Практическая работа №7

Методы сортировки. Оценка сложности алгоритма

Цель работы: изучить алгоритмы сортировки. Определить достоинства и недостатки каждого из рассмотренных алгоритмов.

Теоретическая часть

1. Оценка сложности алгоритма

Сложность алгоритмов обычно оценивают по времени выполнения или по используемой памяти. В обоих случаях сложность зависит от размеров входных данных: массив из 100 элементов будет обработан быстрее, чем аналогичный из 1000. При этом точное время мало кого интересует: оно зависит от процессора, типа данных, языка программирования и множества других параметров.

Допустим, некоторому алгоритму нужно выполнить $4n^3 + 7n$ условных операций, чтобы обработать n элементов входных данных. При увеличении n на итоговое время работы будет значительно больше влиять возведение n в куб, чем умножение его на 4 или же прибавление 7n. Тогда говорят, что временная сложность этого алгоритма равна O(n3), т. е. зависит от размера входных данных кубически.

Использование заглавной буквы O (или так называемая O-нотация) пришло из математики, где её применяют для сравнения асимптотического поведения функций. Формально O(f(n)) означает, что время работы алгоритма (или объём занимаемой памяти) растёт в зависимости от объёма входных данных не быстрее, чем некоторая константа, умноженная на f(n).

Примеры

O(n) — линейная сложность

Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в неотсортированном массиве. Нам придётся пройтись по всем n элементам массива, чтобы понять, какой из них максимальный.

O(log n) — логарифмическая сложность

Простейший пример — бинарный поиск. Если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение, методом деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива — там его точно нет. Если же меньше, то наоборот — отбросим начальную половину. И так будем продолжать делить пополам, в итоге проверим log n элементов.

$O(n^2)$ — квадратичная сложность

Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как n * n, т. е. n^2.

Бывают и другие оценки по сложности, но все они основаны на том же принципе.

Также случается, что время работы алгоритма вообще не зависит от размера входных данных. Тогда сложность обозначают как O(1). Например, для определения значения третьего элемента массива не нужно ни запоминать элементы, ни проходить по ним сколько-то раз. Всегда нужно просто дождаться в потоке входных данных третий элемент и это будет результатом, на вычисление которого для любого количества данных нужно одно и то же время.

Аналогично проводят оценку и по памяти, когда это важно. Однако алгоритмы могут использовать значительно больше памяти при увеличении размера входных данных, чем другие, но зато работать быстрее. И наоборот. Это помогает выбирать оптимальные пути решения задач исходя из текущих условий и требований.

Наглядно

Время выполнения алгоритма с определённой сложностью в зависимости от размера входных данных при скорости 106 операций в секунду:

размер / сложность	10	20	30	40	50	60
n	0,00001 сек.	0,00002 сек.	0,00003 сек.	0,00004 сек.	0,00005 сек.	0,00005 сек.
n^2	0,0001 сек.	0,0004 сек.	0,0009 сек.	0,0016 сек.	0,0025 сек.	0,0036 сек.
n^3	0,001 сек.	0,008 сек.	0,027 сек.	0,064 сек.	0,125 сек.	0,216 сек.
n^5	0,1 сек.	3,2 сек.	24,3 сек.	1,7 минут	5,2 минут	13 минут
2^n	0,0001 сек.	1 сек.	17,9 минут	12,7 дней	35,7 веков	366 веков
3^n	0,059 сек.	58 минут	6,5 лет	3855 веков	$2x10^8$ веков	$1,3x10^{13}$ веков

2. **Алгоритм пузырьковой сортировки** — это довольно простой в реализации алгоритм для сортировки массивов. Можно встретить и другие названия: **пузырьковая сортировка**, **Buble sort** или **сортировка простыми обменами** — но все они будут обозначать один и тот же алгоритм.

Назван так, потому что большее или меньшее значение «всплывает» (сдвигается) к краю массива после каждой итерации, это будет видно в примере.

Сложность этого алгоритма выражается формулой **O(n^2)** (n в степени 2). Алгоритм работает медленно с большими массивами, а поэтому малоэффективен и **на практике используется редко**, чаще всего в учебных целях.

Для сортировки массивов на практике используют другие более быстрые алгоритмы, один из них — QuickSort (быстрая сортировка).

Функция для быстрой сортировки включена во многие стандартные библиотеки языков программирования, например в языке C функция **qsort()** из стандартной библиотеки.

Алгоритм работает очень просто. Программа проходит по всем элементами массива по порядку. Каждый текущий элемент сравнивается с соседним и, если он меньше/больше(если сортируем по убыванию/возрастанию соответственно) меняется местами с соседним.

