Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
«Вятский государственный гуманитарный университет»

Дополнительная подготовка школьников по дисциплине «Информатика и информационные технологии»

Учебный модуль
Сложность алгоритмов

Е. В. Разова

Киров
2011

СОДЕРЖАНИЕ

1. Понятие сложности

С развитием вычислительной техники было создано огромное количество разнообразных алгоритмов в различных прикладных областях, и пришлось обратить серьезное внимание на вопросы их эффективности. Так как ресурсы памяти и времени работы ЭВМ ограничены, недостаточно знать, что существует алгоритм решения данной задачи. Нужно хотя бы в общих чертах представлять, какие ресурсы понадобятся ЭВМ для реализации алгоритма, сможет ли соответствующая программа поместиться в памяти и даст ли она результаты в приемлемое время. Исследования этих вопросов создали новый раздел теории алгоритмов – теорию сложности алгоритмов.

Сложность алгоритма – количественная характеристика, которая говорит о том, сколько времени он работает (временна́я сложность) либо о том, какой объем памяти требуется для его работы (емкостная сложность).

Емкостная сложность алгоритма определяется числом ячеек памяти, используемых в процессе его выполнения.

Развитие технологии привело к тому, что память стала дешевой и компактной. Как правило, она не является критическим ресурсом – ее вполне достаточно для решения задачи. Поэтому чаще всего под анализом сложности алгоритма понимают исследование его временно́й сложности. Далее под сложностью будем понимать именно временну́ю сложность, ее еще называют *трудоемкостью* алгоритма. В каких единицах измерять сложность алгоритмов? Понятно, что обычные единицы измерения времени (секунды и т.д.) здесь не подходят – одна и та же программа при одних и тех же входных данных на разных компьютерах будет выполняться, вообще говоря, разное время.

Физическое время выполнения алгоритма – это величина $\tau \cdot t$, где t – число действий (элементарных шагов, команд), а τ – среднее время выполнения одного элементарного действия.

Число t определяется описанием алгоритма и не зависит от физической реализации модели, а τ зависит от скорости обработки сигналов в элементах и узлах ЭВМ. Поэтому объективной математической характеристикой временно́й сложности алгоритма является число элементарных действий, выполняемых в ходе работы алгоритма. Однако далеко не всегда ясно, какие операции следует считать элементарными. Кроме того, разные операции требуют для своего выполнения разного времени, да и перевод операций, используемых в описании алгоритма, в операции, используемые в компьютере, – дело очень неоднозначное, зависящее от таких, например,

факторов, как свойства компилятора и квалификация программиста. Поэтому утверждение, что такой-то алгоритм при таких-то входных данных требует 150 000 000 операций, фактически не несет информации о реальном времени вычислений. На самом деле задача анализа сложности алгоритма состоит в исследовании того, как меняется время работы при увеличении объема входных данных. Оно, конечно, растет, но с какой скоростью?

Временну́ю сложность алгоритма выражают функцией $T(n)$ зависимости времени работы (числа элементарных действий) от размера входа. Размер входа определяется для каждой задачи индивидуально. Обычно у задачи есть какой-нибудь естественный параметр, характеризующий объем входных данных, и сложность оценивается по отношению к этому параметру. Например:

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

2) в задачах обработки двумерных массивов размером входа так же является количество элементов в массиве, но часто бывает полезно выразить это значение через количество строк и столбцов массива;

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.

К элементарным действиям, определяющим временну́ю сложность алгоритма следует отнести прежде всего операции сравнения и присваивания.

Пример 1. Алгоритм сложения n чисел $x_1, \ldots, x_n$.

```
Sum:=0;
For i:=1 To n Do
     Sum:=Sum + x[i];
```

Если числа не слишком велики и сумма может быть вычислена с использованием стандартной операции сложения, то можно измерять объем входных данных просто числом слагаемых. Считая элементарной операцией сложение, приходим к выводу, что трудоемкость алгоритма равна n. Если же складываются очень большие числа, то для сложения нужно использовать специальные программы («длинную арифметику») и объем входных данных

правильнее измерять максимальной длиной представления слагаемых (например, в битах). Эта мера в любом случае более точная, но с ней и сложнее работать, а иногда эта точность излишняя.

