Алгоритмически неразрешимые задачи

- Любая вычислимая функция может задаваться разными алгоритмами (разными программами для выбранного универсального исполнителя) и может быть вычислена с помощью любого универсального исполнителя: машин Тьюринга и Поста, нормальных алгоритмов Маркова и др., но существуют и алгоритмически невычислимые функции.
- Однако алгоритмическая неразрешимость задачи того или иного класса вовсе не означает невозможность решения любой конкретной задачи из этого класса. Речь идет о невозможности решения всех задач данного класса одним и тем же приемом.
- Формализация понятия алгоритма позволила исследовать существование задач, для которых нет алгоритмических решений.

Примеры алгоритмически неразрешимых задач.

Пример 1.

Вычисление функции h(n), которая для любого натурального числа n равна 1, если в десятичной записи числа π есть n стоящих подряд девяток, окруженных другими цифрами, и равна нулю, если такой цепочки девяток нет.

Пример 2.

Десятая проблема Гильберта, сформулированная в 1900 году, состоит в нахождении алгоритма решения произвольных алгебраических диофантовых уравнений вида $P(x_1, \ldots, x_m) = 0,$ где P — целочисленная функция (например, полином с целыми коэффициентами), а переменные x_i принимают целые значения.

$x^n + y^n = z^n$ - решениями этого уравнения являются пифагоровы тройки.

Согласно теореме Ферма это уравнение не имеет ненулевых целых решений при n>2.

Пример 3.

Проблема Эйлера - любое четное число не меньше четырех можно представить в виде суммы двух простых чисел.

Пример 4.

Теорема Гёделя о неполноте формальной арифметики. Существуют некоторые утверждения, которые не могут быть ни доказаны, ни опровергнуты на основе любого набора непротиворечивых аксиом (такие утверждения называются невыводимыми).

Методы доказательства алгоритмической неразрешимости

- **Прямой метод** использует диагональный метод Кантора. Заключается он в следующем: из предположения о разрешимости данной проблемы в ходе рассуждений приходят к противоречию.
- **Косвенный метод** состоит в следующем: показывается, что разрешимость исследуемой проблемы влечёт разрешимость проблемы, о которой уже известно, что она неразрешима. Метод сведения часто бывает более удобным, чем прямой метод. Применяя метод сведения, обычно ссылаются на искусственные задачи, которые не представляют самостоятельного интереса, но для которых легко непосредственно доказать их неразрешимость.

Диагональный метод Кантора

Теорема Кантора о несчетности множества действительных чисел: множество натуральных чисел и множество действительных чисел сегмента [0, 1] имеют разную мощность.

Доказательство (от противного).

Действительные числа сегмента [0, 1] будем представлять бесконечной десятичной дробью, у которой на первом месте 0, а далее через запятую следует бесконечная последовательность цифр: например 0,31415926536... Положим, что такое соответствие установлено (т.е. положим, что мы занумеровали все числа отрезка [0, 1].)

1 $0, a_{1,1}a_{1,2}a_{1,3}a_{1,4}a_{1,5}a_{1,6}a_{1,7}\ldots$

2 $0, a_{2,1}a_{2,2}a_{2,3}a_{2,4}a_{2,5}a_{2,6}a_{2,7}\ldots$

3 $0, a_{3,1}a_{3,2}a_{3,3}a_{3,4}a_{3,5}a_{3,6}a_{3,7}\ldots$

4 $0, a_{4,1}a_{4,2}a_{4,3}a_{4,4}a_{4,5}a_{4,6}a_{4,7}\ldots$

5 $0, a_{5,1}a_{5,2}a_{5,3}a_{5,4}a_{5,5}a_{5,6}a_{5,7}\ldots$

6 $0, a_{6,1}a_{6,2}a_{6,3}a_{6,4}a_{6,5}a_{6,6}a_{6,7}\ldots$

. . . . . .

n $0, a_{n,1}a_{n,2}a_{n,3}a_{n,4}a_{n,5}a_{n,6}a_{n,7}\ldots$

. . . . . .

$a_{i,j}$ - цифра от 0 до 9, i - номер числа, в записи которого она участвует, j - номер ее позиции в этом числе. Докажем, что есть число не вошедшее в эту нумерацию.

Строим число 0, $b_1 b_2 b_3 b_4 b_5 \ldots$, ставя на n-ое место цифру b_n, такую, что $b_n \neq a_{n,n}$. Таким образом получаем число отличное от числа с номером n. Поскольку так можно проделать для любого числа, получаем противоречие предположению, что можно занумеровать все действительные числа.

Теорема: множество арифметических функций n-переменных несчетно.

Док-во (с помощью диагонального метода):

Для доказательства несчетности множества достаточно доказать несчетность какого-нибудь его подмножества. Рассмотрим функции одной переменной вида Fi(x). Пусть функций одной переменной счетное множество, т.е. их можно перенумеровать. F0(x), F1(x), F2(x), …

Построим новую функцию G(x)=F_x(x)+1. Это так называемая диагональная функция G(0)=F_0(0)+1, G(1)=F_1(1)+1, G(2)=F_2(2)+1 и т.д. G-отлична от всех перечисленных функций, т.к. от каждой из функций она отличается хотя бы в одной точке. От функции F_0(x) отличается в точке x=0, от функции F_1(x) в точке x=1 и т.д. Однако по построению G(x) принадлежит множеству арифметических функций одной переменной, значит должна быть в списке, т.е. совпадать с одной из перечисленных функций.

Получили противоречие, следовательно исходное предположение неверно, и функций одной переменной несчетное множество. А значит и всех функций n переменных – тоже несчетное множество.

Теорема: вычислимых функций счетное множество. (Множество машин Тьюринга счетно).

Программу всякой машины Тьюринга можно интерпретировать как слово в некотором конечном алфавите. Это можно сделать, например, так. Пусть $A = \{0, 1, 2, ..., 9, a, q, R, L, S, \rightarrow, |\}$ – алфавит. Он содержит 17 символов. Рассмотрим программу машины Тьюринга, например, такую:

$q_1a_1 \rightarrow q_ka_lR,$
$q_1a_2 \rightarrow q_ia_jL,$
...
$q_ma_n \rightarrow q_ra_tS.$

Запишем теперь все команды в одну строчку, разделяя их символом | и заменяя нижние индексы на равные им числа, но расположенные уже на уровне основного текста:

$$q1a1 \rightarrow qkalR|q1a2 \rightarrow qiajL|...|qman \rightarrow qratS.$$

Мы получили слово в алфавите A. По этому слову программа машины Тьюринга восстанавливается однозначно. Но очевидно, что не всякое слово из A^* является программой некоторой машины Тьюринга.

Таким образом, каждая машина Тьюринга вполне определяется некоторым конечным словом в конечном стандартном алфавите. Поскольку множество всех конечных слов в конечном алфавите счетно, то и всех мыслимых машин Тьюринга (отличающихся друг от друга по существу своей работы) имеется не более чем счетное множество.

Оценка мощности множества невычислимых арифметических функций.

Невычислимых арифметических функций несчетное множество.

Проблемы самоприменимости и остановки.

Эти задачи часто используют для доказательства неразрешимости других проблем путем сведения к ним.

