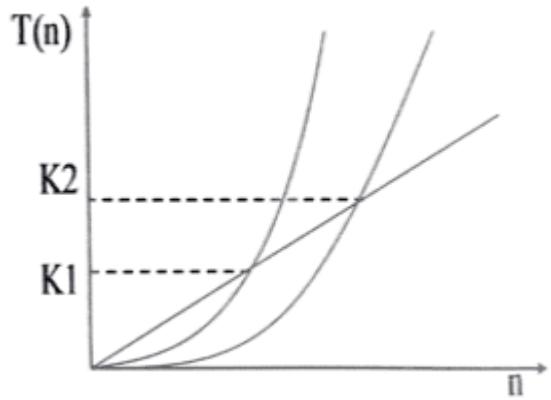
Лекция 4.1. Эффективность алгоритмов (продолжение).

Асимптотика роста

Если асимптотика роста одного алгоритма меньше чем другого, то в большинстве случаев первый алгоритм будет эффективнее для всех входов, кроме самых коротких. При анализе необходимо оценивать сложность вызываемой функции. Если в ней выполняется определённое число инструкций (например, вывод на печать), то на оценку сложности это практически не влияет (на время - очень сильно!!!).



Если же в вызываемой функции выполняется O(N) шагов, то вызов ее в цикле может значительно усложнить алгоритм.

- Если во внутренних циклах одной функции происходит вызов другой функции, то сложности перемножаются.
- Если же основная программавызывает функции по очереди, то их ее сложность будет равна наибольшей из них.

Пример 1

```
void f1() {
    int i,j,k;
    for (i = 1; i < n; i++){
            for ( j = 1; j<n; j++){
                for (k = 1; k < n; k++)
                {
                    //Какое-то действие
                }
            }
```

```
    }
}
void f2()
{
    int i,j;
    for (i = 1; i < n; i++)
        for (j = 1; j <= n; j++)
            f1();
}
void main()
{
    f2();
}
```

Сложность программы: $O(n^2) \cdot O(n^3) = O(n^5)$

Пример 2

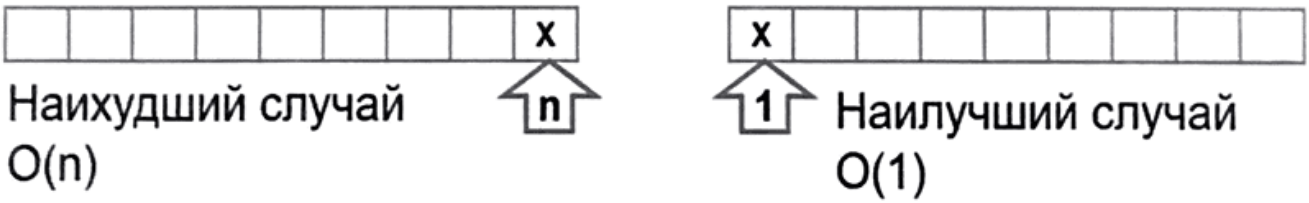
```
void f1() {
    int i,j,k;
    for (i = l; i < n; i++){
            for ( j = l; j<n; j++){
                for (k = l; k < n; k++)
                {
                    //Какое-то действие
                }
            }
    }
}
void f2()
{
    int i,j;
    for (i = 1; i < n; i++)
        for (j = 1; j <= n; j++)
            // Какое-то действие
}
void main()
{
    f1();
    f2();
}
```

Сложность программы: $O(n^2) + O(n^3) = O(n^3)$

> **Примечание**: У Александры Васильевной Силантьевый был свой

смешанный пседокод (смесь Пасаля и Си)

Сложность поиска элемента в векторе



Наилучший случай $O(1)$

- Наступление таких случаев маловероятно.
- Ожидаемый вариант (средний) - это $n/2$ сравнений, чтобы найти требуемый элемент. Значит сложность этого алгоритма в среднем составляет $O(n/2) = O(n)$.