Пример 2. Алгоритм линейного поиска.

```
i:=1;
While (i<=n) And (A[i]<>x) Do
     i:=i+1;
If i<=n Then WriteLn('Искомый элемент с номером ', i)
      Else WriteLn('Искомого элемента в массиве нет');
```

Представительной операцией можно считать операцию сравнения. Число выполняемых сравнений в худшем случае (если элемента нет в массиве) будет равно n+1, а в лучшем (если x в массиве стоит на первом месте) – 1. Но допустим, что массив x представляет собой перестановку чисел 1, 2, ..., n, что все перестановки равновероятны и что искомый элемент обязательно находится в массиве, т.е. представляет собой целое число между 1 и n. Тогда в среднем будет выполняться $\frac{n}{2}$ сравнений. Это наиболее распространенные меры сложности – трудоемкость в худшем, в среднем и в лучшем случаях. Иногда они различаются гораздо сильнее, чем в этом примере.

2. Асимптотические обозначения степени роста. Ограниченность показателя степени роста

Как было сказано ранее, время работы алгоритма в худшем и в лучшем случаях может сильно отличаться. Большей частью нас будет интересовать время работы в худшем случае, которое определяется как максимальное время, так как

1) зная время работы в худшем случае можно гарантировать, что при любом входе работа будет длиться не дольше;

2) на практике худший случай встречается не так уж редко (например, поиск несуществующего элемента);

3) часто время работы в среднем случае часто оказывается ближе к оценке худшего случая.

Для сравнения алгоритмов по скорости (степени) роста времени работы введены асимптотические обозначения.

Определение 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: $T(n) = \Theta(g(n))$. Читается как «Тэта большое от g от n».

$$T(n) = \Theta(g(n)),\ \textit{если}\ \exists n_0 \in N, 0 < c_1 \le c_2 :$$
$$\forall n \ge n_0 \quad c_1 \cdot g(n) \le T(n) \le c_2 \cdot g(n)$$

Замечание. Говорят, что время работы алгоритма $T(n)$ имеет порядок роста $g(n)$ ($T(n) = \Theta(g(n))$), если и сверху и снизу время работы ограничено функцией с одной и той же степенью роста, если это не так, то говорят о верхних (оценка худшего случая) и нижних (оценка лучшего случая) оценках.

Определение 2. Говорят, что время работы алгоритма $T(n)$ имеет нижнюю оценку $g(n)$, если существуют натуральное число n_0 и положительная константа c, такие, что для любого натурального n начиная с n_0 выполняется неравенство $c \cdot g(n) \le T(n)$. Обозначение: $T(n) = \Omega(g(n))$. Читается как «Омега большое от g от n».

$$T(n) = \Omega(g(n)),\ если\ \exists n_0 \in N, c > 0:$$
$$\forall n \geq n_0 \quad c \cdot g(n) \leq T(n)$$

Определение 3. Говорят, что время работы алгоритма $T(n)$ имеет верхнюю оценку $g(n)$, если существуют натуральное число n_0 и положительная константа c, такие, что для любого натурального n начиная с n_0 выполняется неравенство $T(n) \leq c \cdot g(n)$. Обозначение: $T(n) = O(g(n))$. Читается как «О большое от g от n».

$$T(n) = O(g(n)),\ если\ \exists n_0 \in N, c > 0:$$
$$\forall n \geq n_0 \quad T(n) \leq c \cdot g(n)$$

Таким образом, для линейного поиска (см. пример 2) справедливы следующие оценки: $T(n) = \Omega(1)$ и $T(n) = O(n)$.

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. $c \geq 5$.

