Белорусский государственный технологический университет

Факультет информационных технологий

Кафедра программной инженерии

 Реферат

По дисциплине «Основы алгоритмизации и программирования»

На тему «Оценки сложности алгоритмов»

Выполнила:

Студент(ка) 1 курса 7 группы

Подшиваленко Диана Игоревна

Проверил:

Белодед Николай Иванович

2024, Минск

**Содержание**

1. [Предисловие … 3](#_Предисловие)
2. [Основные характеристики алгоритмов … 4](#_1._Основные_характеристики)
3. [Нотация «О большое»… 5](О#_2._Нотация_)
4. [Случаи… 5](#_3._Случаи)
5. [Другие оценки сложности… 5](#_4._Другие_оценки)
6. [Примеры оценки сложности… 5](#_5._Примеры_оценки)
7. [Выводы … 11](#_6._Выводы)

# **Предисловие**

В данном реферате будут рассмотрены основные оценки сложности алгоритмов, а также на примере сортировок показано нахождение их сложности в нотации О большое.

# **1. Основные характеристики алгоритмов**

В мире компьютерных наук и программирования, алгоритмы играют важную роль. Они являются строительными блоками всех наших цифровых достижений - от поисковых систем и социальных сетей до медицинских диагностических систем и самоуправляемых автомобилей. Каждый раз, когда мы решаем задачу, мы обращаемся к алгоритмам, чтобы найти оптимальное решение.

Однако алгоритмы не просто набор инструкций. Они обладают уникальными характеристиками, которые делают их такими мощными и эффективными. Именно эти характеристики определяют, насколько быстро и точно алгоритм может решить задачу. Среди них можно выделить:

1) **Временная сложность**: описывает количество времени, необходимого для выполнения алгоритма в зависимости от размера входных данных. Обычно измеряется в терминах количества операций или шагов, которые выполняются алгоритмом. Временная сложность позволяет оценить эффективность алгоритма и предсказать, как он будет масштабироваться с увеличением размера входных данных.

2) **Пространственная сложность**: описывает количество памяти, необходимое для выполнения алгоритма. Обычно измеряется в терминах количества используемых переменных или объема дополнительной памяти, требуемой для хранения промежуточных данных. Пространственная сложность важна при работе с ограниченными ресурсами памяти или при обработке больших объемов данных.

3) **Оптимальность**: описывает степень, в которой алгоритм достигает наилучшего результата с учетом определенного критерия. Например, некоторые алгоритмы могут быть оптимальными с точки зрения временной сложности, то есть они выполнены за минимально возможное время. Другие алгоритмы могут быть оптимальными с точки зрения пространственной сложности или других критериев.

4) **Устойчивость**: описывает способность алгоритма правильно работать с некорректными или непредсказуемыми входными данными. Устойчивый алгоритм должен предусмотреть возможность обработки ошибок или неожиданных ситуаций, чтобы избежать нежелательных последствий.

5) **Масштабируемость**: описывает способность алгоритма эффективно работать с разными размерами входных данных. Масштабируемый алгоритм должен демонстрировать сравнительно небольшое увеличение временной или пространственной сложности при увеличении размера входных данных.

6) **Понятность и читаемость**: описывает степень, в которой алгоритм понятен и легко читается для программистов и других лиц, занимающихся его анализом или поддержкой. Читаемый алгоритм обычно обладает ясной структурой, хорошо именованными переменными и комментариями, что облегчает его понимание и сопровождение.

Однако время работы одного и того же алгоритма может отличаться на разных устройствах, поэтому это не дает оптимальной оценки алгоритма. Поэтому были предложены другие оценки сложности алгоритмов.

# **2. Нотация «О большое»**

«О» большое (Big O notation) — это математическая нотация, которая позволяет оценить, как изменяется время выполнения алгоритма или объем используемой памяти в зависимости от размера входных данных.

В данной нотации есть 2 правила:

1) **Константы откидываются.** Если имеется сложность O(13n), то число 13 можно отбросить. Тогда сложность данного алгоритма будет равна O(n).

2) **Если в оценке сложности имеется сумма, то выбирается та, которая быстрее растет.** Например, есть сложность O(7n6 + 40n4 + 100n). Тогда нужно выбрать ту функцию из суммы, которая растет быстрее остальных. В данном случае это О(7n6). Но используя предыдущее правило, окончательная сложность данного алгоритма будет O(n6).