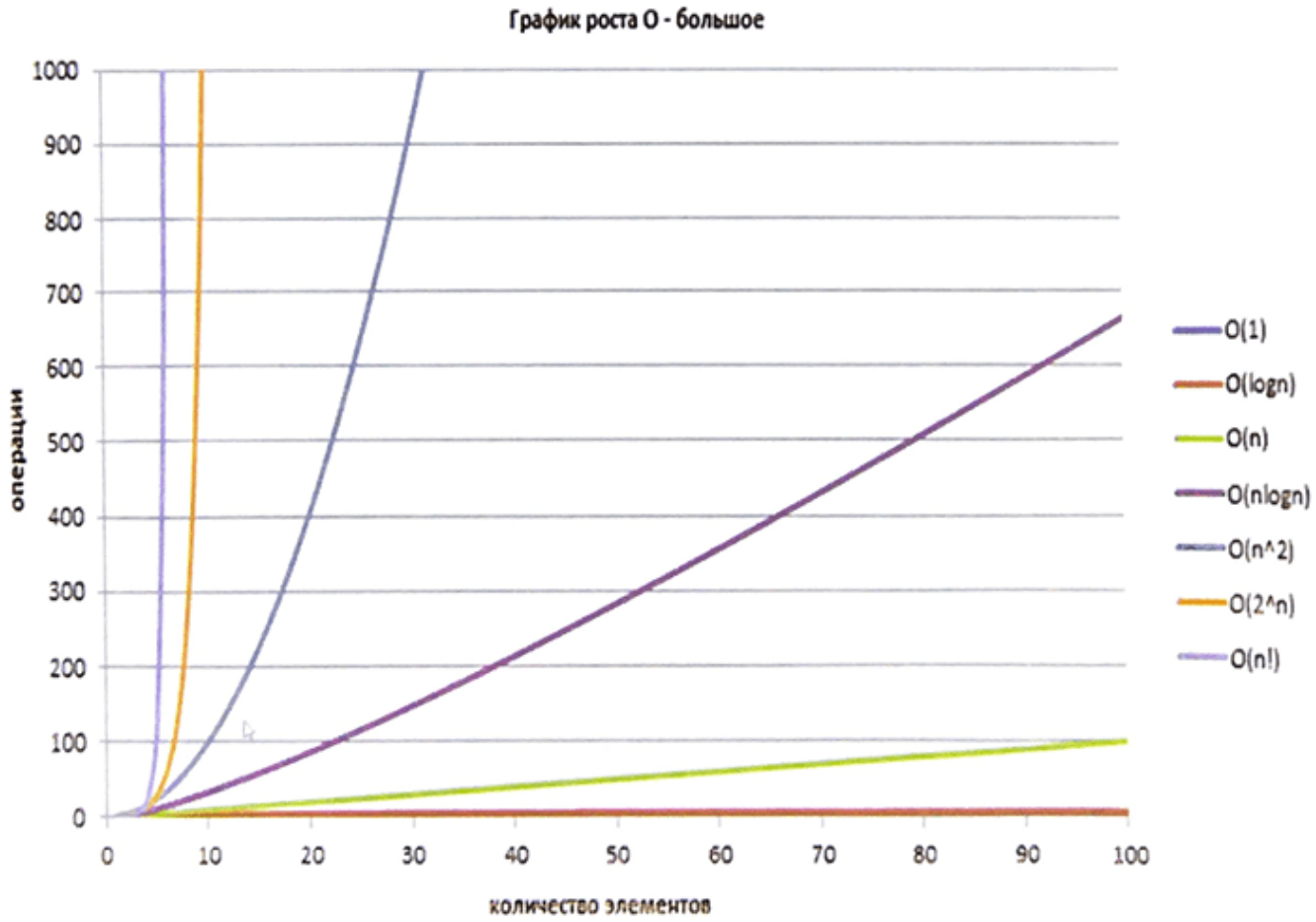
Есть алгортимы, где сложность в наихудшем и ожидаемом случае существенно отличаются. Например, быстрая сортировка:

- в наихудшем случае $O(n^2)$,
- ожидаемый случай - $O(n * \log(n))$, что много быстрее.