Пример работы алгоритма пузырьковой сортировки

Рассмотрим пример работы алгоритма с массивом, состоящим из четырех элементов.

Имеется массив **[4, 5, 2, 6]**. Сортировать его будем по убыванию.

N — количество элементов в массиве. **i** — номер прохода. Алгоритм будет делать проходы по массиву, всего **N-1 проходов до N-i ячейки в каждой итерации** для любого массива.

Первый проход циклом от первого до N-1 элемента, сравнение условием и перемена местами в случае удовлетворения условия — если левый элемент меньше правого.

4 5 2 6

Сравниваем 4 и 5, 4 меньше 5, а значит мы их **меняем местами**.

Следующий проход.

5 **4 2** 6

Сравниваем 4 и 2, 4 не меньше 2, а значит **пропускаем** и идем дальше.

5 4 **2 6**

Сравниваем 2 и 6, 4 меньше 6, а значит мы их **меняем местами**.

Теперь мы у нас массив **[5, 4 ,6, 2]**. Как видно, он еще не упорядочен до конца. После первого прохода в конец массива передвинулось самое маленькое значение, теперь нам нужно сделать еще проход до элемента N-2 (ведь идет 2-ая итерация).

Делаем проход, начиная с первого элемента.

5 4 6 2

Сравниваем 5 и 4, 5 не меньше 4, а значит **пропускаем** и идем дальше.

5 **4 6** 2

Сравниваем 6 и 4, 6 меньше 4, а значит мы их **меняем местами**. Мы достигли элемента N-2, завершаем итерацию.

Теперь мы имеем массив **[5, 6, 4, 2]**. 2 последних элемента упорядочены как нужно. Для завершения нужно выполнить еще один проход до N-3.

5 6 4 2

Сравниваем 5 и 6, 5 меньше 6, а значит мы их **меняем местами**. Мы достигли элемента N-3, завершаем итерацию.

В итоге на выходе мы получили массив упорядоченный по убыванию — **[6, 5, 4, 2]**.

Реализация алгоритма на языке C++

Программа выполнит последовательно следующие действия:

1. Установит размер массива, прося пользователя ввести числовое значение.
2. Заполнит массив значениями, введенными пользователем для каждого элемента массива.
3. Выведет исходный массив.
4. Отсортирует массив по убыванию.
5. Выведет отсортированный массив.

3. Сортировка Шелла

В 1959 году американский ученый Дональд Шелл опубликовал алгоритм сортировки, который впоследствии получил его имя – «Сортировка Шелла». Этот алгоритм может рассматриваться и как обобщение пузырьковой сортировки, так и сортировки вставками.

Сортировка Шелла во многих случаях медленнее, чем быстрая сортировка, но она имеет ряд **преимуществ**, например отсутствие деградации при неудачных наборах данных – в этом случае быстрая сортировка (qsort) легко деградирует до $O(n^2)$, что хуже, чем худшее гарантированное время для сортировки Шелла.

Идея метода заключается в сравнение разделенных на группы элементов последовательности, находящихся друг от друга на некотором расстоянии. Изначально это расстояние равно **d** или **N/2**, где **N** — общее число элементов. На первом шаге каждая группа включает в себя два элемента расположенных друг от друга на расстоянии **N/2**; они сравниваются между собой, и, в случае необходимости, меняются местами. На последующих шагах также происходят проверка и обмен, но расстояние **d** сокращается на **d/2**, и количество групп, соответственно, уменьшается. Постепенно расстояние между элементами уменьшается, и на **d=1** проход по массиву происходит в последний раз.

Далее, на примере последовательности целых чисел, показан процесс сортировки массива методом Шелла. Для удобства и наглядности, элементы одной группы выделены одинаковым цветом.

Исходный массив	5	3	8	0	7	4	9	1	6	2
d=5	4	3	1	0	2	5	9	8	6	7
d=2	1	0	2	3	4	5	6	7	9	8
d=1	0	1	2	3	4	5	6	7	8	9

Первое значение, соответствующее расстоянию **d** равно **10/2=5**. На каждом шаге оно уменьшается вдвое. Элементы, входящие в одну группу, сравниваются и если значение какого-либо элемента, стоящего левее того с

которым он сравнивается, оказывается больше (сортировка по возрастанию), тогда они меняются местами. Так, элементы путем внутригрупповых перестановок постепенно становятся на свои позиции, и на последнем шаге (**d=1**) сортировка сводится к проходу по одной группе, включающей в себя все **N** элементов массива. При этом число требуемых обменов оказывается совсем небольшим.

Пример 2.

Пример 3.