Нумерация алгоритмов

Существует вычислимая функция, которая по номеру машины Тьюринга (алгоритма) восстанавливает её программу (описание алгоритма) ϕ: $N \to A$. Такая функция называется нумерацией алгоритмов. Это позволяет отождествлять алгоритм с его номером. Если $\phi(n)=A$, то число n называется номером алгоритма A. Из взаимной однозначности отображения ϕ следует существование обратной функции ϕ^{-1}, восстанавливающей по описанию алгоритма A_n его номер в этой нумерации $\phi^{-1}(A_n)=n$.

Существование нумераций позволяет работать с алгоритмами как с числами. Это особенно удобно при исследовании алгоритмов над алгоритмами.

Проблема остановки.

Не существует алгоритма (машины Тьюринга), позволяющего по описанию произвольного алгоритма (его номеру) и исходных данных (и алгоритм и данные заданы символами на ленте машины Тьюринга) определить, останавливается ли этот алгоритм на этих данных или будет работать бесконечно.

Самоприменимость (частный случай проблемы остановки) в теории алгоритмов — свойство алгоритма успешно завершаться на данных, представляющих собой формальную запись этого же алгоритма.

Пример самоприменимого алгоритма: тождественные преобразования строк в алфавите A.

Теорема. Не существует машины Тьюринга T_0 , которая решает проблему самоприменимости.

Возьмем в качестве внешнего алфавита для машин Тьюринга A={0,1}. Будем говорить, что МТ T_0 решает **проблему**

самоприменимости, если для любой машины T конфигурацию q_1Код(T) она переводит в конфигурацию $q_0$1, если T самоприменима, и в конфигурацию $q_0$0, если T - несамоприменима.

Доказательство.

Допустим, что существует машина T_0, решающая проблему самоприменимости. Построим машину T_1, в которой вместо состояния q_0 введем новое заключительное состояние q_r и добавим к программе машины T_0 новые команды

- $q_0 1 \rightarrow q_0 1E$, (зацикливание)
- $q_0 0 \rightarrow q_r 0E$ (*)

Машина T_1 построена по машине T_0 вполне конструктивными средствами и применима к кодам несамоприменимых машин и не применима к кодам самоприменимых машин.

Существование такой машины приводит к противоречию, потому что T_1 не может быть ни самоприменимой, ни несамоприменимой.

Действительно, если T_1 - самоприменима, то q_1Код(T_1) переходит в $q_0$1 и согласно (*) $q_0$1 в $q_0$1E и T_1 никогда не остановится, т.е. по построению она не применима к коду самоприменимых машин. Если T_1 - несамоприменима, то q_1Код(T_1) переходит в $q_0$0 и согласно (*) $q_0$0 в qr0 и машина T_1 остановится, т.е. по построению она применима к собственной записи, так как она применима к любой записи несамоприменимой машины, а это как раз означает, что T_1 самоприменима. Получили противоречие, т.е. допущение о существовании МТ, решающей проблему самоприменимости, неверно. В силу тезиса Тьюринга невозможность построения МТ означает отсутствие алгоритма решения данной проблемы.

Проблема останова МТ и доказательство её неразрешимости

- Одна из первых задач, для которой была доказана неразрешимость.
- Доказательство её неразрешимости проводится с помощью диагонального метода и свойства самоприменимости алгоритма.
- Задачу останова часто используют для доказательства неразрешимости других проблем путем сведения к ней.

  Теорема. *Не существует алгоритма (МТ), позволяющего по описанию произвольного алгоритма и его исходных данных (и алгоритм и данные заданы символами на ленте машины Тьюринга) определить, останавливается ли этот алгоритм на этих данных или будет работать бесконечно.*

Доказательство.

Рассмотрим множество всех алгоритмов, получающих на вход натуральное число, и возвращающих в качестве результата натуральное число, т.е. отображения N -> N*, где N* = N ∪ undef, undef — случай, когда алгоритм зацикливается, то есть не заканчивает свою работу. Эта абстракция допустима, так как слова в любом конечном алфавите можно однозначно закодировать натуральными числами.

Докажем, что не существует универсальной функции, которая определяет остановится ли алгоритм на данном входе или будет работать бесконечно.

Пусть существует вычислимая функция F(a,x), принимающая значения на N*. Первый аргумент a — номер описания алгоритма на некотором языке, второй аргумент x — входные данные для этого алгоритма. F(a,x) по определению есть результат выполнения алгоритма a на

Вычислимая функция F(a,x) двух натуральных аргументов как бы перечисляет ВСЕ вычислимые функции с одним натуральным аргументом. (Предполагается, что натуральными числами a шифруются множество всех алгоритмов.)
Рассмотрим эту функцию с точки зрения самоприменимости т.е. F(x,x), где входом для алгоритма с номером x будет формальная запись этого же алгоритма, и построим функцию h(x) = F(x,x)+1. Функция h(x) - вычислимая, так как она использует результат вычислимой функции F и после прибавляет к нему единицу. Пусть функция h(x) имеет номер a, то есть F(a,x)=h(x). Но по определению h(x)=F(x,x)+1 и при x=a имеем F(a,a)=h(a) и h(a)=F(a,a)+1. Получили противоречие.

Таким образом определение того, остановится или нет программа, является **невычислимой функцией**.

Неразрешимость проблемы останова можно интерпретировать как несуществование общего алгоритма для отладки программ, точнее, алгоритма, который по тексту любой программы и данным для нее определял бы, зациклится ли программа на этих данных или нет.

Основы анализа сложности алгоритмов

Критерии оценки эффективности алгоритмов:

- **Процессорное время** (вычислительная сложность)
- **Память** (максимальное количество ячеек задействованных алгоритмом)

Каждое вычислительное устройство имеет свои особенности, которые могут влиять на длительность вычисления при этом алгоритм не становится хуже или лучше!

Пример:

Необходимо отсортировать массив из миллиона чисел. Имеется два алгоритма: один требует выполнения $2n^2$ операций, другой $50n\log(n)$ - операций. Имеется два компьютера: один выполняет 10^8 операций в секунду, др

$$\frac{2\cdot(10^6)^2 \text{ операций}}{10^8 \text{ операций в сек}} = 20000 \text{ секунд} \approx 5{,}56 \text{ часов}$$

$$\frac{50\cdot 10^6 \log(10^6) \text{ операций}}{10^6 \text{ операций в сек}} \approx 1000 \text{секунд} \approx 17 \text{ минут}$$

Модель абстрактного вычислителя - *машина с произвольным доступом к памяти (RAM)*

Модель состоит из памяти и процессора, которые работают следующим образом:

- память состоит из ячеек, каждая из которых имеет адрес и может хранить один элемент данных;
- каждое обращение к памяти занимает одну единицу времени, независимо от номера адресуемой ячейки;
- количество памяти достаточно для выполнения любого алгоритма;
- процессор выполняет любую элементарную операцию (основные логические и арифметические операции, чтение из памяти, запись в память, вызов подпрограммы и т.п.) за один временной шаг;
- циклы и подпрограммы не считаются простыми операциями.

Число элементарных операций алгоритма на этой модели показывает относительное время выполнения алгоритма.

Неудобно оценивать алгоритм по фактическому количеству элементарных операций на тех или иных входных данных.

