Свищев

**Лабораторная работа №1.**

**Оценка временной сложности алгоритмов**

Целью лабораторной работы является приобретение навыков исследования временной сложности алгоритмов и определения ее асимптотических оценок.

* + 1. Требования к содержанию, оформлению и порядку выполнения

В содержательной части отчета по выполнению лабораторной работы требуется привести описание алгоритма, выбранного согласно своему варианту, провести его анализ и определить асимптотические оценки его временной сложности. Алгоритм рекомендуется оформлять с помощью блок-схем.

* + 1. Теоретическая часть

Теоретические сведения, необходимые для выполнения лабораторной работы, представлены в лекции.

* + 1. Общая постановка задачи

Требуется провести анализ и оценку временной сложности заданного алгоритма. Варианты заданий представлены в таблице в следующем разделе.

В качестве дополнительных заданий рекомендуется программно реализовать заданный алгоритм.

* + 1. Список индивидуальных данных

Данные для выполнения лабораторной работы сведены в табл.Л2.1.

Таблица Л2.1.

Варианты заданий к лабораторной работе № 2

|  |  |
| --- | --- |
| *Вариант* | *Алгоритм* |
| 1 | Тривиальный алгоритм возведения в степень (рис. 3.1) |
| 2 | Рекурсивный алгоритм возведения в степень (рис. 3.2) |
| 3 | Алгоритм быстрого возведения в степень (рис. 3.3 а) |
| 4 | Алгоритм быстрого возведения в степень (рис. 3.3 б) |
| 5 | Алгоритм вычисления значения многочлена (рис. 3.4) |
| 6 | Алгоритм вычисления значения многочлена по схеме Горнера (рис. 3.5) |
| 7 | Алгоритм сортировки обменом (рис. 4.5) |
| 8 | Алгоритм сортировки выбором (рис. 4.7) |
| 9 | Алгоритм сортировки вставками (рис. 4.9) |
| 10 | Алгоритм быстрой сортировки (рис. 4.14, 4.16) |

* + 1. Ход работы

Рассмотрим 7 вариант. Требуется провести анализ и оценку временной сложности алгоритма сортировки обменом.

Алгоритм предназначен для сортировки пузырьком.

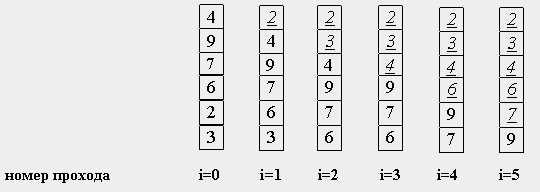
Расположим массив сверху вниз, от нулевого элемента - к последнему.

Идея метода: шаг сортировки состоит в проходе снизу вверх по массиву. По пути просматриваются пары соседних элементов. Если элементы некоторой пары находятся в неправильном порядке, то меняем их местами.



После нулевого прохода по массиву "вверху" оказывается самый "легкий" элемент - отсюда аналогия с пузырьком. Следующий проход делается до второго сверху элемента, таким образом второй по величине элемент поднимается на правильную позицию...

Делаем проходы по все уменьшающейся нижней части массива до тех пор, пока в ней не останется только один элемент. На этом сортировка заканчивается, так как последовательность упорядочена по возрастанию.



template<class T>

void bubbleSort(T a[], long size) {

long i, j;

T x;

for( i=0; i < size; i++) { // i - номер прохода

for( j = size-1; j > i; j-- ) { // внутренний цикл прохода

if ( a[j-1] > a[j] ) {

x=a[j-1]; a[j-1]=a[j]; a[j]=x;

}

}

}

}

Среднее число сравнений и обменов имеют квадратичный порядок роста: Theta(n^2), отсюда можно заключить, что алгоритм пузырька очень медленен и малоэффективен.

Тем не менее, у него есть громадный плюс: он прост и его можно по-всякому улучшать. Чем мы сейчас и займемся.

Во-первых, рассмотрим ситуацию, когда на каком-либо из проходов не произошло ни одного обмена. Что это значит?

