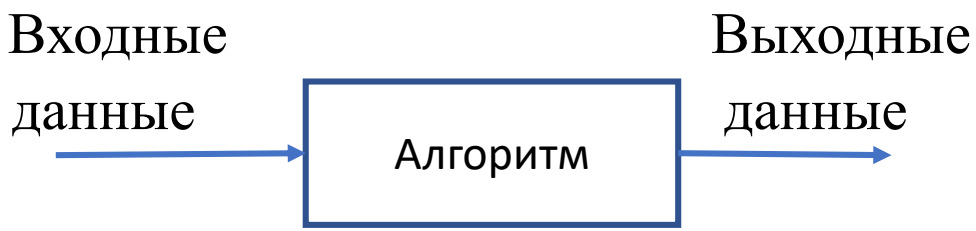
Алгоритмы как технологии

Задача -> Алгоритм->Программа



Алгоритм: определение

Неформально, ***алгоритм*** - конечная совокупность точно заданных правил решения некоторого класса задач или набор инструкций, описывающих порядок действий исполнителя для решения определённой задачи (Wikipedia).

Или: это конечная последовательность команд по преобразованию входных данных задачи в результат задачи. Алгоритм представляет собой последовательность вычислительных шагов, преобразующих входные величины в выходные.

Алгоритм также можно рассматривать как инструмент, предназначенный для решения корректно поставленной *вычислительной задачи*. В постановке задачи в общих чертах задаются отношения между входом и выходом.

Например, сортировка последовательности чисел в неубывающем порядке. Если на вход подается последовательность (31,41,59,26,41,58), то результат алгоритма сортировки должен быть таким: (26,31,41,41,58,59).

Алгоритм может быть задан на естественном языке, в виде компьютерной программы или даже воплощен в аппаратном обеспечении.

Единственное требование - его спецификация должна предоставлять точное описание вычислительной процедуры, которую требуется выполнить.

Свойства алгоритма

Дискретность: алгоритм – это последователь, состоящая из **конечного** числа команд.

Понятность: алгоритм должен быть записан на языке, понятном исполнителю.

Детерминированность (определенность): алгоритм должен **четко** и **полно** описывать вычислительный процесс, не должен иметь двусмысленности.

Результативность: формирование результата **на допустимых входных данных** (если задача имеет решение) и вывод сообщения, если на введенных данных результат получить нельзя.

Массовость: применение алгоритма к большому числу вариантов входных данных.

Свойства алгоритма (продолжение)

Корректность. Разработанный алгоритм, для всех исходных данных, для которых его задача может быть решена, позволяет получить правильный результат и *ни* для каких *исходных* данных не дает неправильного результата.

Алгоритм корректен, если для каждого ввода результатом его работы является корректный вывод.

Если алгоритм некорректный, то для некоторых вводов он может вообще не завершить свою работу или выдать ответ, отличный от ожидаемого.

Свойства алгоритма (эффективность)

Если бы компьютеры были неограниченно быстрыми, подошел бы любой корректный метод решения задачи.

Сегодня есть весьма производительные компьютеры, но их быстродействие не может быть бесконечно большим.

Память дешевеет, но она не может быть бесплатной.

Таким образом, время вычисления – это такой же ограниченный ресурс, как и объем необходимой памяти.

Этими ресурсами следует распоряжаться разумно, чему и способствует применение алгоритмов, эффективных в плане расходов *времени* и *памяти*.

Эффективность означает:

- что все операции, которые необходимо произвести в алгоритме, должны быть достаточно простыми
- что описание логики работы алгоритма понятно
- что операции алгоритма можно было выполнить точно за то время, которым мы располагаем и используя тот объем памяти, которым мы располагаем.

Свойства алгоритма (эффективность, продолжение)

Эффективность — это свойство алгоритма, которое связано с вычислительными ресурсами, используемыми алгоритмом: **время** и требующаяся для хранения данных **память.**

Алгоритм должен быть проанализирован с целью определения необходимых алгоритму ресурсов.

При разработке алгоритма, для достижения максимальной эффективности, требуется стремиться *к уменьшению используемого объема памяти и времени выполнения алгоритма*.

Алгоритмы как технология

- **Разработка и запись алгоритма**
- **Доказательство корректности алгоритма**
- **Анализ алгоритма**
  - **определение вычислительной сложности алгоритма (производительность по времени)**
  - **Определение емкостной сложности алгоритма (объем требующейся памяти)**

Доказательство корректности алгоритма

Методы

- Математическая индукция
- Инвариант цикла

Универсальным методом доказательства корректности алгоритма считается метод математической индукции.