Задача анализа сложности алгоритма состоит в исследовании того, как меняется время работы при увеличении объема входных данных. Поэтому временная сложность алгоритма определяется числовой функцией, соотносящей время работы алгоритма с размером задачи, т. е. показывающей ***зависимость числа операций*** конкретного алгоритма от размера входных данных, ***что дает возможность сравнить два алгоритма по скорости роста числа операций.***
Именно скорость роста играет ключевую роль, поскольку при небольшом размере входных данных алгоритм A на входе длины n может требовать меньшего количества операций, чем алгоритм B, но при росте объема входных данных ситуация может поменяться на противоположную.
Пример:
Сложность алгоритма A = $372n^3 + 15n^2 + 100$
Сложность алгоритма B = $2n^4$
На входе $n=186$ почти одинаковое количество операций, при $n > 187$, второй алгоритм выполняет большее количество операций.

Формальное описание:

Размер входа определяется для каждой задачи индивидуально.

1) в задачах обработки одномерных массивов размером входа принято считать количество элементов в массиве;

2) в задачах обработки двумерных массивов размером входа так же является количество элементов в массиве;

3) в задачах обработки чисел (длинная арифметика, проверка на простоту и т.д.) более естественно считать размером общее число битов, необходимое для представления данных в памяти компьютера;

4) в задачах обработки графов разумно за размер входа принять количество вершин графа, а иногда представить двумя значениями: число вершин и число ребер графа.

Формальное описание

- Конкретная проблема задается N словами памяти по α битов каждое $N_\alpha = N*\alpha$
- Программа, реализующая алгоритм состоит из M машинных инструкций по β битов – $M_\beta = M*\beta$
- S_d – память для хранения промежуточных результатов
- S_r – память для организации вычислительного процесса

Трудоёмкость алгоритма - количество «элементарных» операций совершаемых алгоритмом для решения конкретной проблемы в данной формальной системе.

Функцией трудоемкости T_a **(N)** называется отношение, связывающие входные данные алгоритма с количеством элементарных операций.

Комплексная оценка алгоритма: (*c_i* – веса ресурсов.)

Зависимость трудоемкости от входных данных

Не всегда количество элементарных операций, выполняемых алгоритмом на одном входе длины N, совпадает с количеством операций на другом входе такой же длины.

Пусть D_A – множество конкретных проблем данной задачи, заданное в формальной системе. Пусть $D \in D_A$ –конкретная проблема и $|D| = N$.

Обозначим подмножество множества D_A, включающее все конкретные проблемы, имеющие мощность N через D_N: $D_N = \{D \in D_N : |D| = N\}$ и мощность множества D_N через $M_{DN} = |D_N|$.

Ведем следующие обозначения:

1. $T_a^{\wedge}(N)$ – *худший случай* – наибольшее количество операций, совершаемых алгоритмом A для решения конкретных проблем размерностью N:

$$T_a^{\wedge}(N) = \max_{D \in D_N} \{T_a(D)\}$$ – худший случай на D_N

2. $T_a^{\vee}(N)$ – *лучший случай* – наименьшее количество операций, совершаемых алгоритмом A для решения конкретных проблем размерностью N:

$$T_a^{\vee}(N) = \min_{D \in D_N} \{T_a(D)\}$$ – лучший случай на D_N

3. $\overline{T}_a(N)$ – *средний случай* – среднее количество операций, совершаемых алгоритмом A для решения конкретных проблем размерностью N.

Для нахождения среднего значения, сначала определяются всевозможные группы, на которые следует разбить входные данные так, чтобы время работы алгоритма на всех данных одной группы было одинаковым. Затем подсчитывается время работы алгоритма на данных из каждой группы и определяется вероятность, с которой входной набор данных принадлежит каждой группе. Среднее время вычисляется как сумма (по всем группам) произведений вероятностей того, что входные данные принадлежат данной группе на время обработки данных этой группы.

$$\overline{T}_a(N) = \sum_{D \in D_N} P_d * \{T_a(D)\}$$ – средний случай на D_N

Классификация алгоритмов по виду функции трудоёмкости

- ***1. Количественно-зависимые***

  Это алгоритмы, функция трудоемкости которых зависит только от размерности конкретного входа, и не зависит от конкретных значений:

  $T_a(D) = T_a(|D|) = T_a(N)$

- ***2.Параметрически-зависимые***

  Это алгоритмы, трудоемкость которых определяется конкретными значениями обрабатываемых слов памяти:

  $T_a(D) = T_a(d_1, \ldots, d_n) = T_a(P_1, \ldots, P_m),\ m \le n$

Пример:

- а) Вычисление x^k последовательным умножением $\Rightarrow T_a(x, k) = T_a(k)$.
- б) Вычисление $e^x = \sum(x^n/n!)$, с точностью до $\xi \Rightarrow T_a = T_a(x, \xi)$

- ***3. Количественно-параметрические***

  Функция трудоемкости зависит как от количества данных на входе, так и от значений входных данных, в этом случае:

  $T_a(D) = T_a(|D|, d_1,\ldots,d_m) = T_a(N, P_1,\ldots,P_m)$

- ***3.1. Порядково - зависимые***

  Количество операций зависит от порядка расположения исходных объектов.

  Пусть множество D состоит из элементов $(d_1,\ldots,d_n)$, и $|D|=N$. Определим $D_p = \{(d_1,\ldots,d_n)\}$-множество всех упорядоченных N-ок из $d_1,\ldots,d_n$, отметим, что $|D_p|=n!$.

  Если $T_a({}_iD_p) \neq T_a({}_jD_p)$, где ${}_iD_p, {}_jD_p \in D_p$, то алгоритм будем называть порядково-зависимым по трудоемкости.

Асимптотический анализ алгоритмов

Часто работать непосредственно с функцией трудоемкости сложно, т. к. они обладают такими свойствами:

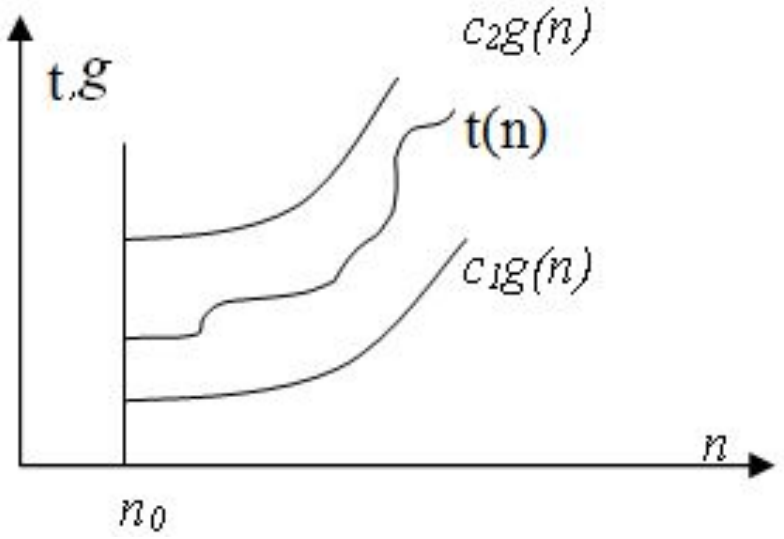
- являются слишком волнистыми, когда сильно влияние исходных данных на функцию трудоемкости;
- требуют слишком много информации для точного определения количества инструкций RAM-машины, а потому реально их определить только для простых алгоритмов;
- при изменении хотя бы одной операции, необходимо по новой пересчитывать коэффициенты;
- при достаточно больших n коэффициенты перестают играть существенную роль.

