldots + n\text{-}1 + n = n(n - 1)$. Сложность алгоритма $O(n^2)$. Скорость выполнения алгоритма обратно пропорциональна его сложности.

11.3.5. Классификация алгоритмов сортировки.

11.3.5.1. Различают *внешнюю* и *внутреннюю* сортировку. Здесь рассматривается только внутренняя сортировка, когда сортируемый массив находится в основной памяти компьютера. Внешняя сортировка применяется к записям внешних файлов.

11.3.5.2. Существует три общих метода внутренней сортировки:

(1) *сортировка обменами*: рассматриваются соседние элементы сортируемого массива и при необходимости меняются местами;

(2) *сортировка выборкой*: см. 11.3.4.2.

(3) *сортировка вставками*: сначала сортируются два элемента массива, потом выбирается третий элемент и вставляется в нужную позицию относительно первых двух и т.д.

11.3.6. Оценка сложности алгоритмов сортировки.

11.3.6.1. Скорость сортировки определяется количеством сравнений и количеством обменов (обмены занимают больше времени). Эти показатели интересны для худшего и лучшего случаев, а также интересно их среднее значение.

11.3.6.2. Кроме скорости оценивается «естественность» алгоритма сортировки: естественным считается алгоритм, который на уже отсортированном массиве работает минимальное время, а на не отсортированном работает тем дольше, чем больше степень его неупорядоченности.

11.3.6.3. Важным показателем является и объем дополнительной памяти для хранения промежуточных данных алгоритма. Для рекурсивных алгоритмов расход памяти связан с необходимостью дублировать стек, в котором расположены некоторые промежуточные данные.

11.3.6.4. Докажем, что *для любого алгоритма S внутренней сортировки массива из n элементов количество сравнений* $C_S \geq n \cdot \log_2(n)$

Можно доказать, что *для любого алгоритма S внутренней сортировки массива из n элементов количество сравнений* $C_S \geq \log_2(n!)$. В самом деле, алгоритм S можно представить в виде двоичного дерева сравнений. Так как любая перестановка индексов рассматриваемого массива может быть ответом в алгоритме, она должна быть приписана хотя бы одному листу дерева сравнений. Таким образом, дерево сравнений будет иметь не менее $n!$ листьев. Нетрудно также показать, что для высоты h_m двоичного дерева с m листьями имеет место оценка: $h_m \geq \log_2 m$. Это следует из того, что любое двоичное дерево высоты h можно достроить до полного двоичного дерева высоты h, а в полного двоичного дерева высоты h 2^h листьев. Следовательно, $m \leq 2^{h_m}$ откуда $h_m \geq \lceil \log_2 m \rceil$. Применив полученную оценку к дереву сравнений, получим $C_S \geq \log_2(n!)$. Для оценки $\log_2(n!)$ воспользуемся формулой Стирлинга $n! \approx \sqrt{2\pi n} \cdot n^n e^{-n}$, что даст требуемую оценку.

11.3.7. Сортировка методом «пузырька» (обменная).

11.3.7.1. Алгоритм. Просматриваем массив **int a[n]**, сравнивая каждый **a[i - 1]** с **a[i]** и меняя их местами, если **a[i - 1] > a[i]** («более легкие элементы всплывают: чем легче, тем выше»).

11.3.7.2. Си-функция:

```
/*Пузырьковая сортировка*/
void bubble_sort(int *a, int n) {
   int i, j;
   int tmp;
   /*Основной цикл*/
   for(j = 1; j < n; ++j)
    for(i = n - 1; i >= j; --i) {
      if(a[i - 1] > a[i]){
           tmp = a[i - 1];
           a[i - 1] = a[i];
           a[i] = tmp;
      }
    }
}
```

11.3.7.3. Общее количество сравнений (действий): $n(n - 1)/2$, так как внешний цикл выполняется $(n - 1)$ раз, а внутренний – в среднем $n/2$ раза.

11.3.7.4. Пузырьковая сортировка – алгоритм порядка n^2: время ее работы пропорционально квадрату количества сортируемых элементов. Алгоритм сортировки с помощью максимумов тоже имеет порядок n^2.

11.3.8. Сортировка вставками.

```
/*Сортировка вставками */
void insert_sort(int *a, int n) {
   int i, j;
   int tmp;
   /*Основной цикл*/
   for(j = 1; j < n; ++j) {
    tmp = a[j];
    for(i = j - 1; (i >= 0)&&(tmp < a[i]; i--) {
      a[i + 1] = a[i];
    a[i + 1] = a[i];
   }
}
```

Количество сравнений зависит от степени перемешанности массива **a**. Если массив **a** уже отсортирован, количество сравнений равно $n - 1$. Если массив **a** отсортирован в обратном порядке (наихудший случай), количество сравнений имеет порядок n^2.

11.3.9. Сортировка методом Шелла (слияние с обменом).

11.3.9.1. Метод. Сначала массив сортируется с шагом *gap* (достаточно большим), потом – с шагом $\lceil gap\ /2 \rceil$ и т.д. и наконец с шагом 1. Пример последовательности шагов: 9, 5, 3, 2, 1. Впоследствии (где-нибудь на третьем курсе) будет доказано, что метод Шелла не только сортирует, но сортирует быстро – порядок этого алгоритма $n^{1.2}$.

11.3.9.2. Последовательности шагов, являющихся степенями двойки уменьшают эффективность сортировки Шелла, так как увеличивается вероятность многократного сравнения одних и тех же пар. Но даже в этом случае алгоритм Шелла сходится!

11.3.9.3. Текст Си-программы сортировки Шелла (для последовательности шагов: 9, 5, 3, 2, 1).

```
/*Сортировка Шелла*/
void Shell_sort(int *a, int n) {
   int i, j, k, gap;
   int x, g[5];
   g[0] = 9; g[1] = 5; g[2] = 3; g[3] = 2; g[4] = 1;
   /*Основной цикл*/
   for(k = 0; k < 5; k++) {
    gap = g[k];
    for(i = gap; i < n; ++i) {
      x = a[i];
      for(j = i - gap; (x < a[j])&&(j >= 0; j -= gap) {
           a[j + gap] = a[j];
      a[j] = x;
    }
   }
}
```