О-оценка позволяет разбить осн. ф-ции на ряд групп (классов сложности) в зав-ти от их роста:

- постоянные - типа $O(1)$
- логарифмические - типа $O(\log_2 n)$
- линейные - типа $O(n)$
- линейно-логарифмические - типа $O(n * \log_2 n)$
- квадратичные - типа $O(n^2)$
- степенные типа $O(n^{\mathrm{a}})$ при $\mathrm{a} > 2$
- Показательные или экспоненциальные - $0(2^n)$
- Факториальные - $O(n!)$

График роста О-большого



Скорость алгоритмов на ПК с быстродействием 1 млн оп в секунду

сложность	N=10	N=30	N=50
N^3	0.001с	0.027с	0.12с
2^n	0.001с	17.9 мин	35.7 лет
3^n	0.059с	6.53 лет	2.28 *1010 лет
$N!$	3.63с	8.41*10 18 лет	9.64 *1050 лет

Примеры сложности алгоритмов
Сложность хорошо известных алгоритмов:

1. Последовательный поиск: $O(n)$
2. Бинарный поиск: $O(\log_a n)$

3. Пузырьковая сортировка: $O(n^2)$
4. Сортировка слиянием (объединением): $\mathrm{O}(n \cdot \log_a n)$

O обозначения (Ландау - верхняя оценка)

Ω нижняя оценка

Θ обозначение (верхняя и нижняя оценка) (Кнут)

Достаточно часто в литературе Θ-оценки и O- оценки считаются идентичными.

Правила определения сложности

- Постоянные множители не имеют значения для определения порядка сложности: $O(kf) = O(f)$;
- Порядок сложности произведения двух функций равен произведению их сложностей (вложенные алгоритмы): $O(f * g) = O(f) * O(g)$;
- Порядок сложности суммы функций равен максимальному порядку из слагаемых (последовательные): $O(N^5 + N^3 + N) = O(N^5)$

Правила асимптотического анализа

1. $O(k * f) = O(f)$ - постоянный множитель к (константа) отбрасывается: $O(9,1n) = O(n)$
2. $O(f * g) = O(f) * O(g)$ - сложность произведения равна произведению сложностей: $O(5n * n) = O(5n) * O(n) = O(n) * O(n) = O(n * n) = O(n^2)$
3. $O(f + g)$ равна доминанте $O(f)$ и $O(g)$ - сложность суммы равна сложности доминанты первого и второго слагаемых: $\mathrm{O}(n^5 + n^{10}) = \mathrm{O}(n^{10})$

Подсчет количества операций - не обязательно. Исходя из правил, достаточно знать какой сложностью обладает та или иная конструкция алгоритма.

Список функционального доминирования

- Если N - переменная,
  - a, b, т, к - константы, то:
- N^N доминирует (является определяющей) над $N!$;
- N! доминирует над a^N;
- a^N доминирует над b^N, если a $>$ Ь;

- a^N доминирует над N^K, если a > 0;
- N^k доминирует над N^m, если $k > m$;
- N доминирует над $\log_a(N)$, если a > 1.

Сортировка вставками.

```
void sort(int* a, int n)
{
    for(int i=1;t<n;i++)
    {
        for(int j=i; j>0 && a[j-1]>a[j];j--)
        {
            int tmp=a[j-1];
            a[j-1]=a[j];
            a[j]=tmp;
        }
    }
}
```

*нумерация элементов массива начинается с 0.