Цель асимптотического анализа – сравнение затрат ресурсов системы различными алгоритмами, предназначенными для решения одной и той же задачи

Основные оценки сложности

1. **Оценка Θ(g(n)) (тетта) - функции, растущие с той же скоростью, что и g.**

Пусть T*(n)* и *g(n)* – положительные функции положительного аргумента, *n ≥ 1.*
Говорят, что время работы алгоритма T(n) имеет порядок роста g(n), если существуют натуральное число n_0 и положительные константы c_1 и c_2 ($0 < c_1 \le c_2$), такие, что для любого натурального n начиная с n_0 выполняется неравенство :
$c_1 \cdot g(n) \le T(n) \le c_2 \cdot g(n)$.
Обозначение: T(n) = Θ(g(n)).
Читается как «Тэта большое от g от n».
T(n) = Θ (g(n)), если ∃ $c_1 > 0$, $c_2 > 0$, $n_0 > 0$ такие, что: $c_1 g(n) \le T(n) \le c_2 g(n)$, *для* $\forall n > n_0$.



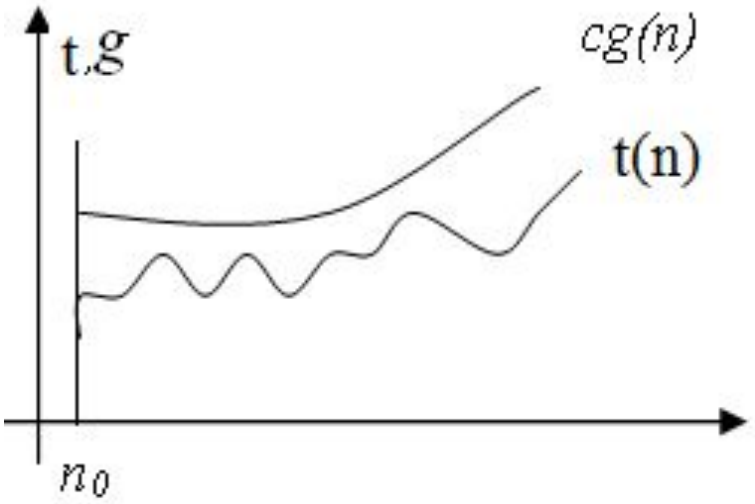
Обычно говорят, что функция *g(n)* является *асимптотически точной оценкой* функции *T(n),* т.к. по определению функция T*(n)* не отличается от функции *g(n)* с точностью до постоянного множителя.
Отношение симметрично:
T*(n) = Θ (g(n))* следует, что *g(n) = Θ (T(n)), но из* $T_1(n) = \Theta(g(n))$ и $T_2(n) = \Theta(g(n))$ не следует, что $T_1(n) = T_2(n)$.

2. Оценка *O(g(n))* (*O* большое) - функции, растущие медленнее чем g.

В отличие от оценки *Θ,* оценка *O* требует только, что бы функция T*(n)* не превышала *g(n)* начиная с некоторого *n > n0*, с точностью до постоянного множителя: $\exists\ c > 0, n_0 > 0 : 0 \le T(n) \le cg(n),\ \forall n > n_0$

Вообще, запись *O(g(n))* обозначает класс функций, таких, что все они растут не быстрее, чем функция *g(n)* с точностью до постоянного множителя, поэтому иногда говорят, что *g(n)* мажорирует функцию T*(n).*

Например, для всех функций: *t(n)=1/n, t(n)= 12, t(n)=3*n+17, t(n)=n*ln(n), t(n)=6*n²+24*n+77* будет справедлива оценка *O(n²). Однако, у*казывая оценку *O* есть смысл указывать наиболее «близкую» мажорирующую функцию.



Отыскивая асимптотическую оценку для суммы, можно отбрасывать члены меньшего порядка, которые при больших n становятся малыми по сравнению с основным слагаемым. Коэффициент при старшем члене роли не играет так как он может повлиять только на выбор констант c.

Пример. Доказать, что функция $T(n) = 3 \cdot n^3 + 2 \cdot n^2$ имеет верхнюю оценку $O(n^3)$.

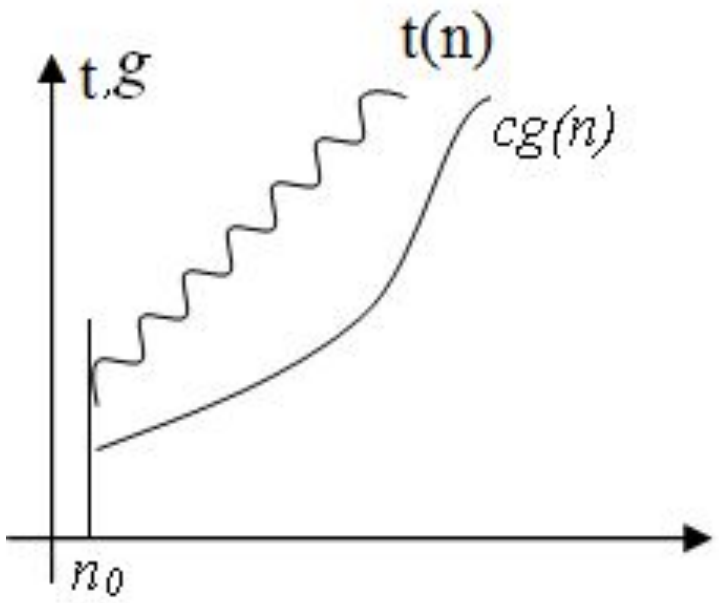
Положим n_0=1, тогда $\forall n \geq 1$ должно выполняться неравенство $3 \cdot n^3 + 2 \cdot n^2 \leq c \cdot n^3$. Преобразуем это неравенство $3 + \frac{2}{n} \leq c$, т. е. $c \geq 5$.

Таким образом, найдены натуральное число n_0=1 и положительная константа c=5, такие, что для любого натурального n начиная с n_0 выполняется неравенство $3 \cdot n^3 + 2 \cdot n^2 \leq c \cdot n^3$, следовательно, $T(n)=O(n^3)$.

3. Оценка Ω (Омега) - функции, растущие быстрее чем g.

В отличие от оценки *O*, оценка *Ω* является оценкой снизу – т.е. определяет класс функций, которые растут не медленнее, чем *g(n)* с точностью до постоянного множителя:

$$\exists\ c > 0,\ n_0 > 0 : 0 \le c\,g(n) \le t(n)\ \ \forall n > n_0$$



Пример:

запись *Ω(n*Ln(n))* обозначает класс функций, которые растут не медленнее, чем *g(n) = n*Ln(n),* в этот класс попадают все полиномы со степенью большей единицы, равно как и все степенные функции с основанием большим единицы.

Свойства асимптотических оценок

Транзитивность

$T(n) = \Theta(g(n))$ и $g(n) = \Theta(h(n))$ влечет $T(n) = \Theta(h(n))$.

$T(n) = O(g(n))$ и $g(n) = O(h(n))$ влечет $T(n) = O(h(n))$.