# **3. Случаи**

У каждого алгоритма есть худший, средний и лучший сценарии работы, в зависимости от того, насколько удачно выбраны входные данные. Часто их называют случаями.

**Худший случай** – это когда входные данные требуют максимальных затрат по времени и памяти.

**Лучший случай** – это когда входные данные самые удачные. Например, при поиске нужный элемент находится вначале. Или при сортировке массив уже отсортирован.

**Средний случай** – это когда берется что-то среднее между лучшим и худшим случаем.

# **4. Другие оценки сложности**

Помимо О большого существуют и другие оценки сложности:

1. Big Omega (Ω) обозначает нижнюю границу сложности, и её правильнее использовать для поиска best case.
2. Big Theta (Θ) располагается между О и омегой и показывает точную функцию сложности алгоритма. С её помощью правильнее искать average case.
3. Small o и Small omega(Ω) находятся по краям этой иерархии и используются в основном для сравнения алгоритмов между собой.

# **5. Примеры оценки сложности**

Рассмотрим оценку сложности на примере алгоритмов сортировок. Пусть есть следующие сортировки: пузырьковая, выбором, вставками, шейкерная, расческой, Шелла, слиянием, быстрая, пирамидальная, двусторонняя.

Реализация данных сортировок:

#include <iostream>

#include<Windows.h>

#include <conio.h>

#include <random>

#include <chrono>

#include <vector>

using namespace std;

void bubble\_sort(vector<int> &b, int n) { // пузырьковая сортировка

while (n--) {

bool t = false;

for (int i = 0; i < n; i++) {

if (b[i] > b[i + 1]) {

swap(b[i], b[i + 1]);

t = true;

}

}

if (!t) {

break;

}

}

}

void insert\_sort(vector<int> &c, int n) { // сортировка вставками

for (int i = 1; i < n; i++) {

int sort = i - 1;

while (sort > -1 and c[sort] > c[sort + 1]) {

swap(c[sort], c[sort + 1]);

sort--;

}

}

}

void choice\_sort(vector<int> &d, int n) { // сортировка выбором

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

int min\_poz = i;

for (int j = i + 1; j < n; j++) {

if (d[j] < d[min\_poz]) {

min\_poz = j;

}

}

if (min\_poz != i) {

swap(d[i], d[min\_poz]);

}

}

}

void quick\_sort(vector<int> &e, int first, int last) { // быстрая сортировка

int middle;

int l = first, r = last;

middle = e[(l + r) / 2];

while (l < r) {

while (e[l] < middle) l++;

while (e[r] > middle) r--;

if (l <= r) {

swap(e[l], e[r]);

l++;

r--;

}

}

if (first < r) quick\_sort(e, first, r);

if (l < last) quick\_sort(e, l, last);

}

void twoway\_sorting(vector<int> &a, int n) { // двусторонняя сортировка

int i = 0;

while (i <= n - i - 1) {

int min = i;

int max = n - i - 1;

for (int j = i; j < n - i; j++) {

if (a[j] < a[min]) {

min = j;

}

if (a[j] > a[max]) {

max = j;

}

}

if (i == max and n - i - 1 == min) {

swap(a[i], a[n - i - 1]);

}

else if (i == max) {

int temp = a[n - i - 1];

a[n - i - 1] = a[max];

a[i] = a[min];

a[min] = temp;

}

else if (n - i - 1 == min) {

int temp = a[i];

a[i] = a[min];

a[n - i - 1] = a[max];

a[max] = temp;

}

else {

swap(a[i], a[min]);

swap(a[n - i - 1], a[max]);

}

i++;

}

}

void sheyker(vector<int> &a, int n) { // шейкерная сортировка

int left = 0;

int right = n - 1;

bool swaped = true;

while (left < right&& swaped) {

for (int i = left; i < right; i++) {

swaped = false;

if (a[i] > a[i + 1]) {

swap(a[i], a[i + 1]);

swaped = true;

}

}

right--;

for (int i = right; i > left; i--) {

if (a[i] < a[i - 1]) {

swap(a[i], a[i - 1]);

swaped = true;

}

}

left++;

}

}

void comb\_sorting(vector<int> &a, int n) { // сортировка расческой

int step = n;

bool swaped = true;

while (step > 1 || swaped) {

step = (int)(step / 1.247);

if (step < 1) step = 1;

swaped = false;

for (int i = 0; i < n - step; i++) {

if (a[i] > a[i + step]) {

swap(a[i], a[i + step]);

swaped = true;