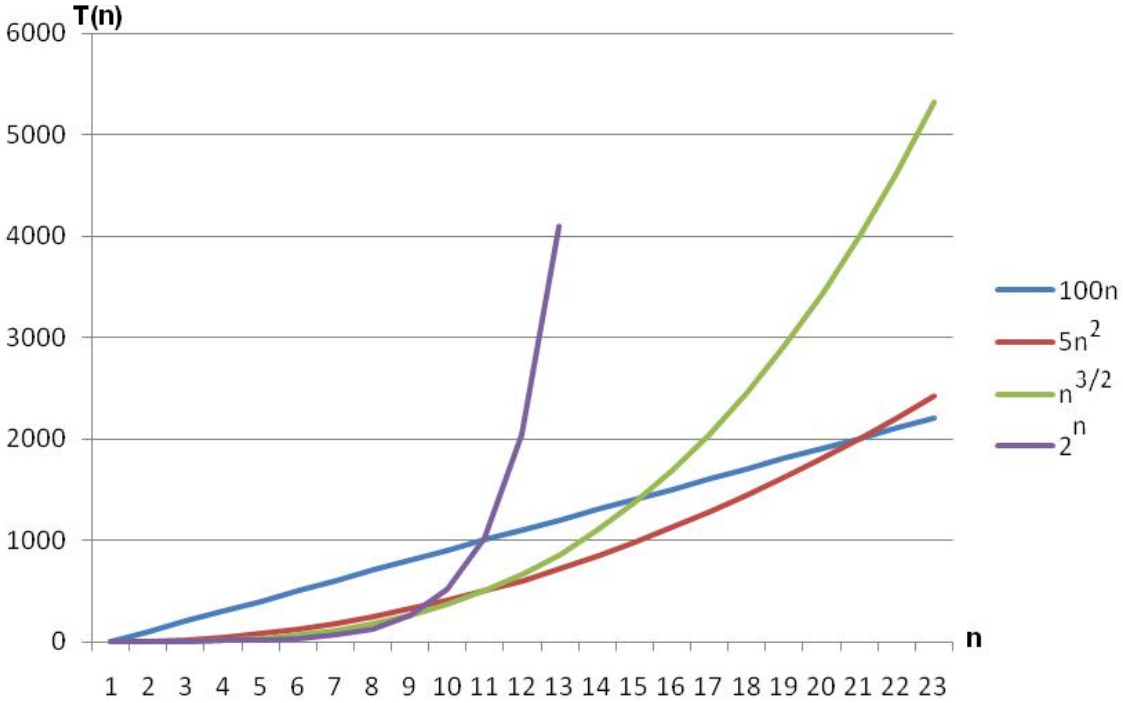
Таким образом, найдены натуральное число $n_0=1$ и положительная константа $c=5$, такие, что для любого натурального n начиная с n_0 выполняется неравенство $3 \cdot n^3 + 2 \cdot n^2 \leq c \cdot n^3$, следовательно, $T(n)=O(n^3)$.

Ограниченность показателя степени роста

Итак, мы предположили, что программы можно оценить с помощью функций времени работы, принебрегая при этом константами пропорциональности. С этой точки зрения программа с временем работы $O(n^2)$ лучше программы с временем работы $O(n^3)$. Однако константы пропорциональности зависят не только от используемых компилятора и компьютера, но и от свойств самой программы.

Пусть одна программа выполняется за $100 \cdot n^2$ миллисекунд, а вторая – за $5 \cdot n^3$ миллисекунд. Может ли вторая программа быть предпочтительней первой? Ответ зависит от размера входа: при $n<20$ предпочтительнее вторая программа, а во всех остальных случаях предпочтительнее будет первая программа. Таким образом, чем меньше степень роста, тем больше размер задачи, которую можно решить за фиксированный промежуток времени:

$T(n)$	Максимальный размер задачи, решаемой за данное время		Рост размера задачи (в количестве раз)
	10^3 секунд	10^4 секунд	
$100 \cdot n$	10	100	10
$5 \cdot n^2$	14	45	3,2
$n^3/2$	12	27	2,3
2^n	10	13	1,3



Функции, часто встречающиеся при анализе алгоритмов:

– $\log n$ (логарифмическое время),
– n (линейное время),
– $n \log n$,
– n^2 (квадратичное время),
– 2^n (экспоненциальное время).

Первые четыре функции имеют невысокую скорость роста и алгоритмы, время работы которых оценивается этими функциями, можно считать быстродействующими. Скорость роста экспоненциальной функции иногда характеризуют как «взрывную».

Полиномиальным алгоритмом (или алгоритмом полиномиальной временной сложности) называется алгоритм, временная сложность которого равна $O(p(n))$, где $p(n)$ – некоторая полиномиальная функция, а n – входная длина.

Однако не всякая задача может быть решена за полиномиальное время.

Примеры.

