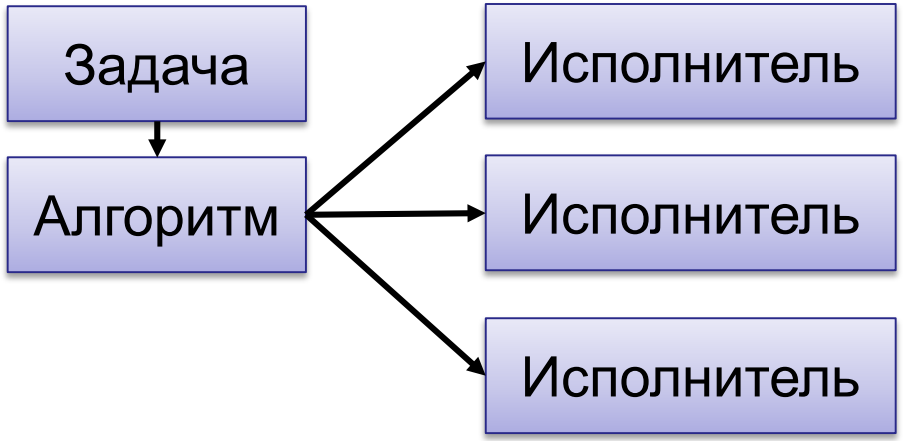
Эффективность алгоритмов

- **Анализ любого алгоритма должен дать четкое представление : 1) о емкостной 2) о временной сложности процесса.**
- Емкостная сложность **(пространственная) (space efficiency)** - это размер памяти, в которой предстоит размещать все данные, участвующие в вычислительном процессе (входные наборы данных, промежуточная и выходная информация).
- Возможно, не все перечисленные наборы требуют одновременного хранения, – значит, удается сэкономить.
- В ряде случаев, оценка емкостной сложности становится менее очевидной, например при использовании динамических структур.
- Временная сложность **(time efficiency) -** время работы программы
- Проведем анализ временной трудоемкости.
- Поставлена некоторая задача, и для ее решения спроектирован алгоритм. Он описывает вычислительный процесс, который завершается за конечное число действий-шагов..

Анализ временной трудоемкости



Реальное время выполнения каждого шага алгоритма зависит от конкретного вычислительного устройства.

Т.е., неотъемлемым участником вычислительного процесса, – не алгоритма! – является *исполнитель*.

Имея ввиду предполагаемого исполнителя, лучше заранее оценить его вычислительные способности.

На эффективность алгоритмов влияет:

- производительность вычислительной системы (набор *элементарных*, инструкций системы)
- представление самих данных
- язык программирования (набор *элементарных* инструкций языка).

Возможны следующие варианты:

- либо алгоритм *явно* предписывает выполнять арифметико-логические операции, – и такой уровень программирования рассчитан непосредственно на работу процессора
- либо используются «укрупненные» инструкции, и для их обработки применяется специальный язык.

При оценке быстродействия алгоритма учитывают:

- Поведение вычислительного процесса в «среднем»
- Работа в «экстремальных» условиях

Моделирование «худших» случаев всегда связано с содержанием самого алгоритма. Возможные варианты:

- Проверка поведения алгоритма на входных данных, принимающих граничные значения из разрешенного диапазона
- Тестирование алгоритма на максимально больших по объему входных наборах (что важно для анализа как временной, так и емкостной эффективности).

Умение предвидеть «нехорошие» ситуации отличает квалифицированного алгоритмиста от обыкновенного кодировщика.

В связи с расширением сферы производства программного обеспечения сформировалась самостоятельная специализация – «тестеры программ».

Эффективность алгоритма оценивается:

- **временем его работы, или временной сложностью (time efficiency)**
- и **объемом памяти**, требуемой для его выполнения, или **емкостной (пространственной) (space efficiency) сложностью**.

Время выполнения зависит

- от **порядка следования элементов** и часто определяется **размером входа**.
- **Размером входа может быть:**
- число элементов на входе (сортировка, преобразование Фурье);
- количество байт памяти для представления данных;

При анализе алгоритмов интерес представляет **максимальный** размер входа.