}

}

}

}

void Shell\_sorting(vector<int> &a, int n) { // сортировка Шелла

for (int i = n / 2; i > 0; i /= 2) {

for (int j = i; j < n; j++) {

for (int k = j - i; k >= 0 && a[k] > a[k + i]; k -= i) {

swap(a[k], a[k + i]);

}

}

}

}

void merge(vector<int> &a, int left, int middle, int right) { // сортировка слиянием

int lefts = middle - left + 1;

int rights = right - middle;

int\* lefta = new int[lefts];

int\* righta = new int[rights];

for (int i = 0; i < lefts; i++) {

lefta[i] = a[left + i];

}

for (int i = 0; i < rights; i++) {

righta[i] = a[middle + 1 + i];

}

int i = 0, j = 0, k = left;

while (i < lefts && j < rights) {

if (lefta[i] <= righta[j]) {

a[k] = lefta[i];

i++;

}

else {

a[k] = righta[j];

j++;

}

k++;

}

while (i < lefts) {

a[k] = lefta[i];

i++;

k++;

}

while (j < rights) {

a[k] = righta[j];

j++;

k++;

}

}

void mergesort(vector<int> &a, int left, int right) {

if (left < right) {

int middle = left + (right - left) / 2;

mergesort(a, left, middle);

mergesort(a, middle + 1, right);

merge(a, left, middle, right);

}

}

void heapify(vector<int> &a, int size, int root) {// пирамидальная сортировка

int largest = root;

int left = 2 \* root + 1;

int right = 2 \* root + 2;

if (left < size && a[left] > a[largest]) {

largest = left;

}

if (right < size && a[right] > a[largest]) {

largest = right;

}

if (largest != root) {

swap(a[root], a[largest]);

heapify(a, size, largest);

}

}

void heapsort(vector<int> &a, int size) {

for (int i = size / 2 - 1; i >= 0; i--) {

heapify(a, size, i);

}

for (int i = size - 1; i > 0; i--) {

swap(a[0], a[i]);

heapify(a, i, 0);

}

}

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

srand(time(NULL));

int n;

cin >> n;

vector<int> a(n);

vector<int> b(n);

vector<int> c(n);

vector<int> d(n);

vector<int> e(n);

vector<int> f(n);

vector<int> g(n);

vector<int> h(n);

vector<int> l(n);

vector<int> m(n);

for (int i = 0; i < n; i++) {

a[i] = b[i] = c[i] = d[i] = e[i] = f[i] = g[i] = h[i] = l[i] = m[i] = rand() % 100000 + 1;

}

cout << "Исходный массив:\n";

for (int i = 0; i < n; i++) {

cout << a[i] << ' ';

}

cout << '\n';

auto start = chrono::high\_resolution\_clock::now();

bubble\_sort(a, n);

auto end = chrono::high\_resolution\_clock::now();

auto time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Пузырьковая сортировка: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

insert\_sort(b, n);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Сортировка вставками: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

choice\_sort(c, n);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Сортировка выбором: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

quick\_sort(d, 0, n - 1);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Быстрая сортировка: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

twoway\_sorting(e, n);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Двусторонняя сортировка: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

sheyker(f, n);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Шейкерная сортировка: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

comb\_sorting(g, n);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Сортировка расческой: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

Shell\_sorting(h, n);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Сортировка Шелла: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

mergesort(l, 0, n - 1);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Сортировка слиянием: " << time.count() << '\n';

start = chrono::high\_resolution\_clock::now();