1. Задача об укладке рюкзака.

Имеется n предметов, для каждого из которых известны стоимость C_i и объем V_i . Необходимо определить какие предметы необходимо уложить в рюкзак, чтобы их суммарный объем не превышал допустимый объем V рюкзака, а их общая стоимость C была наибольшей.

При условии, что каждый из n предметов может быть упакован или не упакован в рюкзак, всего будет 2^n вариантов взятых предметов.

2. Задача о коммивояжере.

Даны n городов и известны расстояния между каждыми двумя городами; коммивояжёр, выходящий из какого-нибудь города, должен посетить n–1 других городов и вернуться в исходный. В каком порядке ему нужно посещать города (по одному разу каждый), чтобы общее пройденное расстояние было минимальным?

Решение задачи коммивояжера сводится в худшем случае к перебору всех возможных маршрутов, а их всего существует $(n–1)!$. Эта зависимость не может быть выражена полиномиальной функцией, причем можно заметить, что $n! \geq 2^n$.

Задачи, для решения которых не существует полиномиального алгоритма, но существует экспоненциальный алгоритм ($O(a^n)$), называют *труднорешаемыми*.

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^5	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.	$2\cdot10^8$ стол.	$1.3\cdot10^{13}$стол.
$n!$	3.6 с	771,5 стол.	$8\cdot10^{16}$стол	…	…	…

Таким образом, понятие полиномиально разрешимой задачи принято считать уточнением идеи «практически разрешимой» задачи.

3. 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

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

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

Пример 1. Пусть имеются два фрагмента со временем выполнения $O(f(n))$ и $O(g(n))$, где

$$f(n)=\begin{cases} n^4, \text{если } n-\text{четно} \\ n^2, \text{если } n-\text{нечетно} \end{cases}, \quad f(n)=\begin{cases} n^2, \text{если } n-\text{четно} \\ n^3, \text{если } n-\text{нечетно} \end{cases}.$$

Так как $T(n) = O(max(f(n), g(n)))$, то $T(n)=O\left(\begin{cases} n^4, \text{если } n-\text{четно} \\ n^3, \text{если } n-\text{нечетно} \end{cases}\right)$.

Следствие. Если $g(n) \le f(n)$ для всех $n \ge n_0$, то $O(f(n)+g(n)) = O(f(n))$.

Пример 2. $O(n^2+n) = O(n^2)$, т. е. слагаемыми, имеющими меньший порядок рост, в силу правила сумм можно пренебречь.

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 $c>0$.

Пример. $O(n^2/2)=O(n^2)$, т. е. постоянным множителем, в силу правила произведений можно пренебречь.

Однако существуют ситуации, когда постоянный множитель следует учитывать. Это бывает, когда сравниваются разные алгоритмы для одной задачи.

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

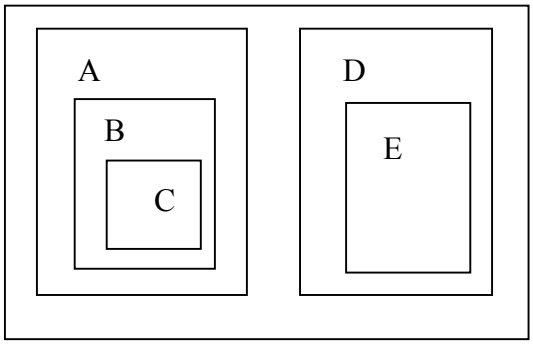
Правило 4. Время выполнения последовательности операторов определяется с помощью правила сумм, следовательно, равно наибольшему времени выполнения оператора в данной последовательности.

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

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 7. Вызов процедур.

Проиллюстрируем на примере.



Из рисунка видно, что в основной программе вызываются процедуры *A* и *D*. В свою очередь, в процедуре *A* вызывается процедура *B*, а в *B* – *C*. Из процедуры *D* вызывается процедура *E*.

Определять время работы такой программы следует в порядке, обратном порядку вызова процедур, а именно:

1) $T(C) \rightarrow T(B) \rightarrow T(A)$;
2) $T(E) \rightarrow T(D)$;
3) $T(A) + T(D)$.

Правило 8. Анализ времени работы рекурсивной процедуры.