Порядком некоторой функции F(n)

- (при достаточно больших **n)** называют **другую функцию G(n), такую, что:**
- $\lim_{n\to\infty} \frac{F(n)}{G(n)} = const \neq 0$
- Обозначение: F(n) = **O**[G(n)]
- (**O-**нотация)
- Например, $F(n) = 2n^4 - 3n^3 + 4n^2 - 5$
- порядком является $\mathbf{n^4}$ или
- $F(n) = O(n^4)$,

Время работы

- оценивается **числом элементарных шагов (операций) для каждой строки алгоритма.**
- При **вызове** подпрограмм оценивается время их **исполнения.**
- Т. о., для каждого конкретного алгоритма мы можем приблизительно **подсчитать** время его выполнения.

Для C++ на Pentium

№	Операторы C++	Обоз-наче-ния	Количество тактов
1	a+b, a>b, a>>b и т.д.	t_+	2
2	a*b	t_*	20
3	a/b	$t_/$	28
4	t++, a+, a>const и т.д.+	t_{++}	1
5	*p	t_p	1
6	a[]	$t_{[]}$	2
7	if (…) P_1 else P_2	t_{if}	$t_{усл}+1+ P_1 t_{ветвь1}+P_2 t_{ветвь2}$
8	for () // n раз	t_{for}	$t_{усл}+t_{пров.}+2+n(t_{тела}+t_{модиф}.+ t_{пров}.+1)$

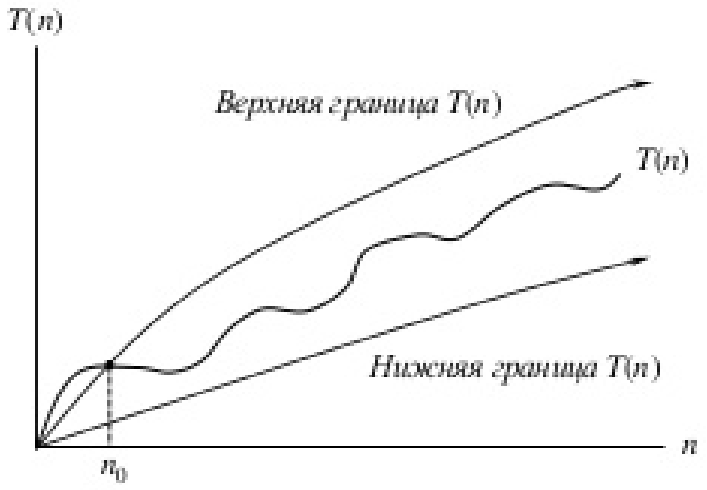
Микроконтроллеры AVR семейства Tiny и Mega фирмы ‘ATMEL’

№	Команда	Кол-во машинных циклов (тактов)
1	Условный переход	Результат True - 2 или 3 Результат False - 1
2	Безусловный переход	2
3	Прерывание	Max 4
4	Вызов подпрограмм	3 (2 из кот. –сохран. в стеке 2 байта сч. команд)
5	Возврат из п/программ и обработки прерываний	4
6	Логические операции	1
7	Арифметические операции и сдвиг	1
8	Умножение и обработка 2-х байтовых значений	2
9	Сложение с константой	2

- Если алгоритм не содержит вложенных циклов, то функция времени - будет какая-то линейная функция от **n**: **T (n) = (n),** где n – размер входа и тип цикла не влияет на сложность алгоритма.
- Если алгоритм использует вложенные циклы, то это будет квадратичная функция **$T(n^2) = (n^2)$,**
- Т. е., вложенность повторений является основным фактором усложнения алгоритмов.
- Рекурсивность алгоритма обычно более затратна по времени и памяти, чем вложенные циклы.
- *В общем случае, анализируя алгоритмы, можно не подсчитывать общее количество выполняемых операций, а проанализировать применяемые управляющие структуры и оценить* ***асимптотику роста*** *времени работы алгоритма при стремлении* ***n*** *в бесконечность.*

Асимптотические обозначения

- Как правило, функция времени $T(n)$ выполнения алгоритма имеет большое количество локальных экстремумов – неровный график с выпуклостями и впадинами
- Проще работать с верхней и нижней оценками (границами) времени выполнения алгоритма