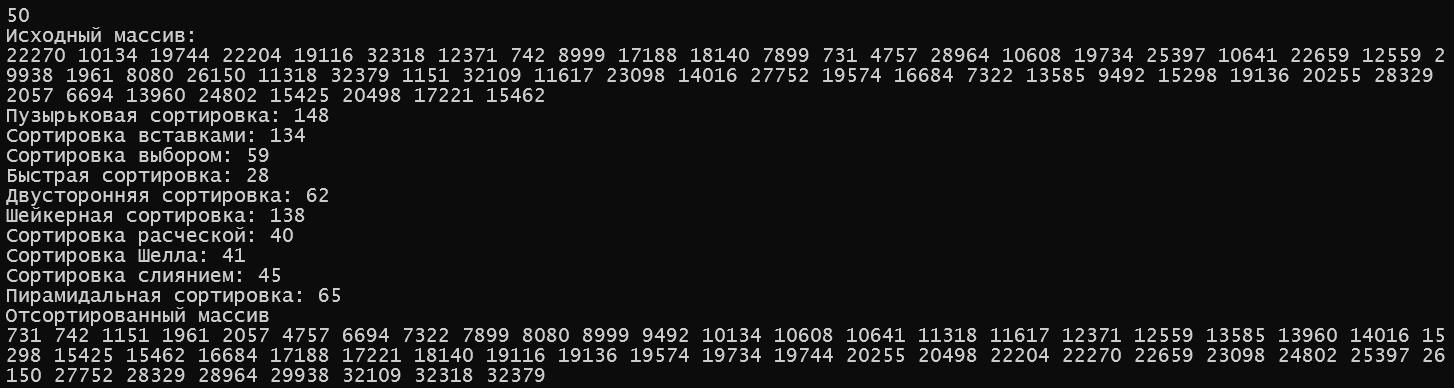
heapsort(m, n);

end = chrono::high\_resolution\_clock::now();

time = chrono::duration\_cast<std::chrono::microseconds>(end - start);

cout << "Пирамидальная сортировка: " << time.count() << '\n';

}

Результат выполнения(время указано в микросекундах):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Сортировка** | **Временная сложность** | | | **Затраты памяти** |
|  | **Лучшее** | **Среднее** | **Худшее** |  |
| **Пузырьковая** | **O(n)** | **O(n^2)** | **O(n^2)** | **O(1)** |
| **Выбором** | **O(n^2)** | **O(n^2)** | **O(n^2)** | **O(n)** |
| **Вставками** | **O(n)** | **O(n^2)** | **O(n^2)** | **O(n)** |
| **Шейкерная** | **O(n)** | **O(n^2)** | **O(n^2)** | **O(1)** |
| **Расческой** | **O(n\*log(n))** | **O(n^2 / 2^p)** | **O(n^2)** | **O(1)** |
| **Шелла** | **O(n\*log(n))** | **O(n\*log(n))** | **O(n^2)** | **O(1)** |
| **Слиянием** | **O(n\*log(n))** | **O(n\*log(n))** | **O(n\*log(n))** | **O(n)** |
| **Быстрая** | **O(n)** | **O(n\*log(n))** | **O(n^2)** | **O(n)** |
| **Пирамидальная** | **O(n\*log(n))** | **O(n\*log(n))** | **O(n\*log(n))** | **O(n) или O(1)** |
| **Двусторонняя** | **O(n^2)** | **O(n^2)** | **O(n^2)** | **O(1)** |

Таблица оценки алгоритмов сортировки:

# **6. Выводы**

Знание оценок сложности, таких как O (большое), o (малое), Θ (тета) и Ω (омега) является важным инструментом в анализе алгоритмов и проектировании программного обеспечения. Они позволяют оценивать и сравнивать эффективность алгоритмов, предсказывать их поведение при увеличении размера входных данных и выбирать наиболее подходящий алгоритм для конкретной задачи. Вот некоторые ключевые причины, почему знание этих оценок сложности важно:

1) **Сравнение алгоритмов**: оценки сложности позволяют нам сравнивать алгоритмы и определять, какой из них будет более эффективным для решения конкретной задачи.

2) **Прогнозирование производительности**: зная оценки сложности, можно предсказать, как алгоритм будет масштабироваться с ростом размера входных данных. Это позволяет оценить, насколько быстро или медленно алгоритм будет работать при различных объемах данных. Такие прогнозы помогают принимать решения о выборе алгоритма и оптимизации программного обеспечения.

3) **Оптимизация и улучшение алгоритмов**: оценки сложности помогают идентифицировать узкие места в алгоритмах и определять, где можно внести улучшения. Если алгоритм имеет высокую сложность, можно искать способы снизить его сложность или найти альтернативные методы решения задачи.

4) **Анализ ресурсов**: оценки сложности также позволяют анализировать затраты ресурсов, таких как время и память, необходимые для выполнения алгоритмов. Это особенно важно при работе с ограниченными ресурсами, такими как мобильные устройства или системы с низкой вычислительной мощностью.

Знание оценок сложности помогает программистам и исследователям принимать осознанные решения при проектировании и разработке программного обеспечения. Оно позволяет оптимизировать алгоритмы, предсказывать их производительность и обеспечивать эффективное использование ресурсов.