Если есть рекурсивные процедуры, то нельзя упорядочить все процедуры таким образом, чтобы каждая процедура вызывала только процедуры, время выполнения которых подсчитано на предыдущем шаге. В этом случае необходимо с каждой рекурсивной процедурой связать временную функцию $T(n)$, где n определяет объем аргументов процедуры. Затем необходимо получить рекуррентное соотношение для $T(n)$, т. е. уравнение для $T(n)$, где участвуют значения $T(k)$ для различных значений k.

4. Анализ временной сложности нерекурсивных алгоритмов

Оценка временной сложности алгоритма по выше описанным правилам при неаккуратном их применении может дать слишком грубый результат. Как можно иначе организовать подсчет числа выполняемых элементарных операций?

Запишем алгоритм так, чтобы в каждой строке располагалось по одной команде. Пронумеруем строки с командами (строки, содержащие только операторные скобки не нумеруем). Так как в каждую команду каждой строки заложен фиксированный (постоянный) набор элементарных действий (присваивание, сравнение, обращение к ячейке памяти и т. д.) введем понятие стоимости одного выполнения каждой строки. Обозначим через константу c_i стоимость одного выполнения i-й строки.

Определим количество выполнений каждой строки в ходе исполнения алгоритма n_i. Тогда временная сложность алгоритма будет выразаться функцией $T(n)=c_1 \cdot n_1 + c_2 \cdot n_2 + \ldots + c_k \cdot n_k$.

4.1. Анализ линейного поиска

Запишем алгоритм так, чтобы в каждой строке располагалось по одной команде. Пронумеруем строки. Обозначим через константу c_i стоимость одного выполнения i-й строки.

Определим количество выполнений каждой строки в ходе исполнения алгоритма.

Команды строк 1, 4, 5, 6 выполняются в ходе работы алгоритма 1 раз. Для 2-й (заголовка цикла *While*) и 3-й (команда тела цикла) строк нельзя однозначно сказать, сколько раз они будет выполняться на i-й итерации внешнего цикла, так как это зависит от значения элементов в массиве. Поэтому формально число выполнений 2-й строки обозначим через t. А так как 3-я строка – это команда тела цикла, то она будет выполняться на один раз меньше, чем заголовок, следовательно, выполнится $t–1$ раз.

№ строки алгоритма	Алгоритм	Стоимость одного выполнения строки	Количество выполнений строки
1.	`i:=1;`	c_1	1
2.	`While (i<=n) And (A[i]<>x) Do`	c_2	t
3.	`I:=i+1;`	c_3	t–1
4.	`If i<=n`	c_4	1
5.	`Then WriteLn('Yes')`	c_5	1
6.	`Else WriteLn('No');`	c_6	1

Тогда время работы алгоритма (суммарное количество элементарных операций) будет определяться выражением

$$T(n) = c_1 + c_2 \cdot t + c_3 \cdot (t-1) + c_4 + c_5 + c_6 = (c_2 + c_3) \cdot t + (c_1 - c_3 + c_4 + c_5 + c_6).$$

Так как функция времени работы алгоритма $T(n)$ должна зависеть только от n, то необходимо избавиться от формально введенных t_i. Для этого рассмотрим работу алгоритма в лучшем и худшем (с точки зрения числа выполняемых операций) случаях.

Лучший случай – самый благоприятный случай, когда искомый элемент стоит на первом месте, тогда условие цикла *While* (2-я строка алгоритма) не выполняется уже в ходе первой проверки, следовательно, t=1, тогда

$$T_{л.с.}(n) = c_1 + c_2 \cdot 1 + c_3 \cdot (1-1) + c_4 + c_5 + c_6 = c_1 + c_2 + c_4 + c_5 + c_6 =$$

$$= A - константная \quad зависимость.$$

Худший случай – элемента в массиве нет, тогда заголовок цикла *While* выполнится $n+1$ раз, следовательно, $t=n+1$. Тогда

$$T(n) = c_1 + c_2 \cdot (n+1) + c_3 \cdot n + c_4 + c_5 + c_6 =$$

$$= (c_2 + c_3) \cdot n + (c_1 + c_2 + c_4 + c_5 + c_6) =$$

$$= A \cdot n + B - линейная \quad зависимость.$$

Таким образом, в соотвтетствии с правилами 1 и 2, время работы алгоритма линейного поиска имеет нижнюю оценку $\Omega(1)$ и верхнюю оценку $O(n)$.