Это значит, что все пары расположены в правильном порядке, так что массив уже отсортирован. И продолжать процесс не имеет смысла (особенно, если массив был отсортирован с самого начала!).

Итак, первое улучшение алгоритма заключается в запоминании, производился ли на данном проходе какой-либо обмен. Если нет - алгоритм заканчивает работу.

Процесс улучшения можно продолжить, если запоминать не только сам факт обмена, но и индекс последнего обмена k. Действительно: все пары соседих элементов с индексами, меньшими k, уже расположены в нужном порядке. Дальнейшие проходы можно заканчивать на индексе k, вместо того чтобы двигаться до установленной заранее верхней границы i.

Качественно другое улучшение алгоритма можно получить из следующего наблюдения. Хотя легкий пузырек снизу поднимется наверх за один проход, тяжелые пузырьки опускаются со минимальной скоростью: один шаг за итерацию. Так что массив 2 3 4 5 6 1 будет отсортирован за 1 проход, а сортировка последовательности 6 1 2 3 4 5 потребует 5 проходов.

Чтобы избежать подобного эффекта, можно менять направление следующих один за другим проходов. Получившийся алгоритм иногда называют "шейкер-сортировкой".

template<class T>

void shakerSort(T a[], long size) {

long j, k = size-1;

long lb=1, ub = size-1; // границы неотсортированной части массива

T x;

do {

// проход снизу вверх

for( j=ub; j>0; j-- ) {

if ( a[j-1] > a[j] ) {

x=a[j-1]; a[j-1]=a[j]; a[j]=x;

k=j;

}

}

lb = k+1;

// проход сверху вниз

for (j=1; j<=ub; j++) {

if ( a[j-1] > a[j] ) {

x=a[j-1]; a[j-1]=a[j]; a[j]=x;

k=j;

}

}

ub = k-1;

} while ( lb < ub );

}

Насколько описанные изменения повлияли на эффективность метода? Среднее количество сравнений, хоть и уменьшилось, но остается O(n^2), в то время как число обменов не поменялось вообще никак. Среднее(оно же худшее) количество операций остается квадратичным.

Дополнительная память, очевидно, не требуется. Поведение усовершенствованного (но не начального) метода довольно естественное, почти отсортированный массив будет отсортирован намного быстрее случайного. Сортировка пузырьком устойчива, однако шейкер-сортировка утрачивает это качество.

Реализация сортировки методом «пузырька» на Си

#define \_CRT\_SECURE\_NO\_WARNINGS // для корректной работы scanf()

#include <stdio.h>

// Функция сортировки прямым обменом (метод "пузырька")

void bubbleSort(int \*num, int size)

{

// Для всех элементов

for (int i = 0; i < size - 1; i++)

{

for (int j = (size - 1); j > i; j--) // для всех элементов после i-ого

{

if (num[j - 1] > num[j]) // если текущий элемент меньше предыдущего

{

int temp = num[j - 1]; // меняем их местами

num[j - 1] = num[j];

num[j] = temp;

}

}

}

}

int main()

{

int a[10]; // Объявляем массив из 10 элементов

// Вводим значения элементов массива

for (int i = 0; i < 10; i++)

{

printf("a[%d] = ", i);

scanf("%d", &a[i]);

}

bubbleSort(a, 10); // вызываем функцию сортировки

// Выводим отсортированные элементы массива

for (int i = 0; i<10; i++)

printf("%d ", a[i]);

getchar(); getchar();

return 0;

}

Вывод. В лабораторной работе проведен анализ алгоритма сортировки обмена. На практике метод пузырька, даже с улучшениями, работает, увы, слишком медленно. А потому - почти не применяется. Данный алгоритм простейший и он эффективен для небольших массивов.

Контрольные вопросы к защите:

1. Понятие временной сложности алгоритма.
2. Определение асимптотических оценок временной сложности.
3. Основные принципы получения асимптотических оценок.
4. *Правила анализа алгоритмов с целью определения их временной сложности.*