Метод математической индукции

Допустим у нас есть некоторое утверждение P. Для того, чтобы доказать его истинность для всех натуральных чисел n нужно проделать несколько шагов:

Процесс доказательства

Гипотеза индукции : определите правило, которое мы хотим доказать для каждого n , назовем его P(n)

Шаг 1. Доказать, что P истинно в самом простом случае - для n = 1. Это называется **базой индукции**.

Шаг 2. Далее мы **предполагаем**, что утверждение верно для некоторого k и доказываем, что если оно верно для k, то верно и для k + 1. Это называется **шагом индукции** или индукционным переходом.

Если мы это доказываем, тогда можем сделать вывод о том, что утверждение P справедливо для всех n.

Пример:
Доказать методом индукции, что 1 + 2 + 3 + ... + n = n (n + 1)/2

Шаг : n=1, тогда¶ 1 = 1*(1+1)/2 - верно

Шаг 2: Предполагая, что верно для случая k=n-1¶: 1+2+3+...+(n-1) = (n-1)n/2¶

Используйте гипотетические выводы для

1+2+3...+(n-1)+n = (n-1)n/2+n = n(n+1)/2

Индуктивные функции, предикат, инвариант

Индуктивные функции

Пусть имеется множество M и функция f (x1, x2, . . . xn) где x1, x2, . . . xn ϵM, а значения функции— элементы множества N.

Тогда, если значение функции f на последовательности x1, x2, . . . x**n** можно определить по её значению на последовательности x1, x2, . . . , x**n−1** и элементе x**n**, то такая функция называется индуктивной.

Например. Найти наибольшее значение из всех элементов последовательности, то функция maximum (поиска максимального — индуктивна), так как **maximum(x1, x2, . . . , xn) = max(maximum(x1, x2, . . . , xn−1), xn).**

Предикат — логическое утверждение, содержащее переменную величину.

Инвариант — предикат, сохраняющий своё значение после исполнения заданных шагов алгоритма.

Инвариант цикла

Инвариант цикла – предикат (условие, логическое выражение), который должен обязательно обладать тремя свойствами:

Инициализация. Он истинен при первой инициализации цикла.

***Сохранение*.** Если он истинен **перед очередной итерацией** цикла, то **он истинен и после итерации**.

Завершение. После завершения цикла инвариант также остается истинным. Это свойство позволяет убедиться в правильности алгоритма.

И если такое условие (предикат) будет найдено, то можно сделать вывод о правильности разрабатываемого алгоритма или фрагмента алгоритма.

Инвариант цикла должен быть связан **с переменными**, которые изменяются в теле цикла.

Примеры инварианта цикла

Пример1. Определения инварианта цикла

```
j ←9
for i←0 to i<10
do
  j←j-1;
Od
```

Инвариант: для каждой итерации, i + j = 9.

Инициализация переменных и цикла: i=0, j=9, i+j=9 – истинно.

Сохранение: при изменении i и j в теле цикла и завершении первой итерации i=1, j=8, i+j=9 и для любого i>=0 и i<10 и j>0 и j<=9 i+j=9

Слабый инвариант, i >= 0 && i <= 10.

Завершение:

Пример 2. Определить инвариант алгоритма поиска наименьшего значения в массиве a[n].

Алгоритм

```
Min←A[1]
for i←2 to n do
  if A[i] < Min then
      Min←A[i]
  endif
od
```

Инвариант формулируется так: *для i<=n в переменную Min записан минимальный из* ***первых i элементов [1,i) т.е. a[1], f[2], … a[i-1]***

Инициализация: *i=1, Min из области [1] элемента*

Сохранение: *при шаге i<=n, в Min записано минимальное из a[1], f[2], … a[i-1]*

Завершение: после завершения цикла i=n+1.

Подставим в инвариант: *в переменную Min записан минимальный из* ***первых i элементов [1,n+1) т.е. a[1], f[2], … a[n]***

Подход к составлению циклов и определению инварианта

При составлении циклов может оказаться удобным понятие ***области неопределенности,*** используемое в вычислительных методах математики.

Рассмотрим подход на алгоритме поиска минимального в массиве.

Область изменения параметров задачи (в нашем примере [1,n)) можно разделить на две части:

- ***исследованную область*** (для которой найден Min: [1,i))
- ***область неопределенности*** ([i,n)) – еще не рассмотренная часть данных.