4.2. Анализ сортировки вставками

Запишем алгоритм так, чтобы в каждой строке располагалось по одной команде. Обозначим через константу c_i стоимость одного выполнения i-й строки.

Определим количество выполнений каждой строки в ходе исполнения алгоритма.

Заголовок цикла проверяется на один раз больше, чем команда внутри цикла, поэтому 1-я строка выполняется n раз, а команды строк 2, 3, 7 – (n–1) раз, так как содержатся в цикле и не зависят ни от каких условий. Для 4-й строки (заголовка цикла *While*) нельзя однозначно сказать, сколько раз она будет выполняться на i-й итерации внешнего цикла, так как это зависит от расположения элементов в массиве. Поэтому формально это число обозначим через t_i.

№ строки алгоритма	Алгоритм	Стоимость одного выполнения строки	Количество выполнений строки
1.	`For i:=2 To n Do Begin`	c_1	n
2.	`  x:=A[i];`	c_2	$n-1$
3.	`  j:=i-1;`	c_3	$n-1$
4.	`  While (j>0) And (A[j]>x) Do Begin`	c_4	$\sum_{i=2}^{n} t_i$
5.	`    A[j+1]:=A[j];`	c_5	$\sum_{i=2}^{n} (t_i - 1)$
6.	`    j:=j-1`	c_6	$\sum_{i=2}^{n} (t_i - 1)$

7.	`  End;`
	`  A[j]:=x`
	`End;`
		c_7	$n-1$

Тогда время работы алгоритма будет определяться выражением

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \sum_{i=2}^{n} t_i + c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7 \cdot (n-1).$$

Так как функция времени работы алгоритма $T(n)$ должна зависеть только от n, то необходимо избавиться от формально введенных t_i. Для этого рассмотрим работу алгоритма в лучшем и худшем (с точки зрения числа выполняемых операций) случаях.

Лучший случай – самый благоприятный случай, когда массив уже отсортирован, тогда на каждой итерации внешнего цикла действие цикла *While* (4-я строка алгоритма) прекращается сразу же после первой проверки, следовательно, все t_i=1, тогда $\sum_{i=2}^{n} t_i = n-1$, $\sum_{i=2}^{n} (t_i - 1) = 0$, т. е.

$$T_{л.сл.}(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_5 \cdot 0 + c_6 \cdot 0 + c_7 \cdot (n-1) =$$
$$= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n + (-c_2 - c_3 - c_4 - c_7) =$$
$$= A \cdot n + B - \textit{линейная зависимость.}$$

Худший случай – массив отсортирован в обратном порядке, тогда на каждой *i*-й итерации внешнего цикла 4-я строка алгоритма выполняется *i* раз, следовательно, все $t_i = i$. Тогда

$$\sum_{i=2}^{n} t_i = \sum_{i=2}^{n} i = \frac{(2+n)\cdot(n-1)}{2} = \frac{n^2+n-2}{2},$$

$$\sum_{i=2}^{n} (t_i - 1) = \sum_{i=2}^{n} (i-1) = \frac{(1+(n-1))\cdot(n-1)}{2} = \frac{n^2-n}{2}, \text{ т. е.}$$

$$T_{х.сл.}(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \frac{n^2+n-2}{2} + c_5 \cdot \frac{n^2-n}{2} +$$
$$+ c_6 \cdot \frac{n^2-n}{2} + c_7 \cdot (n-1) =$$
$$= (c_4 + c_5 + c_6) \cdot n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7 \right) \cdot n +$$
$$+ (-c_2 - c_3 - c_4 - c_7) = A' \cdot n^2 + B' \cdot n + C' - \textit{квадратичная зависимость.}$$

Таким образом, в соответствии с правилами 1 и 2, время работы алгоритма сортировки вставками (метод прямого включения) имеет нижнюю оценку $\Omega(n)$ и верхнюю оценку $O(n^2)$.