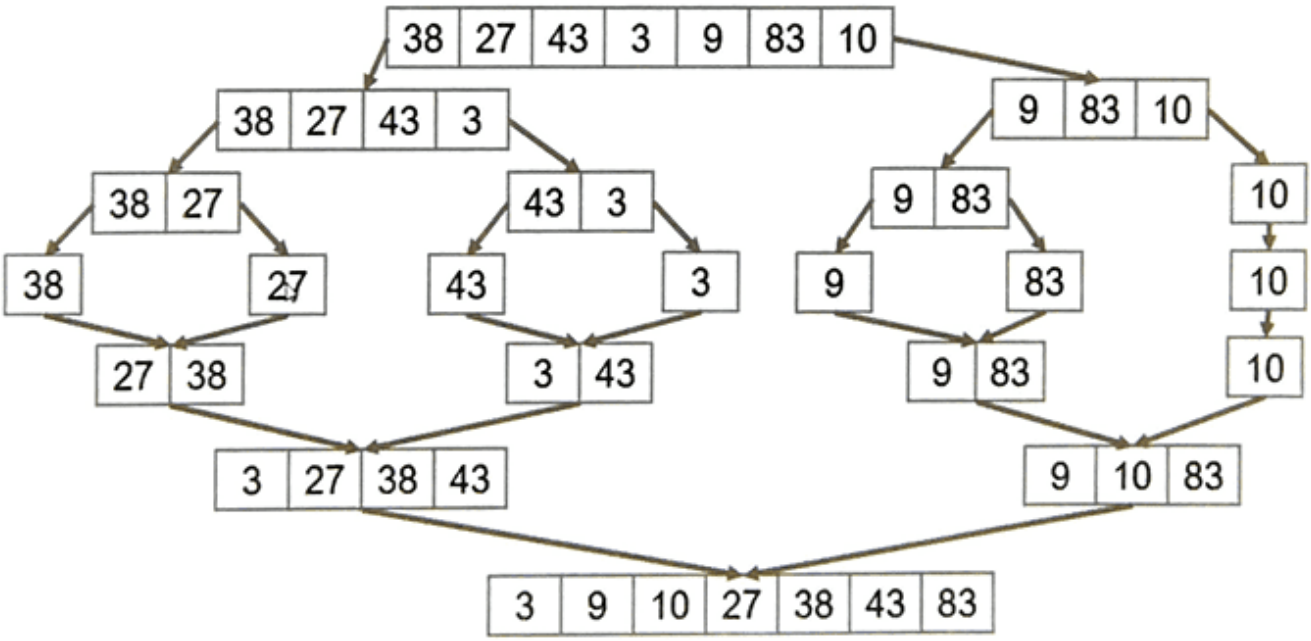
Алгоритм сортировки вставками массива A.

(Обозначения (часто используемое для описания алгоритмов)
`<-` присваивание, отступы - вложенность
операторы - как в Паскале)

```
for j <- 2 to length (A)
    do key <- A[j]
        i <- j - 1
        while (i > 0) and (A[i] > key)
            do A[i + 1] <-A[i]
                i <- i - 1
            A [i + 1] <- key
```

Алгоритм оценивается как $C * n^2$.

Сортировка слиянием. $O(n \log n)$



Функция Merge сливает два массива `a[p..q]` и `a[q+1..r]` будет описана как `void merge(int *a, int p, int q, int r)`

Процедура сортировки слиянием

`merge_Sort (int *a, int p, int r)`, сортирует участок массива `a[p..r]`, не меняя остальную часть массива

При $p >= r$ участок содержит 1 элемент и, он отсортирован. Иначе ищем число q, делящее участок на примерно равные части:

```
void merge_Sort(int *a, int p, int r) {
    if (p < r) {
        q = (p + r) / 2;
        merge_sort (a,p,q);
        merge_sort (a,q+1,r);
        merge (a,p,q,r);
    }
}
```

Алгоритм сортировки слиянием

(Часто используемое описание алгоритмов:)

//Отсортированная последовательность:
1 2 3 4 5 8 9

```
Merge_Sort (A,p,г)
    if p < r  //3-й шаг: 1 4 5 8  2 3 9
    then q = (p + г) / 2 //2-й шаг: 5 8  1 4  3 9  2
        Merge_Sort (A,p,q)
        Merge_Sort (A,q+1,r)
        Merge (A,p,q,r)
```

Оценка -$O(n \log n)$

Разница в применении 2-х алгоритмов

Сортируем массив с n = 1 000 000 чисел = 10^6.