Необходимо составлять цикл так, чтобы на каждой итерации область неопределенности сокращалась.

```
Min←a[1]
for i←1 to n do
  if a[i] < Min
then
      Min←a[i]
endif
od
```

Описание примера

В начале первой итерации исследованная область представляла собой единственную точку 1, а область неопределенности составляла нерассмотренную часть массива [1,n). На втором шаге область неопределенности сократилась до [2,n), на третьем -- до [3,n) и т. д., пока, наконец, не превратилась в пустое множество, не содержащее исследуемых значений.

Порядок доказательства корректности цикла с помощью инварианта

1. Доказывается, что выражение инварианта **истинно перед началом цикла**.
2. Доказывается, что **выражение инварианта сохраняет свою истинность после выполнения тела цикла на текущей итерации.**
3. Доказывается, что **при истинности инварианта после завершения цикла переменные примут именно те значения, которые требуется получить** (это элементарно определяется из **выражения** инварианта и известных конечных значениях переменных, на которых основывается условие завершения цикла).
4. Доказывается (возможно — без применения инварианта), что цикл завершится, то есть условие завершения рано или поздно будет **выполнено**. Истинность утверждений, доказанных на предыдущих этапах, однозначно свидетельствует о том, что цикл выполнится за конечное время и даст желаемый результат.

Инварианты используют при проектировании и оптимизации циклических алгоритмов. Например, чтобы убедиться, что оптимизированный цикл остался корректным, достаточно доказать, что инвариант цикла не нарушен и условие завершения цикла достижимо.

Пример определения инварианта цикла при обработке массива

Поиск индекса заданного значения (key) в массиве a[N].
Рассмотрим код и определим инвариант.

LINEAR-SEARCH(A, key)

```
1  for i = 1 to N
2      if A[i] == key
3          return i
4  return -1
```

Что справедливо до начала цикла, после каждой итерации и в конце?

Инвариант: *для всех k ∈ [1, i) A[k] ≠ key.*

Инициализация:

i = 1,⟹ [1, i) т.е. [] ⟹ *для всех* k ∈ [] A[k] ≠ key, что верно, так как любое утверждение относительно пустого множества истинно

Сохранение:

Предположим, что инвариант цикла истинен в начале i-ой итерации цикла.

1) Если A[i] == key текущая итерация является последней, так как выполняется строка 3; Доказывается завершение.

2) Если A[i] ≠ key мы имеем:
для всех k ∈ [1, i) A[k] ≠ key и для A[i] ≠key (так как не вышли)

Что равносильно:

для всех k ∈ [1, i+1) A[k] ≠ key,
это означает, что инвариантный цикл все равно будет истинным в начале следующей итерации (i+1).

Завершение: два случая

1. return i , если A[i] == key; - успешный
2. i == N+ 1 (последний тест цикла – завершение цикла), в этом случае мы находимся в начале N+1-ой итерации, поэтому инвариант цикла

для всех k ∈ [1, N+ 1) A[k] ≠ key

Что эквивалентно: *для всех* k ∈ [1, N] A[k] ≠ key и возвращается значение -1.

Самостоятельно

Описать инвариант алгоритма линейного поиска индекса значения key в массиве a[N], если код алгоритма такой:

```
int i=0;
While(i<N && a[i]!=key)
    i++;
Return (i!=N):i -1;
```

Метод индукции для алгоритма бинарного поиска

Докажем корректность алгоритма двоичного поиска.

```
int binary_search(int arr[], int start, int end, int key)
{
   if (start == end)
   {
      return arr[start] == key ? start : -1;
   }

   int mid = floor(start + (end - start) / 2);

   if (key <= arr[mid])
   {
      // Левая часть
      return binary_search(arr, start, mid, key);
   }
   else
   {
      // Правая часть
      return binary_search(arr, mid + 1, end, key);
   }
}
```

2 4 7 12 18 19 27 30

Нам нужно найти утверждение, которое будет зависеть от n.

Для доказательства корректности работы нам достаточно показать, что алгоритм завершает работу за **конечное число шагов**.

Если алгоритм завершается, то он работает с подмассивами длиной **1.**

Доказательство операции сравнения тривиально.

Нужно удостоверится только в том, что при любых предусмотренных входных данных алгоритм корректно завершится и значение будет возвращено.

Утверждение: Количество рекурсивных вызовов – $\log_2 n$ – натуральное число

Рекурсия спускается до тривиального случая: n=1

Если n не кратно 2, и key в левой части, то действий на один шаг больше (из-за округлений).

При доказательстве будем использовать этот «худший» случай.

Доказательство корректности алгоритма бинарного поиска методом индукции

Доказательство сводится к тому, что нужно определить, что **корректно выполняется самый нижний уровень рекурсии** (когда вызывается функция с одним элементом), а после чего доказать, что самый нижний уровень действительно достижим.