5. Анализ рекурсивных алгоритмов. Решение рекуррентных соотношений

Время работы рекурсивных алгоритмов складывается:

1) из $D(n)$ – времени разбиения исходной задачи на подзадачи (действия на входе в рекурсию);

2) $T(n_1) + ... + T(n_k)$ – времени решения каждой подзадачи в отдельности (количество подзадач k определяется числом рекурсивных вызовов);

3) $S(n)$ – времени на соединение полученных решений (действия на выходе из рекурсии).

Рассмотрим пример анализа рекурсивного алгоритма на примере анализа сортировки слиянием.

5.1. Анализ сортировки слиянием

```
Type OdMas=Array[1..1000] Of Integer;
Var A : OdMas;
    n : Integer;
…
Procedure Sl(l, s, r : Integer);
Var B : OdMas;
    i, j, k : Integer;
Begin
k:=l; i:=l; j:=s; B:=A;
Repeat
 If A[i]<A[j] Then Begin B[k]:=A[i]; Inc(i) End
               Else Begin B[k]:=A[j]; Inc(j) End;
  Inc(k);
 Until (i>s) Or (j=r);
 If i>s Then
   For i:=j To r Do Begin B[k]:=A[i]; Inc(k) End
  Else For j:=i To s Do Begin B[k]:=A[j]; Inc(k) End;
 A:=B
End;

Procedure Sort_Sl(l, r : Integer);
Var s : Integer;
Begin
 If l<r Then Begin
  s:=(l+r) Div 2;
  Sort_Sl(l, s);
```

```
  Sort_Sl(s+1, r);
  Sl(l, s, r)
 End
End;

Begin {основная программа}
 …
 Sort_Sl(1, n);
 …
End.
```

Процедура *Sl* (процедура слияния отсортированных частей массива) – нерекурсивная процедура, следовательно, анализируется ранее рассмотренным методом. Временная сложность данной процедуры – $\Theta(n)$.

Процедура *Sort_Sl* (основная процедура сортировки слиянием) – рекурсивная процедура. Определим временную сложность данной процедуры.

Очевидно, что если массив состоит из одного элемента (n=1), то выполняется только проверка условия ($l<r$), следовательно, время работы пропорционально единице ($T(1)=\Theta(1)$).

Если в массиве более одного элемента ($n>1$), то выполняются следующие действия.

`If l<r Then Begin` `  s:=(l+r) Div 2;`	Действия на входе в рекурсию, разбиение исходной задачи на подзадачи, т. е. $D(n)= \Theta(1)$
`  Sort_Sl(l, s);` `  Sort_Sl(s+1, r);`	Два рекурсивных вызова. Задача разбивается на две подзадачи равной длины $T(n/2)+T(n/2)=2\cdot T(n/2)$
`  Sl(l, s, r)` `End`	Действия на выходе из рекурсии – соединение полученных решений (процедура слияния), т. е. $S(n)= \Theta(n)$

Таким образом, время работы данной процедуры определяется соотношением $T(n)=D(n)+T(n/2)+T(n/2)+S(n)=2\cdot T(n/2)+\Theta(1)+\Theta(n)$.

Так как сложность $\Theta(1)$ меньше сложности $\Theta(n)$, то с учетом подбора констант пропорциональности слагаемое $\Theta(1)$ можно отбросить (внести под знак $\Theta(n)$).

Таким образом, время работы сортировки слиянием определяется соотношением

$$T(n) = \begin{cases} 2 \cdot T(n/2) + \Theta(n), & \text{если } n > 1 \\ \Theta(1), & \text{если } n = 1 \end{cases}.$$

Далее необходимо решить полученное рекуррентное соотношение.

5.2. Способы решения рекуррентных соотношений

Рассмотрим три подхода к решению рекуррентных соотношений:

– метод подстановки;

– метод итераций.

1. Метод подстановки

Идея метода заключается в нахождении (угадывании) функции $f(n)$, такой, что для любого n ($n \geq 1$): $T(n) \leq f(n)$.

Для подбора функции выполняются следующие действия:

1) определяется вид функции $f(n)$, предполагая, что она зависит от некоторых пока неопределенных параметров;

2) подбираются значения параметров так, чтобы для всех $n \geq 1$ выполнялось $T(n) \leq f(n)$;

3) доказывается по индукции правильность подбора функции и значений параметров.