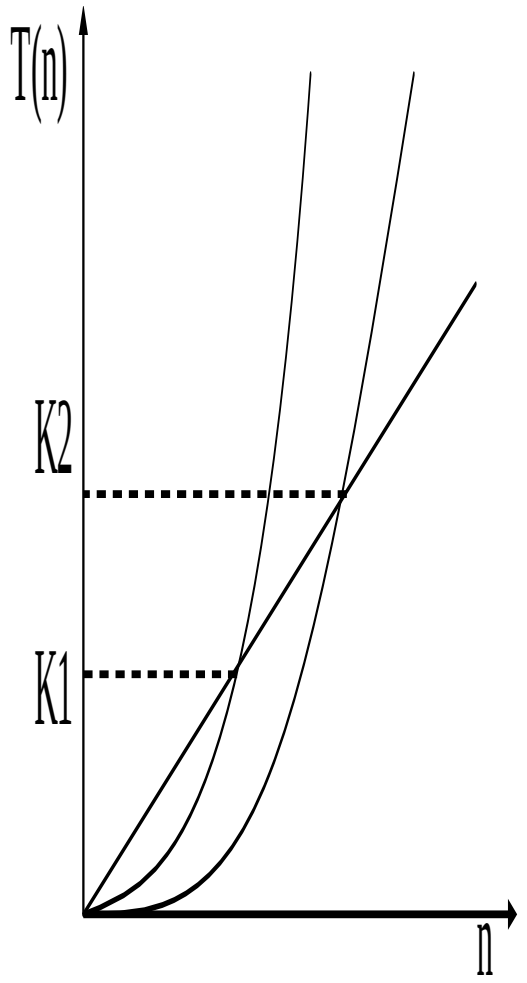


- В теории вычислительной сложности алгоритмов для указания границ функции используют асимптотические обозначения:

- **O (O большое)**
- Функция f(n) ограничена сверху функцией g(n) (порядком ф-ции f(n)) с точностью до постоянного множителя
- **Ω (омега большое)**
- Функция f(n) ограничена снизу функцией g(n) с точностью до постоянного множителя
- **Θ(тета большое**)
- Функция f(n) ограничена снизу и сверху функцией g(n) с точностью до постоянного множителя

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 оценить сложность вызываемой функции.

Если в ней выполняется определённое число инструкций (например, вывод на печать), то на оценку сложности это практически не влияет.

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

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

```
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}
```

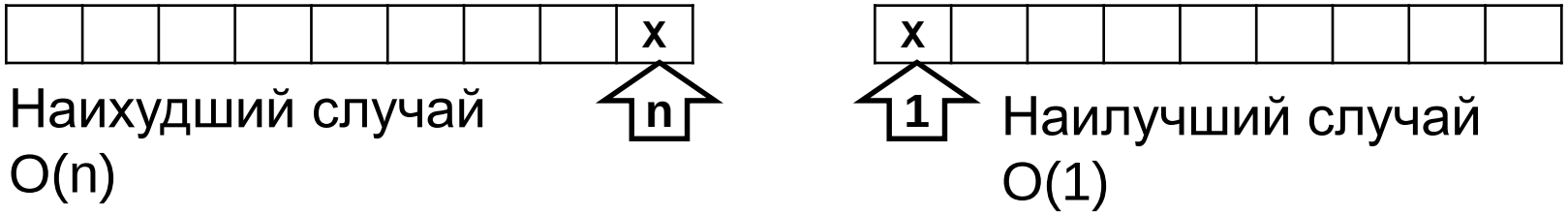
$O(n^2)*O(n^3)=O(n^6)$.

```
void f2() {
int i,j;
for(i:=1;i<n;i++){
  for(j:=1;j<n;j++){
    какое-то действие
}}}
```

```
void main()
 {f1;
  f2;}
```

$O(n^2)+O(n^3)=O(n^3)$.

Сложность поиска элемента в векторе



- 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).