Шаг 1. n= 1, тогда $\lceil \log_2 n \rceil = 0$. При вызове функции с массивом из одного элемента цепочка вызовов прекращается, так как алгоритм натыкается на оператор return, который возвращает позицию найденного элемента, либо -1.

Шаг 2. Далее предполагаем, что наше утверждение верно для k = n.

Докажем, что результатом $\lceil \log_2 (k+1) \rceil$ будет натуральное число.

Так как k является натуральным числом, а логарифм любого натурального числа не может быть отрицательным числом.

Получившийся ответ округляется в большую сторону, что также делает его натуральным числом.

Анализ алгоритма

Анализ алгоритма предназначен для того, чтобы определить требуемые для его выполнения ресурсы:

- время выполнения алгоритма (скорость)
- объем внутренней памяти для хранения входных и промежуточных данных.

Оценка требуемых алгоритму ресурсов выполняется для доказательства его эффективности по времени и памяти.

Эффективность алгоритма

Алгоритм А1, эффективнее алгоритма А2, если алгоритм А1, выполняется за меньшее время и (или) требует меньше компьютерных ресурсов (оперативной памяти, дискового пространства, сетевого трафика и т.п.).

Разрабатываемая программа предназначена для пользователя, и должна устойчиво работать (без прерываний), быть удобной в использовании, а так же эффективной.

Устойчивость и удобство программы зависят от процедур тестирования и разработки.

Эффективность алгоритма зависит от множества факторов, которые включают:

- **вычислительную систему** – системная эффективность (быстродействие системы)
- **объем доступной ОП вычислительной системы**
- **вычислительную сложность алгоритма**

Критерии эффективности алгоритма

Системная эффективность

Оценить относительное **время** выполнения алгоритмов, **используя внутрисистемные часы.**

Оценка времени становится **мерой системной эффективности**.

Этот способ оценки эффективности популярен, и, безусловно, полезен, но он порождает определенные проблемы.

Определяемое таким образом время зависит не только от алгоритма, но и от архитектуры и системы команд компьютера, от качества компилятора, от профессионализма программиста реализовавшего алгоритм.

Эффективность пространства

Это мера используемой алгоритмом внутренней памяти – объем требующейся памяти.

Оценка **объема оперативной памяти**, может указать какого типа компьютер может выполнить этот алгоритм и его полную системную эффективность.

Оценка **объема внешней памяти**, может указать какого объема требуются внешние носители.

Критерии эффективности алгоритма (продолжение)

Вычислительная сложность алгоритма (временная сложность алгоритма)

Этот критерий измеряет **вычислительную сложность алгоритма** относительно **количества (n) обрабатываемых (тестовых) данных.**

Данный параметр оценивает внутреннюю структуру алгоритма, ***т.е. количество выполняемых инструкций (операторов).***

При этом определяется функция, которая укажет **порядок роста времени** выполнения алгоритма от длины тестовых данных: n, n^2 , n^3, log(n), n log(n) и др.

Но время может зависеть не только от размера тестовых данных и от самих тестовых данных.

Этот тип измерения не зависит от **какой-либо машинной системы.**

Понятия количество операций алгоритма и время выполнения алгоритма (execution time) мы будем использовать как синонимы

Сложность алгоритма

Эффективный алгоритм должен удовлетворять требованиям:

- *приемлемое время исполнения*
- *разумной объем оперативной памяти*.

Совокупность этих характеристик составляет понятие ***сложность алгоритма.***

При увеличении задействованных ресурсов **сложность возрастает, а эффективность падает.**

Различают:

1. ***Количественная сложность*** определяется значениями**:**

Временная сложность алгоритма - характеристика алгоритма, отражающая временные затраты на его выполнение.

Емкостная сложность - *х*арактеристика алгоритма, отражающая объем памяти, требующейся алгоритму для хранения входных и промежуточных данных.

2. ***Качественная сложность*** определяет эффективности:

Эффективный алгоритм

Не эффективный алгоритм

Время – мера вычислительной эффективности алгоритма

В реальных вычислениях вопрос состоит в том: *существует ли алгоритм, решающий данную задачу за время, которым мы располагаем?*

Как правило, в алгоритмах с хорошей временной оценкой сложностью используется дополнительная память.

Например, алгоритм сортировки *Простое слияние* существенно быстрее, чем алгоритм сортировки *Пузырьком*. Алгоритм сортировки *Пузырьком* сортирует массив «на месте» – т.е. не использует дополнительную память. Алгоритм сортировки *Простое слияние* использует дополнительную память.