У нас имеются компьютеры с быстродействием: в 10^{10}оп/сек и в 10^8оп/сек

Сорт, вставками написана на Ассемблере, сложность $2n^2$.

Сорт, слиянием сложность $50n \log n$

Тогда для сортировки вставками получим:

$(2 * (10^6)^2 \text{оп})/(10^{10})\text{оп/сек} = \ 200\text{сек}$

В то время как для сортировки слиянием имеем: $(50 * (10^6) * \log(10^6)\text{оп})/(10^8)\text{оп/сек} = -10\text{сек}$

Сложность рекурсивных алгоритмов

- Простая рекурсия
  - Сложность алгоритмов зависит от сложности внутренних циклов и от количества итераций рекурсии.
  - Рассмотрим рекурсивную реализацию вычисления факториала (нотация паскаля):

```
Function Factorial (n: Word): integer;
    begin
    if n > 1
    then Factorial <- n*Factorial(n-1)
    else Factorial <- 1;
    end;
```

Зта процедура выполняется N раз, т. о., вычислительная сложность этого алгоритма

равна $O(N)$.
Сложность рекурсивных алгоритмов

Многократная рекурсия

Рекурсивный алгоритм, вызывающий себя несколько раз, называется многократной рекурсией, они могут сделать алгоритм гораздо сложнее. Например (нотация C):

```
void DoubleRecursive(int N) {
    if (N>0) {
        DoubleRecursive(N-1);
        DoubleRecursive(N-1);
    }
}
```

Предположение - рабочий цикл будет равен $O(2N) = O(N)$.
На самом деле - сложнее.
Сложность равна $O(2^{(N+1)} - 1) = O(2^N)$., НЕ $O(2N)$!!!

Всегда надо помнить, что анализ сложности рекурсивных алгоритмов весьма нетривиальная задача.

Например, вычисление чисел Фибоначчи

`FIB(n+1) = FIB(n) + FIB(n-1)` для n > 0 и `FIB(1) = 1, FIB(0) = 0`

приводит к следующей рекурсивной подпрограмме:
(нотация паскаля)

```
Function FIB (N : Integer): Longint;
    Begin
        If N<2
        Then FIB <- N
        Else FIB <- FIB(N -1) + FIB(N - 2);
End;
```

Временная сложность - $O(2^n)$

Объёмная сложность рекурсивных

- При каждом вызове процедура запрашивает небольшой объём памяти, но этот объём может значительно увеличиваться в процессе рекурсивных вызовов. По

этой причине всегда необходимо проводить хотя бы поверхностный анализ объёмной сложности рекурсивных процедур, оценивая средний и наихудший случай

- Рекурсивные алгоритмы более затратны с точки зрения времени и памяти, чем итерационные , кроме того на их сложность влияет организация рекурсии

Резюме

- Анализ производительности алгоритмов позволяет сравнить разные алгоритмы. Он помогает оценить поведение алгоритмов при различных условиях. Выделяя только части алгоритма, которые вносят наибольший вклад во время исполнения программы, анализ помогает определить, доработка каких участков кода позволяет внести максимальный вклад в улучшение производительности.
- Учитывать, что один алгоритм м. б. быстрее, но за счет использования большого объема памяти. Другой алгоритм, использующий коллекции, может быть более медленным, но зато его проще разрабатывать и поддерживать.
- После анализа доступных алгоритмов, понимания того, как они ведут себя в различных условиях и их требований к ресурсам, вы можете выбрать оптимальный алгоритм для вашей задачи.