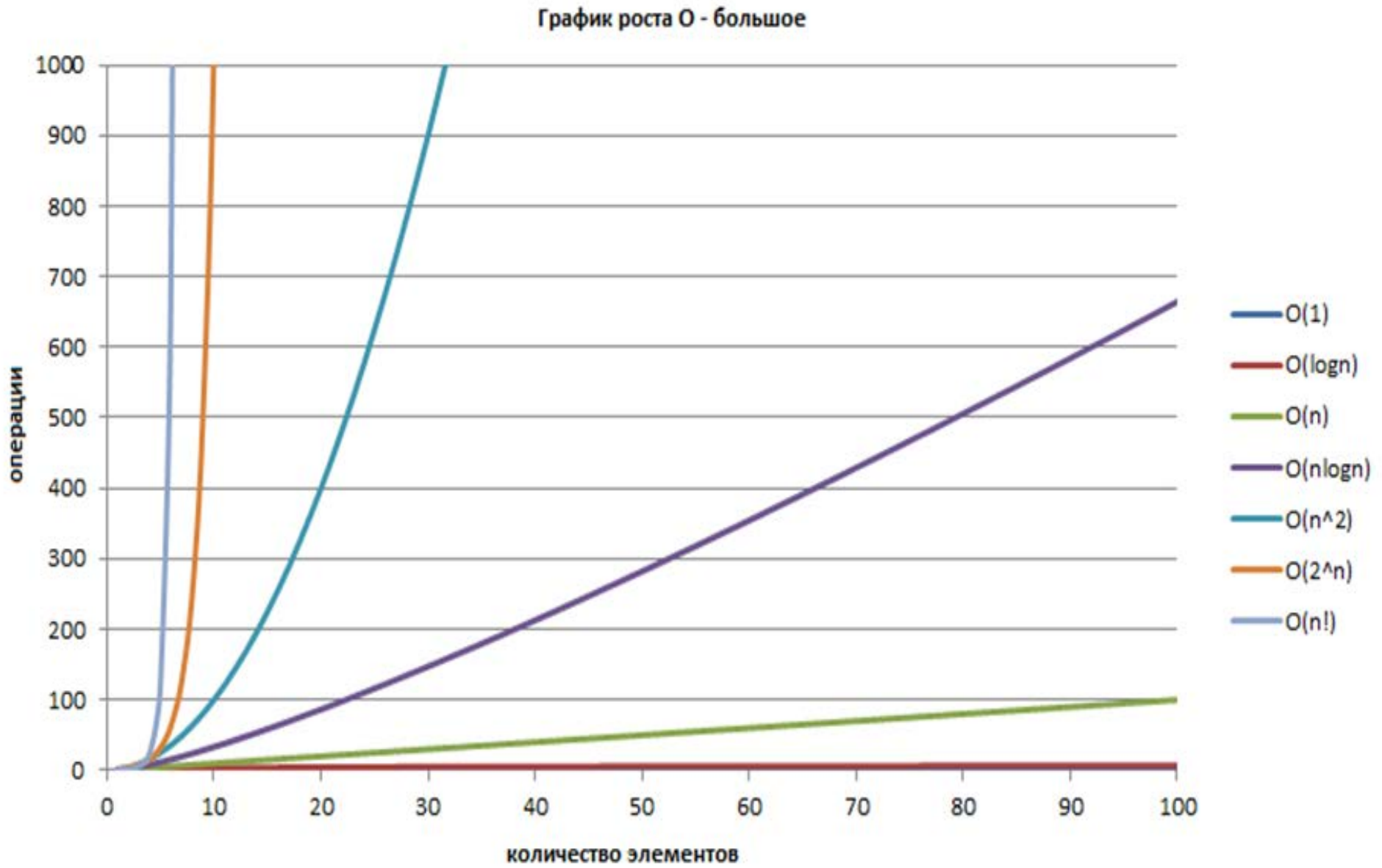
Есть алгортимы, где сложность в наихудшем и ожидаемом случае существенно отличаются. Например, быстрая сортировка:

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

- **O-**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 **$O(n^a)$** при a>2
- Показательные или экспоненциальные - **$O(2^n)$**
- Факториальные - **O(n!)**

График роста О-большого



Скорость алгоритмов на ПК с быстродействием 1млн 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	$2.28 * 10^{10}$ лет
N!	3.63с	$8.41 * 10^{18}$ лет	$9.64 * 10^{50}$ лет

Примеры сложности алгоритмов

Сложность хорошо известных алгоритмов:

1. Последовательный поиск: **0(n)**
2. Бинарный поиск: **0 ($\log_a$ n)**
3. Пузырьковая сортировка: **0 (n^2)**
4. Сортировка слиянием (объединением):