Увеличение скорости сортировки *Простое слияние* за счет большого объема необходимой для сортировки памяти.

Пользователь всегда предпочитает более эффективное решение даже в тех случаях, когда эффективность не является решающим фактором.

И за меру вычислительной эффективности алгоритма можно брать время выполнения алгоритма.

Подходы к определению сложности алгоритма

При количественной (время и объем) и качественной (эффективен или не эффективен) оценках алгоритма, связанных с определением сложности, используют два подхода — практический и теоретический.

- Практическая сложность
- Теоретическая сложность

Практическая сложность

Выполнение алгоритма на конкретных физических устройствах и измерение количественных характеристик алгоритма.

Временная сложность при таком подходе может выражаться **во временных единицах (например, миллисекундах)** или количестве тактов процессора, затрачиваемых на выполнение алгоритма.

Емкостная сложность может выражается в **битах (или других единицах измерения информации),** минимальных аппаратных требованиях, необходимых для выполнения алгоритма, и пр.

Практическая оценка ***не является абсолютным показателем*** эффективности алгоритма. Количественные значения, получаемые при таком подходе, зависят от множества факторов, таких как:

- технические характеристики компонент, составляющих вычислительную систему. Так, чем выше тактовая частота работы процессора, тем больше элементарных операций в единицу времени может быть выполнено;
- характеристики программной среды (количество запущенных процессов, алгоритм работы планировщика заданий, особенностей работы операционной системы и т.д.);
- выбранный язык программирования для реализации алгоритма. Программа, написанная на языке высокого уровня, скорее всего, будет выполняться медленнее и потребует больше ресурсов, нежели программа, написанная на низкоуровневых языках, имеющих непосредственный доступ к аппаратным ресурсам;
- опыт программиста, реализовавшего алгоритм. Скорее всего, начинающий программист напишет менее эффективную программу, чем программист, имеющий опыт.

Таким образом, алгоритм, выполняемый в одной и той же вычислительной системе для одних и тех же входных данных, может иметь различные количественные оценки в различные моменты времени. Поэтому более важным оказывается теоретический подход к определению сложности.

Инструменты C++ определения практической сложности

1) **Использование программного таймера**: замерить время используя системные часы

#include <ctime>

clock() – вычисляет время выполнения программы в миллисекундах

Константа clocks_per_sec , определяющая соответствие 1с= 1000млс

2) **Использование стабильных часов**

При помощи средств, **библиотеку chrono,** появившихся в стандартной библиотеке C++ можно получить более высокую точность измерения и замерить время независимо от системных часов, с помощью так называемых стабильных часов.

#include <chrono>

std::chrono::steady_clock::now()

Теоретическая сложность алгоритма

Характеризует алгоритм без привязки к конкретному оборудованию, программному обеспечению и средствам реализации.

Временная сложность* выражается *в количестве операций, выполняемых алгоритмом* на *некотором идеализированном компьютере, *для которого время выполнения каждого вида операций известно и постоянно.*

Емкостная сложность определяется объемом данных (входных, промежуточных, выходных - n), числом задействованных ячеек памяти.

При такой оценке считается, что ресурсы ***идеализированного компьютера*** бесконечны, поэтому **емкостную сложность при теоретическом подходе обычно *не определяют***.

Количественные оценки, получаемые при теоретическом подходе, зависят

- **от *объем входных данных (от n)***. Чем он больше, тем больше времени потребуется на выполнение алгоритма;
- ***от самих исходных данных (случайно заполненный массив, упорядочены все значения или часть)***
- ***выбранный метод решения задачи***. Например, для большого объема входных данных алгоритм быстрой сортировки эффективное, чем алгоритм пузырьковой сортировки, хотя и приводит к такому же результату.

Временная сложность алгоритма - T(n)

Обозначим: n — **объем** входных данных (размер входа) некоторого алгоритма или **размер задачи.** Время выполнения алгоритма обычно зависит от размера решаемой задачи и от самих данных.

Так при сортировке множества значений объем данных составляет количество элементов этого множества.

А при обработке строк объемом входных данных считается длина строки.

При оценке временной сложности подсчитывается количество выполняемых операций. Часто подсчитывают только критические операции – которые использует алгоритм чаще остальных.

Обозначим за T(n) – функцию, определяющую зависимость времени выполнения алгоритма от размера n обрабатываемых данных, выполняемых на идеализированном компьютере при исполнении алгоритма, в **«наихудшем случае»**.

«Наихудший случай» временной сложности – количество операций, выполняемых алгоритмом максимально.