Среднее время работы алгоритма зависит от длин промежутков h, на которых будут находиться сортируемые элементы исходного массива

ёмкостью n на каждом шаге алгоритма. Существует несколько подходов к выбору этих значений.

1. Первоначально **Шеллом** использовалась **последовательность длин промежутков 1, 2, 4, 8, 16, 32 и т.д**. В обратном порядке она вычисляется по формулам: h1=n/2, hi=hi-1/2, h0=1, **В худшем случае сложность алгоритма составит O(n^2)**.

Код программы на C++:

```
using namespace std;
int i, j, n, d, c;
void Shell(int A[], int n) //сортировка Шелла
{
   d = n;
   d = d / 2;
   while (d > 0)
   {
      for (i = 0; i < n - d; i++)
      {
         j = i;
         while (j >= 0 && A[j] > A[j + d])
         {
            c = A[j];
            A[j] = A[j + d];
            A[j + d] = c;
            j--;
         }
      }
      d = d / 2;
   }
   for (i = 0; i < n; i++) cout << A[i] << " "; //вывод массива
}
//главная функция
int main()
{
   setlocale(LC_ALL, "Rus");
   cout << "Размер массива > "; cin >> n;
   int* A = new int[n]; //объявление динамического массива
   for (i = 0; i < n; i++) //ввод массива
   {
      cout << i + 1 << " элемент > "; cin >> A[i];
   }
   cout << "\nРезультирующий массив: ";
   Shell(A, n);
   delete[] A; //освобождение памяти
   system("pause>>void");
   return(10);
}
```

4. Быстрая сортировка

Выбирая алгоритм сортировки для практических целей, программист, вполне вероятно, остановиться на методе, называемом «Быстрая сортировка» (также «qsort» от англ. quick sort). Его разработал в 1960 году английский ученый Чарльз Хоар, занимавшийся тогда в МГУ машинным переводом. Алгоритм, по принципу функционирования, входит в класс обменных сортировок (сортировка перемешиванием, пузырьковая сортировка и др.), выделяясь при этом высокой скоростью работы.

Отличительной особенностью быстрой сортировки является операция разбиения массива на две части относительно опорного элемента. Например, если последовательность требуется упорядочить по возрастанию, то в левую часть будут помещены все элементы, значения которых меньше значения опорного элемента, а в правую элементы, чьи значения больше или равны опорному.

Вне зависимости от того, какой элемент выбран в качестве опорного, массив будет отсортирован, но все же наиболее удачным считается ситуация, когда по обеим сторонам от опорного элемента оказывается примерно равное количество элементов. Если длина какой-то из получившихся в результате разбиения частей превышает один элемент, то для нее нужно рекурсивно выполнить упорядочивание, т. е. повторно запустить алгоритм на каждом из отрезков.

Таким образом, алгоритм быстрой сортировки включает в себя два основных этапа:

- разбиение массива относительно опорного элемента;
- рекурсивная сортировка каждой части массива.

Разбиение массива.

Еще раз об опорном элементе. Его выбор не влияет на результат, и поэтому может пасть на произвольный элемент. Тем не менее, как было замечено выше, наибольшая эффективность алгоритма достигается при выборе опорного элемента, делящего последовательность на равные или примерно равные части.

В следующих пяти пунктах описана общая схема разбиения массива (сортировка по возрастанию):

1. вводятся указатели *first* и *last* для обозначения начального и конечного элементов последовательности, а также опорный элемент *mid*;
2. вычисляется значение опорного элемента (*first*+*last*)/2, и заноситься в переменную *mid*;
3. указатель *first* смещается с шагом в 1 элемент к концу массива до тех пор, пока *Mas*[*first*]>*mid*. А указатель *last* смещается от конца массива к его началу, пока *Mas*[*last*]<*mid*;
4. каждые два найденных элемента меняются местами;
5. пункты 3 и 4 выполняются до тех пор, пока first<last.

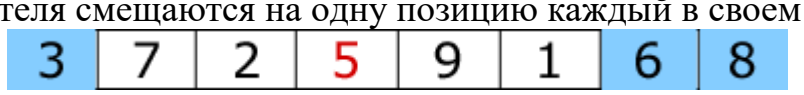
После разбиения последовательности следует проверить условие на необходимость дальнейшего продолжения сортировки его частей. Этот этап будет рассмотрен позже, а сейчас на конкретном примере выполним разбиение массива.

Имеется массив целых чисел *Mas*, состоящий из 8 элементов : *Mas*[1..8]. Начальным значением *first* будет 1, а *last* – 8. Пройденная часть закрашивается голубым цветом.