   0 (n $\log_a$ 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:

$$\theta(kf) = \theta(f);$$

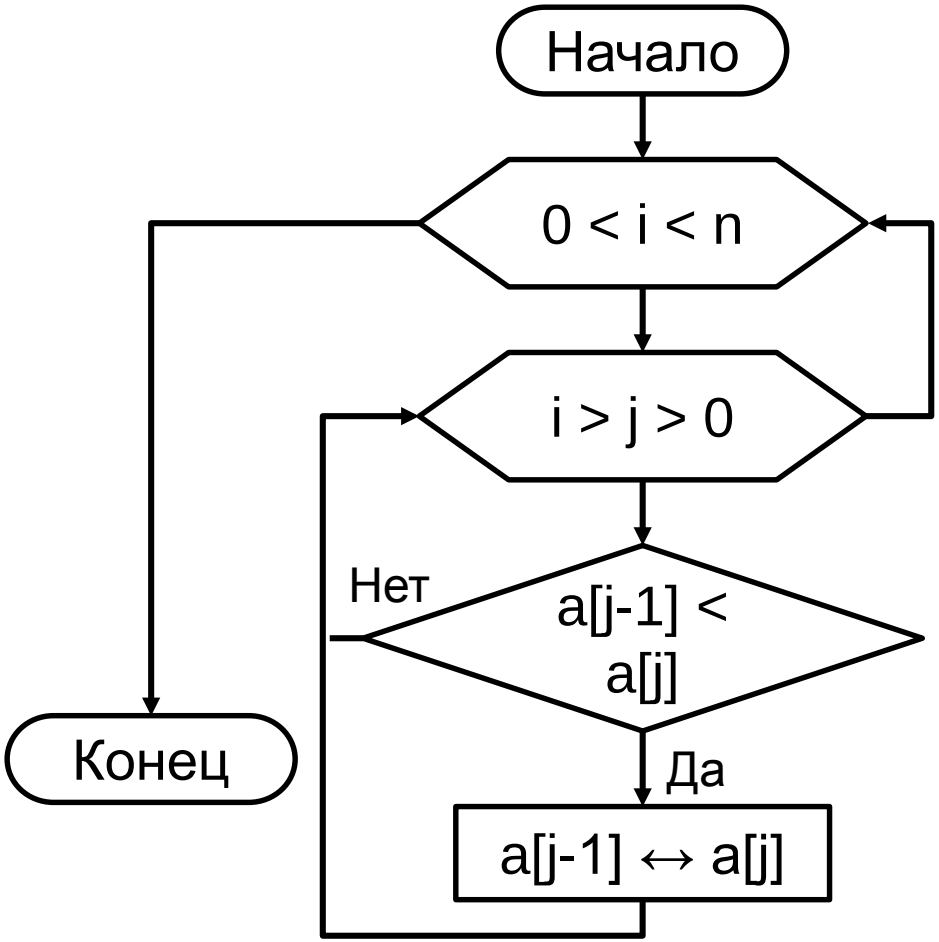
- Порядок сложности произведения двух функций равен произведению их сложностей (вложенные алгоритмы):
  - $\theta(f * g) = \theta(f) * \theta(g);$
- Порядок сложности суммы функций равен максимальному порядку из слагаемых (последовательные):
  - $\theta(N^5 + N^3 + N) = \theta(N^5)$

список функционального доминирования

- Если **N** – переменная,
- а **a, b, m, k** – константы, то:
- **N** ^{N} доминирует (является определяющей) над **N!;**
- **N!** доминирует над $\mathbf{a^N}$;
- $\mathbf{a^N}$ доминирует над $\mathbf{b^N}$, если **a > b**;
- $\mathbf{a^N}$ доминирует над $\mathbf{N^K}$, если **a > 0**;
- $\mathbf{N^k}$ доминирует над $\mathbf{N^m}$, если **k > m;**
- **N** доминирует над $\mathbf{\log_a(N)}$, если **a > 1**.

Сортировка вставками. $O(c*n^2)$

```
void sort(int* a, int n)
{
   for(int i=1;i<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;
      }
   }
}
```