$T(n) = \Omega(g(n))$ и $g(n) = \Omega(h(n))$ влечет $T(n) = \Omega(h(n))$.

Рефлексивность

$T(n) = \Theta(T(n))$, $T(n) = O(T(n))$, $T(n) = \Omega(T(n))$

Симметричность

$T(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$

Сравнение скорости роста

Чем меньше степень роста функции трудоемкости, тем больше размер задачи, которую можно решить на компьютере.

Предположим, что имеются 4 программы, временная сложность которых $100n, 5n^2, n^3/2, 2^n$ и два компьютера: на первом можно использовать 1000 единиц машинного времени для решения задачи, на втором в 10 раз больше.

Время выполнения T(n)	*Максимальный размер задачи для 10^3 единиц*	*Максимальный размер задачи для 10^4 единиц*	*Увеличение максимального размера задачи*
$100n$	10	100	10
$5n^2$	14	45	3.2
$n^3/2$	12	27	2.3
2^n	10	13	1.3

Классы сложности.

Классы сложности алгоритмов в зависимости от функции трудоемкости	
Вид f(n)	**Характеристика класса алгоритмов**
1	Большинство инструкций большинства функций запускается один или несколько раз. Если все инструкции программы обладают таким свойством, то время выполнения программы постоянно.
log N	Когда время выполнения программы является логарифмическим, программа становится медленнее с ростом N. Такое время выполнения обычно присуще программам, которые сводят большую задачу к набору меньших подзадач, уменьшая на каждом шаге размер задачи на некоторый постоянный фактор.
N	Когда время выполнения программы является линейным, это обычно значит, что каждый входной элемент подвергается небольшой обработке.
N log N	Время выполнения, пропорциональное N log N, возникает тогда, когда алгоритм решает задачу, разбивая ее на меньшие подзадачи, решая их независимо и затем объединяя решения. Время выполнения такого алгоритма равно N log N.
N^2	Когда время выполнения алгоритма является квадратичным, он полезен для практического использования при решении относительно небольших задач. Квадратичное время выполнения обычно появляется в алгоритмах, которые обрабатывают все пары элементов данных (возможно, в цикле двойного уровня вложенности).
N^3	Похожий алгоритм, который обрабатывает тройки элементов данных (возможно, в цикле тройного уровня вложенности), имеет кубическое время выполнения и

	практически применим лишь для малых задач. Когда N=100, время выполнения равно одному миллиону. Когда N удваивается, время выполнения увеличивается в восемь раз.
2^N	Лишь несколько алгоритмов с экспоненциальным временем выполнения имеет практическое применение, хотя такие алгоритмы возникают естественным образом при попытках прямого решения задачи, например полного перебора. Когда N=20, время выполнения имеет порядок одного миллиона. Когда N удваивается, время выполнения увеличивается экспоненциально.

Класс 0 – это класс быстрых алгоритмов с постоянным временем выполнения, их функция трудоемкости O(1). Промежуточное состояние занимают алгоритмы со сложностью O(log N), которые также относят к данному классу.

Класс P – это класс рациональных или полиномиальных алгоритмов, функция трудоемкости которых определяется полиномиально от входных параметров. Например, O(N), $O(N^2)$, $O(N^3)$, к этому классу можно отнести и алгоритмы со степенью трудоемкости O(N log N).

Класс E – это класс собственно экспоненциальных алгоритмов со степенью трудоемкости $O(2^N)$.

Класс F – это класс надэкспоненциальных алгоритмов. Существуют алгоритмы с факториальной трудоемкостью, но они в основном не имеют практического применения.

Отличие полиномиальных и экспоненциальных алгоритмов будет более ощутимо, если обратиться к таблице, в которой отображено время работы алгоритма на компьютере, выполняющем 1 000 000 оп/с:

$O(n)$ \ n	10	20	30	40	50	60
n	0.00001 с	0.00002 с	0.00003 с	0.00004 с	0.00005 с	0.00006 с
n^2	0.0001 с	0.0004 с	0.0009 с	0.0016 с	0.0025 с	0.0036 с
n^3	0.1 с	3.2 с	24.3 с	1.7 мин	5.2 мин	13 мин
2^n	0.001 с	1 с	17.9 мин	12.7 дней	35.7 лет	366 столет.
3^n	0.059 с	58 мин	6.5 лет	3855 стол.	$2 \cdot 10^8$ стол.	$1.3 \cdot 10^{13}$стол.
$n!$	3.6 с	771,5 стол.	$8 \cdot 10^{16}$стол	…	…	…

Задачи со сложностью O(1):

- вставка и удаление элемента в односвязном и двусвязном списке;
- добавление вершины или ребра в графе.

Задачи со сложностью O(log N):

- двоичный поиск в линейном упорядоченном массиве;

Задачи с полиномиальной сложностью:

- задача сортировки;
- задача поиска эйлерова цикла на графе;
- поиск некоторого слова в тексте длиной n символов;
- поиск кратчайшего пути на графе;
-линейное программирование.

Задачи экспоненциальной сложности:

- задача коммивояжёра, задача выполнимости булевых формул;
- построение всех подмножеств данного множества;
- задачи распознавания правильных выражений в несложных языках;
- составление расписаний и раскрасок.

Класс NP

- Задачи, которые нельзя отнести ни к классу P, ни к классу E.
- Задачи, которые недетерминированная машина Тьюринга может решить за полиномиальное время, тогда как для детерминированной машины Тьюринга полиномиальный алгоритм неизвестен.
- Для этих задач до сих пор не разработан эффективный (т.е. полиномиальный) алгоритм, но и не доказано, что таких алгоритмов не существует.
- К классу NP относятся все задачи, решение которых можно *проверить* за полиномиальное время. Оракул предлагает решения, которые после проверки верификатором за полиномиальное время приобретают «юридическую» силу.

Пример.

Дано n чисел $a_1,\ldots a_n$ и число V.

Задача: Найти вектор (массив) $X=(x_1,\ldots,x_n)$, $x_i \in \{0,1\}$, такой, что $\sum a_i x_i = V$.

Т.е. может ли быть представлено число V в виде суммы каких- либо элементов массива A.

Если какой-то алгоритм выдает результат – массив X, то проверка правильности этого результата может быть выполнена с полиномиальной сложностью: проверка $\sum a_i x_i = V$ требует не более $\Theta(N)$ операций.

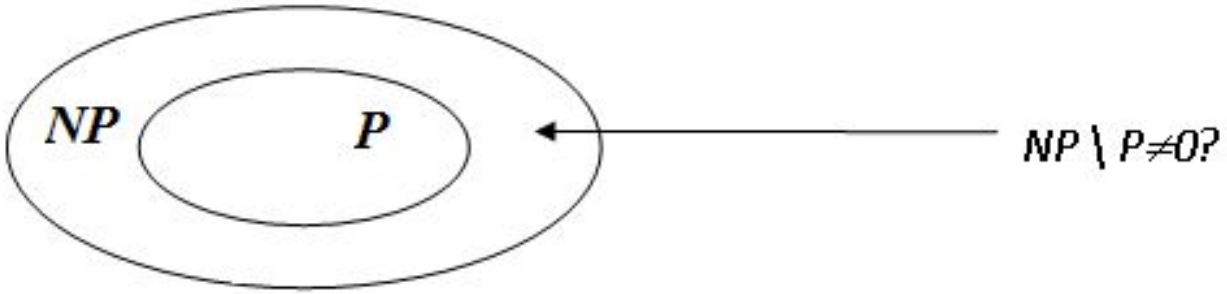
Проблема равенства классов P и NP.