Три случая оценки временной сложности алгоритма (зависимость от значений данных)

Наилучший случай (best case) – это экземпляр задачи (набор входных данных), на котором алгоритм выполняет наименьшее число операций

Для алгоритма линейного поиска в массиве – это входной массив, первый элемент которого содержит искомое значение x (одно сравнение)

$T_{Best}\ n = 1$

Наихудший случай (worst case) – это экземпляр задачи (набор входных данных), на котором алгоритм выполняет наибольшее число операций

Для алгоритма линейного поиска в массиве – это массив, в котором отсутствует искомый элемент или он расположен в последней ячейке (n сравнений)

$T_{Worst}\ n = n$

Средний случай (average case) – это “средний” экземпляр задачи (набор входных данных), набор “усреднённых” входных данных

В среднем случае оценивается *математическое ожидание количества операций, выполняемых алгоритмом*

Не всегда очевидно, какие входные данные считать «усреднёнными» для задач

Основные правила определения количества операторов в алгоритме

1) В строке алгоритма расположена одна простая команда, то количество равно 1.
2) Учитывается каждая команда в блоке команд.
3) Оператор цикла, в котором количество повторений зависит от n: *оценивается через количество выполняемых сравнений в условии цикла и это количество равно n+1.*
4) Тело цикла выполняется n раз. Тогда количество операций в теле цикла после выполнения всех итераций цикла = количество операторов тела цикла*n.

<u>При выполнении анализа алгоритма и расчете теоретической сложности будем поддерживаться технологии:</u>

- алгоритм выполняется на однопроцессорной машине с памятью RAM (Random-Access Mamory);
- команды выполняются последовательно и одновременно выполняемые команды отсутствуют;
- выполнение каждой команды имеет фиксированное время (оно имеет целый тип или тип с плавающей точкой).

Пример 1. Определение теоретической вычислительной сложности алгоритма

Вычислить среднее арифметическое всех положительных чисел массива A из n элементов. Результат - функция T(n), определяющая зависимость количества выполняемых операций от n. Где ci – константа времени выполнения оператора на идеализированной ЭВМ

Оператор	Количество выполнений оператора	
Sum←0	1	c1
count ←0	1	c2
For i←1 to n **do**	n+1	c3
If(A[i]>0)	n	c4
sum←sum+A[i]	n	c5
count←count+1 endIf	n	c6
od		
If (count≠0)	1	c7
return sum/count	1	c8
Else		
return -1 endIf	1	c9

Определение функции зависимости времени выполнения алгоритма T(n) от размера входных данных

$T(n)=c1*1+c2*1+c3*(n+1)+c4*n+c5*n+c6*n+c7*1+c8*1+c9*1=(c3+c4+c5+c6)n+(c1+c2+c3+c7+c8+c9)=An+B$

Наихудший случай (все числа >0), все n элементов используются в вычислениях:

$T(n)=c1*1+c2*1+c3*(n+1)+c4*n+c5*n+c6*n+c7*1+c8*1=$
$=(c3+c4+c5+c6)n+(c1+c2+c3+c7+c8)=An+B$

Наилучший случай, когда все числа в массиве <0

$T(n)=c1*1+c2*1+c3*(n+1)+c4*n+c7*1+c9*1= =(c3+c4)n+(c1+c2+c3+c7+c9)=An+B$

Средний случай, когда числа в массиве и положительные и отрицательные и нулевые.

Порядок роста: A и B константы, рост времени будет определяться значением переменной n. Константы при определении порядка роста в выражении игнорируются. *при больших* ***n*** *членами меньшего порядка можно пренебречь.*

Т.е. налицо ***линейная зависимость количества операций от количества элементов*** n: $T(n)<n$. Рост времени пропорционален росту объема входных данных n.

Пример 2. Теоретическая сложность алгоритма, содержащего вложенный цикл, при этом и внешний и внутренний цикл зависят от размера задачи – n. Дана матрица размером n*m. Найти максимальный элемент среди положительных чисел матрицы

Оператор	Количество выполнений оператора	Время выполнения операции
max←A[1,1]	1	c1
For i←1 to n **do**	n+1	c2
For j←1 to m **do**	n*(m+1)	c3
If(A[I,j]>max)	n*m	c4
max←A[i,j]	n*m	c5
endIf		
od **od**		
return max	1	c6

Определение функции зависимости времени выполнения алгоритма от размера входных данных

Наихудший случай – все числа положительные.