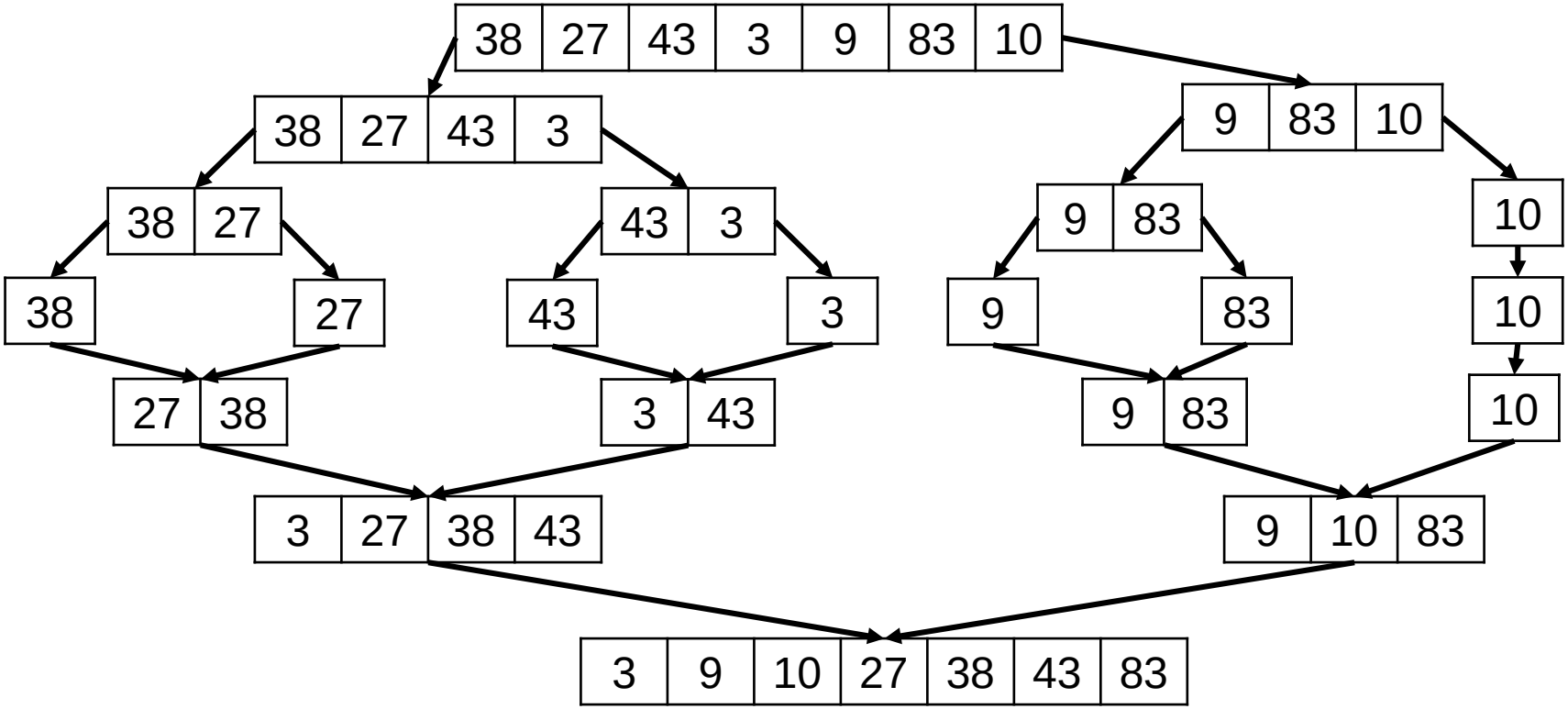


**нумерация элементов массива начинается с 0*

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,r)**
- **if p < r** //3-й шаг: 1 4 5 8 2 3 9
- **then q = (p + r)/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)**
- // исх: 8 5 4 1 9 3 2

Оценка -**(n log n)**

Разница в применении 2-х алгоритмов

- Сортируем массив с n = 1 000 000 чисел = $\mathbf{10^6}$.
- У нас имеются компьютеры с быстродействием:
- в $\mathbf{10^{10}}$ оп/сек и в $\mathbf{10^8}$ оп/сек
- Сорт. **вставками** написана на Ассемблере, сложность $\mathbf{2n^2}$.
- Сорт. **слиянием** сложность **50 n log n**
- Тогда для сортировки **вставками** получим:

- $\mathbf{(2 * (10^6)^2}$ оп$\mathbf{)\ /\ (10^{10})}$ оп/сек = ~ 200 сек
-
- В то время как для сортировки **слиянием** имеем:
- $\mathbf{(50 * (10^6) * \log(10^6)}$ оп$\mathbf{)\ /\ (10^8)}$ оп/сек = ~ 10 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;**

- Эта 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 алгоритмов

- 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.