Поскольку детерминированная машина Тьюринга может рассматриваться как специальный случай недетерминированной машины Тьюринга, в которой отсутствует стадия угадывания, а стадия проверки совпадает с ДМТ, класс NP включает в себя класс P, а также некоторые проблемы, для решения которых известны лишь алгоритмы, экспоненциально зависящие от размера входа (то есть неэффективные для больших входов).

Вопрос о равенстве этих двух классов считается одной из самых сложных открытых проблем в области теоретической информатики.

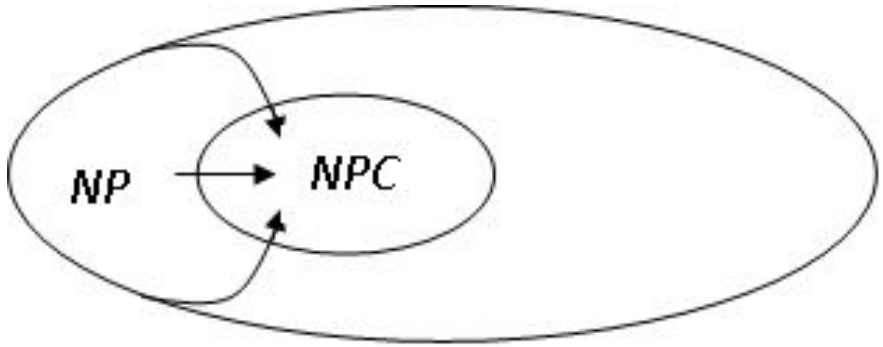
На сегодня отсутствуют теоретические доказательства как совпадения этих классов (P=NP), так и их несовпадения.

Предположение состоит в том, что класс P является собственным подмножеством класса NP, т.е. NP \ P не пусто.



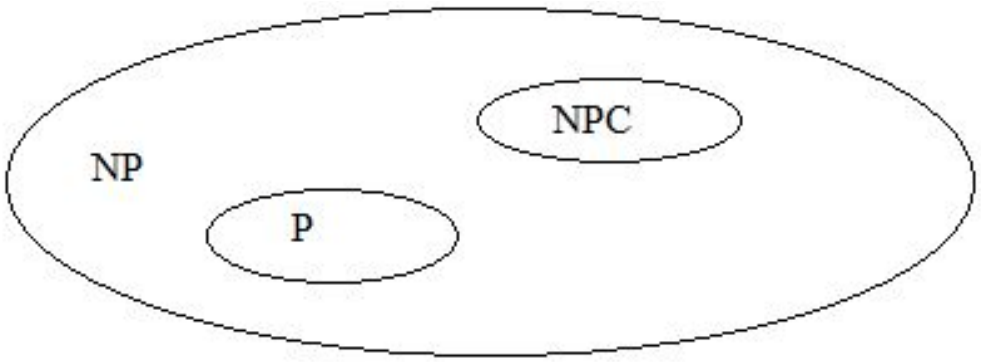
Класс *NPC* (*NP* – полные задачи)

Определение класса *NPC* (*NP*-complete) или класса *NP*-полных задач требует выполнения следующих двух условий: во-первых, задача должна принадлежать классу *NP,* и, во-вторых, к ней полиномиально должны сводиться все задачи из класса *NP*



Для класса *NPC* доказана следующая **теорема**: если существует задача, принадлежащая классу *NPC*, для которой существует полиномиальный алгоритм решения, то класс *P* совпадает с классом *NP*, т.е. *P=NP*

В настоящее время доказано существование сотен NP–полных задач, но ни для одной из них пока не удалось найти полиномиального алгоритма решения. В настоящее время исследователи предполагают следующее соотношение классов:



Примеры *NP* – полных задач

К этому классу относятся следующие задачи:

- задача о выполнимости: существует ли для данной булевской формулы, находящейся в КНФ, такое распределение истинностных значений, что она имеет значение истина?
- задача коммивояжера (Коммивояжер хочет объехать все города, побывав в каждом ровно по одному разу, и вернуться в город, из которого начато путешествие. Известно, что переезд из города i в город j стоит $c(i, j)$ рублей. Требуется найти путь минимальной стоимости.);
- решение систем уравнений с целыми переменными;
- составление расписаний, учитывающих определенные условия;
- размещение обслуживающих центров (телефон, телевидение, срочные службы) для максимального числа клиентов при минимальном числе центров;
- оптимальная загрузка емкости (рюкзак, поезд, корабль, самолёт) при наименьшей стоимости;
- оптимальный раскрой (бумага, картон, стальной прокат, отливки), оптимизация маршрутов в воздушном пространстве, инвестиций, станочного парка;

Пути решения NP-полных задач.

1. Поиск приближенного решения (например, использование жадных алгоритмов для решения задач о коммивояжёре, о рюкзаке);
2. Организация “разумной” стратегии перебора (например, метод ветвей и границ);
3. Сведение NP-полных задач друг к другу (например, сведение задачи коммивояжёра к задаче линейного программирования);
4. Выделение из общей NP-полной задачи эффективно разрешимых частных случаев.

Когда временными оценками можно пренебречь.

- Если создаваемая программа будет использована только несколько раз, тогда стоимость написания и отладки программы будет доминировать в общей стоимости программы.
- Если программа будет работать только с «малыми» входными данными, то степень роста времени выполнения будет иметь меньшее значение, чем константа, присутствующая в формуле времени выполнения.
- Эффективные, но сложные алгоритмы могут быть нежелательны, если готовые программы будут поддерживать лица, не имеющие достаточной квалификации для того, чтобы в них разобраться.
- Известно несколько примеров, когда эффективные алгоритмы требуют таких больших объемов машинной памяти, что этот фактор сводит на нет их преимущество.
- В численных алгоритмах точность и устойчивость не менее важны, чем их временная эффективность.

Вычисление времени выполнения не рекурсивных алгритмов

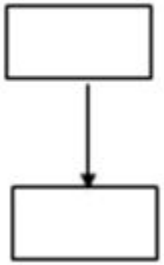
I. Нахождение функции трудоемкости по фактическому количеству элементарных операций.

В качестве «элементарных» операций алгоритма, представленного в формальной системе RAM будем использовать следующие:

1) простое присваивание: $a \leftarrow b$;
2) одномерная индексация *a[i]*;
3) арифметические операции: *(*, /, -, +);*
4) операции сравнения;
5) логические операции.

Анализ трудоемкости основных алгоритмических конструкций

А) Конструкция «Следование»

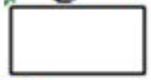


Трудоемкость конструкции есть сумма трудоемкостей блоков, следующих друг за другом.

$T_{«следование»} = t_1 + \ldots + t_k$, *где* k – количество блоков.