T(n)=c1*1+c2*(n+1) +c3*(n*m+n)+ c4*n*m + c5*n*m n+c6*1=A(n*m)+Bn+C т.е. сложность алгоритма зависит от произведения nm. Если матрица была квадратной и m=n, то зависимость была бы n2. Порядок роста времени квадратично зависит от объема входных данных n

Наилучший случай – если бы не было положительных совсем, то операторы под if не выполнялись, и сложность вычислялась бы так же, только без одного слагаемого n*m.

T(n)=A(n*m)+Bn+ C т.е. сложность тоже n*m

Средний случай - если бы было какое-то количество положительных, то

2nm+2n+3<=T(n)<= 3nm+2n+3, и 2nm<=T(n)<= 3nm.

Емкостная сложность: O(n*m).

Примечание. Если матрица квадратная, то m=n , $T(n) = An^2+Bn+C$ и рост определяется слагаемым $T(n) = An^2$

Определение функции зависимости времени выполнения алгоритма от размера входных данных. Алгоритм сортировки Простой вставки

Insertion_Sort(A)	**Количество операторов**	**Время одного оператора**
j←2;	1	c1
while(j<=n) do	n	c2
key ← a[j];	n-1	c3
i ← j;	n-1	c4
while(i>0 и a[i-1]>key) **do**	$\sum_{j=2}^{n} t_j$	c5
a[i] ← a[i-1];	$\sum_{j=2}^{n} (t_j - 1)$	c6
i ← i-1; **od**	$\sum_{j=2}^{n} (t_j - 1)$	c7
a[i] ← key;	n-1	c8
j ← j+1; od	n-1	c9

Вывод функции зависимости времени выполнения алгоритма от размера входных данных: Алгоритм сортировки Простой вставки

Время работы алгоритма Insertion_Sort (T(n))(где tj количество

выполнения цикла while) :

T(n) =c1*1+c2*n+c3*(n-1)+c4*(n-1)+c5* $\sum_{j=2}^{n}(t_j)$+c6* $\sum_{j=2}^{n}(t_j-1)$+c7* $\sum_{j=2}^{n}(t_j-1)$+c8*(n-1)+c9*(n-1)

Наилучший случай (все исх. данные упорядочены) – тело внутреннего цикл не выполняется (суммы при c6 и c7 равны 0)

Тогда для каждого j = 2,3,..., n и tj = 1- заголовок while выполняется $\sum_{j=2}^{n} 1 = (n-1)$

T(n) = c1*1+c2*n+c3*(n-1)+c4*(n-1)+c5* $(n-1)$+c8*(n-1)+c9*(n-1) =A*n+B где A и B ***константы***, зависящие от величин c_i , т.е. это время является ***линейной функцией*** от n. Т.е. функциональная зависимость времени от размера массива определяется линейной функцией n.

Наихудший случай (все исх. данные в обратном порядке)

Тогда для каждого j = 2,3,..., n t_j = j, и, с учетом того, что

$\sum_{j=2}^{n}(t_j)$=2+3+…+ n = n(n-1) /2 -1 - сумма членов арифметической прогрессии

1+2+…+ (n-1) = n(n-1) /2 , получаем, приведя подобные:

T (n) = $((c_5 + c_6 + c_7)/2)\, n^2 + (c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8)\, n - (c_2 + c_4 + c_5 + c_8)$.

Это время работы можно записать как **A n^2** + B n + C , где A, B, C - ***константы***, зависящие от величин c_i , т.е. это ***квадратичная функция*** от n.

Получаем, что функциональная зависимость времени от n в худшем случае определяется функцией n^2

Емкостная сложность так же линейно зависит от размера массива n, так как дополнительный массив не использовался.

Примечание

В процессе теоретического исследования можно использовать метод прямого подсчета количества критических операций (или групп операций). Для этого следует выполнить предварительный анализ, в результате которого выявить т.н. критические операции (группы операций).

Критическими (основными) называют операции, которые выполняются *наиболее часто и время выполнения которых составляет основную часть общего времени выполнения алгоритма.*

Например.

Для алгоритма поиска – это операция сравнения.

Для алгоритма сортировки это количество сравнений и перемещений элементов.

Как определить эффективный алгоритм?

Пусть имеются два алгоритма решения одной и той же задачи. Сложность алгоритмов определена функциями.