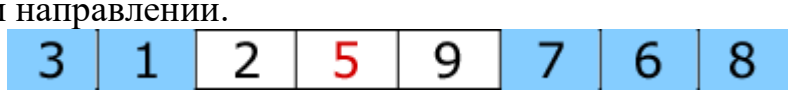
6	7	2	5	9	1	3	8

В качестве опорного элемента возьмем элемент со значением 5, и индексом 4. Его мы вычислили, используя выражение (*first*+*last*)/2, отбросив дробную часть. Теперь *mid*=5.

Первый элемент левой части сравнивается с *mid*. *Mas*[1]>*mid*, следовательно *first*остается равным 1. Далее, элементы правой части сравниваются с *mid*. Проверяется элемент с индексом 8 и значением 8. *Mas*[8]>*mid*, следовательно *last* смещается на одну позицию влево. *Mas*[7]<*mid*, следовательно *last* остается равным 7. На данный момент *first*=1, а *last*=7. Первый и седьмой элементы меняются местами. Оба указателя смещаются на одну позицию каждый в своем направлении.



Алгоритм снова переходит к сравнению элементов. Второй элемент сравнивается с опорным: *Mas*[2]>mid, следовательно *first* остается равным 2. Далее, элементы правой части сравниваются с *mid*. Проверяется элемент с индексом 6 и значением 1: *Mas*[6]<*mid*, следовательно *last* не изменяет своей позиции. На данный момент *first*=2, а *last*=6. Второй и шестой элементы меняются местами. Оба указателя смещаются на одну позицию каждый в своем направлении.



Алгоритм снова переходит к сравнению элементов. Третий элемент сравнивается с опорным: *Mas*[3]<*mid*, следовательно *first* смещается на одну позицию вправо. Далее, элементы правой части сравниваются с *mid*. Проверяется элемент с индексом 5 и значением 9: *Mas*[5]>*mid*, следовательно *last* смещается на одну позицию влево. Теперь *first*=*last*=4, а значит, условие *first*<*last* не выполняется, этап разбиения завершается.

На этом этап разбиения закончен. Массив разделен на две части относительно опорного элемента. Осталось произвести рекурсивное упорядочивание его частей.

Рекурсивное доупорядочивание

Если в какой-то из получившихся в результате разбиения массива частей находиться больше одного элемента, то следует произвести рекурсивное упорядочивание этой части, то есть выполнить над ней операцию разбиения, описанную выше. Для проверки условия «количество элементов > 1», нужно действовать примерно по следующей схеме:

Имеется массив *Mas*[*L*..*R*], где *L* и *R* – индексы крайних элементов этого массива. По окончанию разбиения, указатели *first* и *last* оказались примерно в середине последовательности, тем самым образуя два отрезка: левый от *L* до *last* и правый от *first* до *R*. Выполнить рекурсивное упорядочивание левой части нужно в том случае, если выполняется условие *L*<*last*. Для правой части условие аналогично: *first*<*R*.

Реализации алгоритма быстрой сортировки:

```
#include <iostream>
#include <ctime>
using namespace std;
const int n = 15;
```

```
int first, last;
//функция сортировки
void quicksort(int* mas, int first, int last)
{
	int mid, count;
	int f = first, l = last;
	mid = mas[(f + l) / 2]; //вычисление опорного элемента
	do
	{
		while (mas[f] < mid) f++;
		while (mas[l] > mid) l--;
		if (f <= l) //перестановка элементов
		{
			count = mas[f];
			mas[f] = mas[l];
			mas[l] = count;
			f++;
			l--;
		}
	} while (f < l);
	if (first < l) quicksort(mas, first, l);
	if (f < last) quicksort(mas, f, last);
}
//главная функция
 int main()
{
	setlocale(LC_ALL, "Rus");
	int* A = new int[n];
	srand(time(NULL));
	cout << "Исходный массив: ";
	for (int i = 0; i < n; i++)
	{
		A[i] = rand() % 10;
		cout << A[i] << " ";
	}
	first = 0; last = n - 1;
	quicksort(A, first, last);
	cout << endl << "Результирующий массив: ";
	for (int i = 0; i < n; i++) cout << A[i] << " ";
	delete[]A;
	return(10);
}
```

Задание

1. Реализовать программы трех сортировок: пузырьком, Шелла и быстрая сортировки на массивах сгенерированных случайным образом из 100 вещественных чисел в диапазоне -500 до 500. Сравнить время сортировки.
2. Построить блок-схемы сортировки Шелла (1 вариант) и быстрой сортировки (2 вариант)
3. Оформить отчет сделать выводы о сложности алгоритмов сортировки.