В) Конструкция «Ветвление»

if (logical expression) then



t_{then} с вероятностью p

else

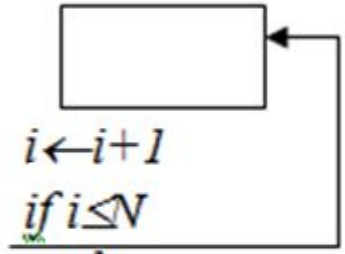
t_{else} с вероятностью $(1-p)$

Общая трудоемкость конструкции «Ветвление» требует анализа вероятности выполнения переходов на блоки «Then» и «Else» и определяется как:

$$T_{«ветвление»} = t_{logical\ expression} + t_{then} * p + t_{else} * (1-p).$$

C) Конструкция «Цикл»

for i ← 1 to N



i←i+1

if i≤N

end

После сведения конструкции к элементарным операциям ее трудоемкость определяется как:

$$T_{«цикл»} = 1+3*N+N*t_{«тела\ цикла»}$$

Примеры анализа простых алгоритмов

Пример 1. Задача суммирования элементов квадратной матрицы

Алгоритм выполняет одинаковое количество операций при фиксированном значении n, и следовательно является количественно-зависимым.

```
Sum ← 0
For i ← 0 to n-1
For j ← 0 to n-1
        Sum ← Sum + A[i][j]
end for
```

$T_A(n)=1+1+3n + n\,(1+3n+4n)=7\,n^2+4*n+2 = \Theta(n^2)$, заметим, что под n понимается линейная размерность матрицы, в то время как на вход алгоритма подается n^2 значений.

Пример 2. Задача поиска максимума в массиве

Данный алгоритм является количественно-параметрическим, поэтому для фиксированной размерности исходных данных необходимо проводить анализ для худшего, лучшего и среднего случая.

```
Max ← S[0]
For i ← 1 to n-1
    if Max < S[i]
      then     Max ← S[i]
end for
```

Худший случай

Максимальное количество переприсваиваний максимума (на каждом проходе цикла) будет в том случае, если элементы массива отсортированы по возрастанию. Трудоемкость алгоритма в этом случае равна:

$$T_A^{\wedge}(n)=2+1+3\,(n-1)+(n-1)\,(2+2)=7n-4=\Theta(n).$$

Лучший случай

- Минимальное количество переприсваивания максимума будет в том случае, если максимальный элемент расположен на первом месте в массиве. Трудоемкость алгоритма в этом случае равна:

$$T_a^{V}(n)=2+1+3\,(n-1)+2(n-1)=5n-2=\Theta(n).$$

Средний случай

Элементарное усреднение

$T_{cp}(n) = (T_a^{\vee}(n) + T_A^{\wedge}(n))/2 = 6n-3 = \Theta(n).$

Вероятностный подход

- Переприсваивание максимума при просмотре *к-го* элемента происходит, если в подмассиве из первых *к* элементов максимальным элементом является последний. В случае равномерного распределения вероятность этого равна *1/к*. Тогда в массиве из *n* элементов математическое ожидание среднего количества операций присваивания определяется как:

$$\sum_{k=1}^{N} 1/k = Hn \approx Ln(N) + \gamma, \quad \gamma \approx 0{,}57$$,где γ - постоянная Эйлера

$$\gamma = \lim_{n\to\infty}\left(\sum_{k=1}^{n}\frac{1}{k} - \ln n\right) = \lim_{n\to\infty}\left(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} - \ln n\right)$$

$\gamma \approx 0{,}577\,215\,664\,901\,532\,860\,606\,512\,090\,082\,402\,431\,042\,159\,335\,939\,923$
$598\,805\,767\,234\,884\,867\,726\,777\,664\,670\,936\,947\,063\,291\,746\,749\,515\ldots$

Величина *Hn* называется *n-ым* гармоническим числом.

- ***$T_a(n) = 2+1+3(n-1) + 2(n-1)+2(Ln(n) + \gamma)) = 5n-2+2Ln(n)+2\gamma = \Theta(n).$***

Некоторые математические формулы, необходимые для анализа алгоритмов.

Логарифмы

$$\log_B 1 = 0;$$

$$\log_B B = 1;$$

$$\log_B(XY) = \log_B X + \log_B Y;$$

$$\log_B X^Y = Y \log_B X;$$

$$\log_A X = \frac{\log_B X}{\log_B A};$$

$$A^{\log_B C} = C^{\log_B A};$$

$$\log_A B = \frac{1}{\log_B A}.$$

Формулы суммирования

$$\sum_{i=1}^{N} Ci = C \sum_{i=1}^{N} i$$

$$\sum_{i=L}^{N} i = \sum_{i=0}^{N-L} (i + L)$$

$$\sum_{i=L}^{N} i = \sum_{i=0}^{N} i - \sum_{i=0}^{L-1} i$$

$$\sum_{i=1}^{N} (A + B) = \sum_{i=1}^{N} A + \sum_{i=1}^{N} B$$

$$\sum_{i=0}^{N} (N - i) = \sum_{i=0}^{N} i$$

$$\sum_{i=1}^{N} 1 = N$$

$$\sum_{i=1}^{N} C = CN$$

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

$$\sum_{i=1}^{N} A^i = \frac{A^{N+1} - 1}{A - 1} \text{ для любого числа } A$$

$$\sum_{i=1}^{N} i2^i = (N-1)2^{N+1} + 2$$

$$\sum_{i=1}^{N} \frac{1}{i} \approx \ln N$$

$$\sum_{i=1}^{N} \log_2 i \approx N \log_2 N - 1.5$$

Математическое ожидание дискретной случайной величины.

Если известен закон распределения случайной величины x, то есть известно, что случайная величина x может принимать значения x_1, x_2, ..., x_k с вероятностями p_1, p_2, ..., p_k, тогда математическое ожидание M_x случайной величины x равно

$$M_x = \sum_{i=1}^{k} p_i x_i$$

Если случайная величина x имеет дискретное равномерное распределение:

$$\mathrm{P}(\mathrm{x} = x_i) = \frac{1}{n},\ i = 1, \ldots, n$$

, то её математическое ожидание равн $\mathrm{Mx} = \frac{1}{n}\sum_{i=1}^{n} x_i$

Свойства математического ожидания:

Математическое ожидание постоянной равно этой постоянной $M_c=C$

Математическое ожидание суммы случайных величин равно сумме их математических ожиданий: $M_{x+y}=M_x+M_y$

Математическое ожидание произведения независимых случайных величин равно произведению математических ожиданий этих величин: $M_{x*y}=M_x*M_y$

II. Нахождение вида функции трудоемкости без подсчета фактической стоимости команд.

Пример анализа алгоритма сортировки «вставками»:

Сортировка вставками	*Стоимость*	*Количество повторений*
for (i=0; i < n; i++)	c_1	n
{ x = a[i];	c_2	n-1
j=i-1;	c_3	n-1
while (j>=0 && a[j] > x)	c_4	$\sum_{i=1}^{n-1} t_i$
{		
a[j+1] = a[j];	c_5	$\sum_{i=1}^{n-1} (t_i - 1)$
j--;	c_6	
}		
a[j+1] = x;	c_7	n-1
}		

где t_i — количество проверок условия во внутреннем цикле for.

Время работы алгоритма сортировки вставками — это сумма времён работы каждого шага.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n-1} t_i + c_5 \sum_{i=1}^{n-1} (t_i - 1) + c_6 \sum_{i=1}^{n-1} (t_i - 1) + c_7(n-1)$$

Анализ наилучшего случая.