- $T1\ (n) = 90n^2 + 201n + 2000$
- $T2\ (n) = 2n^3 + 3$

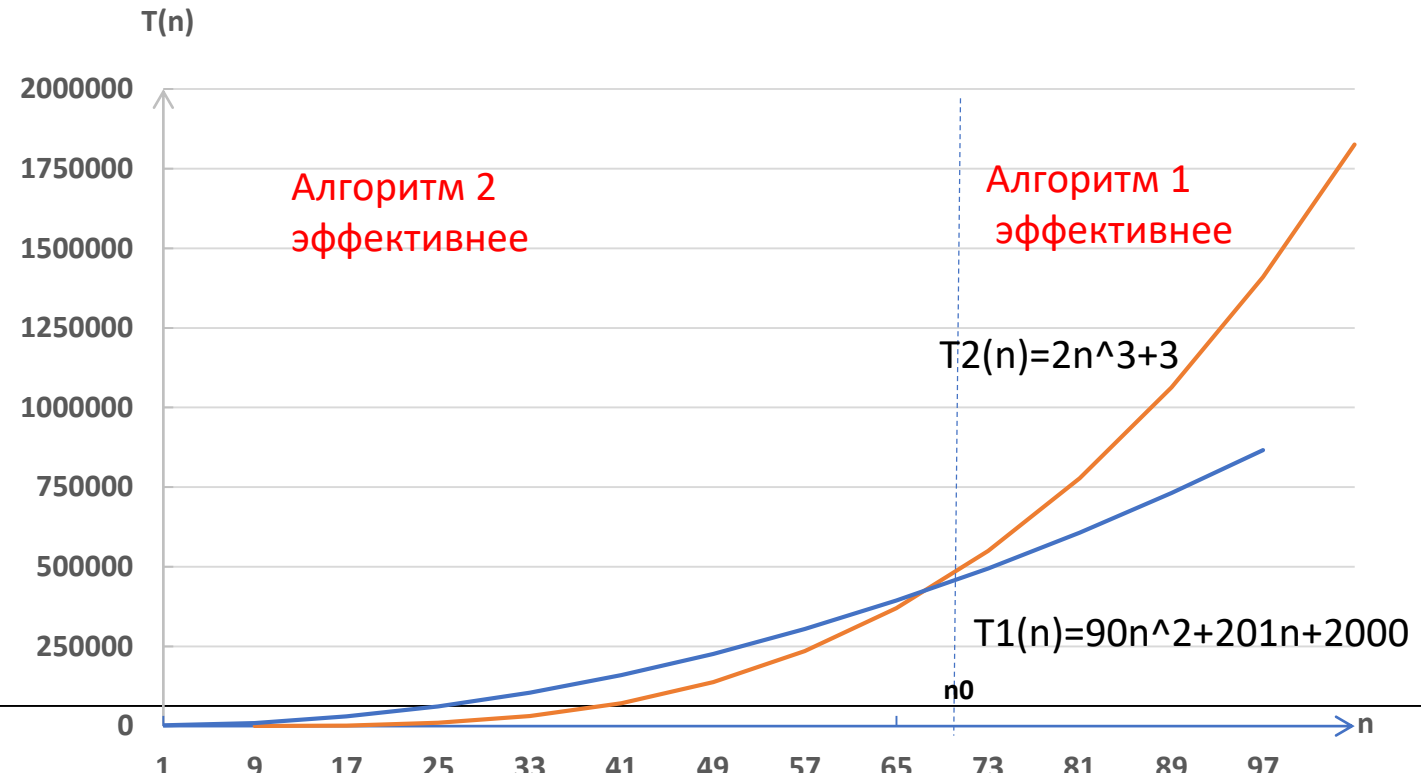
Какой из алгоритмов выбрать в качестве эффективного?

Потребуется математический аппарат, дающий ответ на вопрос какая из функций растёт быстрее при $\boldsymbol{n} \to \infty$

Ответы на эти вопросы дает ***асимптотический анализ*** (asymptotic analysis), который позволяет оценивать скорость роста функций $T(n)$ при стремлении размера входных данных к бесконечности (при n → ∞)

Анализ сложности алгоритма

- Если известно, что на вход будут поступать **данные небольших размеров**, то вопрос о выборе эффективного алгоритма не является первостепенным – можно использовать самый “простой” алгоритм
- Вопросы, связанные с эффективностью алгоритмов, приобретают смысл при больших размерах входных данных – при $n \to \infty$



Мы можем найти такое значение n0 при котором происходит пересечение функций T1(n) и T2(n), и на основе n0 отдавать предпочтение тому или иному Алгоритму

n	T1(n)	T2(n)	
1	2291	5	-2286
9	11099	1461	-9638
17	31427	9829	-21598
25	63275	31253	-32022
33	106643	71877	-34766
41	161531	137845	-23686
49	227939	235301	7362
57	305867	370389	64522
65	395315	549253	153938
73	496283	778037	281754
81	608771	1062885	454114
89	732779	1409941	677162
97	868307	1825349	957042