На вход подается уже отсортированный массив. При этом все внутренние циклы состоят всего из одной итерации, то есть $t_i = 1$ для всех i. Тогда время работы алгоритма составит:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$
$$= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

Таким образом, в лучшем случае время T(n), необходимое для сортировки массива размера n, является линейной функцией от n и имеет вид $T(n)=an+b$ для некоторых констант a и b.

Анализ наихудшего случая.

Входной массив, отсортирован в обратном порядке. При этом каждый a[i] элемент сравнивается со всеми уже отсортированными элементами a[0]…a[i-1]. Это означает, что все внутренние циклы состоят из i итераций, то есть t_i =i для всех i. Тогда время работы алгоритма составит:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \qquad \sum_{i=1}^{n-1} (i-1) = \frac{n(n-3)}{2} + 1$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\frac{n(n-1)}{2} + c_5\left(\frac{n(n-3)}{2} + 1\right)$$

$$+ c_6\left(\frac{n(n-3)}{2} + 1\right) + c_7(n-1)$$

$$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 - \frac{c_4}{2} - \frac{3c_5}{2} - \frac{3c_6}{2} + c_7\right)n - (c_2$$

$$+ c_3 - c_5 - c_6 + c_7)$$

Теперь функция T(n) квадратичная и имеет вид $T(n)=an^2+bn+c$

Анализ среднего случая.

Характер поведения "усредненного" времени работы часто ничем не лучше поведения времени работы для наихудшего случая. Предположим, что последовательность, к которой применяется сортировка методом вставок, сформирована случайным образом. Сколько времени понадобится, чтобы определить, в какое место подмассива a[1..i-1] следует поместить элемент a[i]? Предположим, что в среднем половина элементов этого подмассива меньше, чем a[i], а половина — больше его. Таким образом, в среднем нужно проверить половину элементов подмассива a[1..i-1], поэтому t_i приблизительно равно i/2.

$$\sum_{i=1}^{n-1} \frac{i}{2} = \frac{n(n-1)}{4} \qquad \sum_{i=1}^{n-1} \frac{(i-1)}{2} = \frac{n(n-3)}{4} + \frac{1}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \frac{n(n-1)}{4} + c_5\left(\frac{n(n-3)}{4} + \frac{1}{2}\right) + c_6\left(\frac{n(n-3)}{4} + \frac{1}{2}\right) + c_7(n-1) =$$

$$= \left(\frac{c_4}{4} + \frac{c_5}{4} + \frac{c_6}{4}\right)n^2 + \left(c_1 + c_2 + c_3 - \frac{c_4}{4} - \frac{3c_5}{4} - \frac{3c_6}{4} + c_7\right)n - (c_2 + c_3 - \frac{c_5}{2} - \frac{c_6}{2} + c_7)$$

В результате получается, что среднее время работы алгоритма является квадратичной функцией от количества входных элементов, т.е. характер этой зависимости такой же, как и для времени работы в наихудшем случае.

III. Оценивание порядков роста времени работы.

Правила вычисления времени выполнения программ

Правило 1. Правило сумм

Пусть $T_1(n)$ и $T_2(n)$ – время выполнения двух последовательных фрагментов программы P_1 и P_2 соответственно. Пусть $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$. Тогда $T_1(n) + T_2(n) = O(max(f(n), g(n)))$.

Доказательство

$T_1(n) = O(f(n))$, следовательно, существуют такая константа c_1 и натуральное число n_1, что $T_1(n) \le c_1 \cdot f(n)$ для любого $n \ge n_1$.

$T_2(n) = O(g(n))$, следовательно, существуют такая константа c_2 и натуральное число n_2, что $T_2(n) \le c_2 \cdot g(n)$ для любого $n \ge n_2$.

Пусть $n_0 = max(n_1, n_2)$. Если $n \ge n_0$, то, очевидно, что

$T_1(n)+T_2(n) \le c_1 \cdot f(n)+c_2 \cdot g(n) \Rightarrow T_1(n)+T_2(n) \le (c_1+c_2) \cdot max(f(n), g(n)) \Rightarrow$

$T_1(n)+T_2(n) \le c \cdot max(f(n), g(n)) \Rightarrow T_1(n)+T_2(n) = O(max(f(n), g(n)))$.

Следствие. Из правила сумм также следует, что если f(n) < g(n) для всех n, превышающих n_0, то выражение O(f(n) + g(n)) эквивалентно O(g(n)). Например, $O(n^2 + n)$ то же самое, что $O(n^2)$.

Правило 2. Правило произведений

Пусть $T_1(n)$ и $T_2(n)$ – время выполнения двух вложенных фрагментов программы P_1 и P_2 соответственно. Пусть $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$. Тогда $T_1(n) \cdot T_2(n) = O(f(n) \cdot g(n))$.

Доказательство

$T_1(n) = O(f(n))$, следовательно, существуют такая константа c_1 и натуральное число n_1, что $T_1(n) \le c_1 \cdot f(n)$ для любого $n \ge n_1$.

$T_2(n)=O(g(n))$, следовательно, существуют такая константа c_2 и натуральное число n_2, что $T_2(n) \le c_2 \cdot g(n)$ для любого $n \ge n_2$.

Пусть $n_0 = max(n_1, n_2)$. Если $n \ge n_0$, то очевидно:

$T_1(n) \cdot T_2(n) \le c_1 \cdot f(n) \cdot c_2 \cdot g(n) \Rightarrow T_1(n) \cdot T_2(n) \le (c_1 \cdot c_2) \cdot (f(n) \cdot g(n)) \Rightarrow$

$T_1(n) \cdot T_2(n) \le c \cdot (f(n) \cdot g(n)) \Rightarrow T_1(n) \cdot T_2(n) = O(f(n) \cdot g(n))$

Следствие.

$O(c \cdot f(n))$ эквивалентно $O(f(n))$, если $c>0$.

Правило 3. Время выполнения операторов присваивания, чтения, записи, сравнения обычно пропорционально единице, т. е. равно $O(1)$.

Правило 4. Время выполнения последовательности операторов определяется с помощью правила сумм, следовательно, равно наибольшему времени выполнения оператора в данной последовательности.

Правило 5. Время выполнения условных операторов состоит из времени условно исполняемых операторов и времени вычисления самого логического выражения, т. е. $O(if\text{–}then\text{–}else)=O(if)+O(max(then, else))$.

Правило 6. Время выполнения цикла является суммой времени всех исполняемых итераций цикла, в свою очередь состоящих из времени выполнения операторов тела цикла и времени вычисления условия прекращения цикла. Часто время выполнения цикла вычисляется, пренебрегая определением констант пропорциональности, как произведение количества выполненных итераций цикла на наибольшее возможное время выполнения операторов тела цикла.

Правило 6.

Для программ, содержащих несколько нерекурсивных процедур, можно подсчитать общее время выполнения программы путем последовательного нахождения времени выполнения процедур, начиная с той, которая не имеет вызовов других процедур. (Так как мы предположили, что все процедуры нерекурсивные, то должна существовать хотя бы одна такая процедура). Затем можно определить время выполнения процедур, вызывающих эту процедуру, используя уже вычисленное время выполнения вызываемой процедуры. Продолжая этот процесс, найдем время выполнения всех процедур.

Правило 7.

Для рекурсивных процедур действуют иные правила.