Решим рекуррентное соотношение, полученное для сортировки слиянием, данным методом. Определим верхнюю оценку (нижняя определяется аналогично).

Так как на каждом этапе рассматриваемая часть массива сокращается в два раза, а при слиянии выполняются действия, время которых пропорционально n, т. е. $S(n) = \Theta(n)$, то следует предполагать, что функция $f(n)$ – логарифмическая функция. Пусть $f(n) = a \cdot n \cdot \log_2 n$, где a – пока неопределенный параметр.

Должно выполняться неравенство $T(n) \leq f(n)$ при любом натуральном n. Проверим при n=1.

$T(n) = \Theta(1)$, т. е. существует константа c_1, такая, что при любом натуральном n справедливо неравенство $T(n) \leq c_1$.

$f(1) = a \cdot 1 \cdot \log_2 1 = 0$.

Таким образом, вид функции подобран неверно, так как не выполняется неравенство $T(1) \le f(1)$. Добавим еще один пока неопределенный параметр b следующим образом: $f(n) = a \cdot n \cdot \log_2 n + b$.

Проверим при $n=1$:

$f(1) = a \cdot 1 \cdot \log_2 1 + b = b \ge T(1) = c_1$ при $b \ge c_1$.

Докажем методом математической индукции справедливость для любого натурального n.

1) При $n=1$ справедливость неравенства доказана выше – база индукции.
2) Предположим, что для всех $k < n$ неравенство $T(k) \le a \cdot k \cdot \log_2 k + b$ выполняется – индуктивное предположение.
3) Докажем справедливость для n.

Так как $n>1$, то из рекуррентного соотношения следует, что $T(n) \le 2 \cdot T(n/2) + c_2 \cdot n$. Поскольку $n/2 < n$, то для $n/2$ выполняется индуктивное предположение, т. е.

$T(n) \le 2 \cdot (a \cdot (n/2) \cdot \log_2 (n/2) + b) + c_2 \cdot n =$

$= a \cdot n \cdot \log_2 n - a \cdot n + c_2 \cdot n + 2 \cdot b \le a \cdot n \cdot \log_2 n + b$, при $a \ge c_2 + b$.

Таким образом, при $b = c_1$, $a = c_1 + c_2$ для любого натурального n выполняется неравенство $T(n) \le (c_1 + c_2) \cdot n \cdot \log_2 n + c_1$, т. е. $T(n)$ имеет верхнюю оценку $O(n \cdot \log_2 n)$.

2. Метод итераций

Идея метода – итерирование рекуррентного соотношения (подстановка его самого в себя) и получение ряда, который следует оценить тем или иным способом. Этот способ всегда позволяет получить точное решение для $T(n)$, но сложен, так как часто приводит к решению в виде достаточно сложных сумм.

Проиллюстрируем идею этого метода на примере решения рекуррентного соотношения для сортировки слиянием, взяв правые части при раскрытии рекуррентных соотношений, т. е. определим верхнюю оценку сложности.

1-я итерация: $n > 1$, следовательно, $T(n) \le 2 \cdot T(n/2) + c_2 \cdot n$.

2-я итерация: раскроем $T(n/2)$, подставив $n/2$ вместо n в рекуррентное соотношение: $T(n) \le 2 \cdot (2 \cdot \mathrm{T}(n/4) + c_2 \cdot (n/2)) + c_2 \cdot n = 4 \cdot T(n/4) + 2 \cdot c_2 \cdot n$.

3-я итерация: $T(n) \le 4 \cdot (2 \cdot T(n/8) + c_2 \cdot (n/4)) + 2 \cdot c_2 \cdot n = 8 \cdot T(n/8) + 3 \cdot c_2 \cdot n$

и т. д.

Можно заметить закономерность:

i-ая итерация: $T(n) \le 2^i \cdot T(n/2^i) + i \cdot c_2 \cdot n$.

Предположим, что i-ая итерация последняя, т. е. $n/2^i = 1$, а $T(n/2^i) = T(1) \le c_1$.

Из равенства $n/2^i = 1$ следует $i = \log_2 n$.

Сделаем подстановку в неравенство i-й итерации:

$T(n) \le 2^{\log_2 n} \cdot c_1 + \log_2 n \cdot c_2 \cdot n = c_1 \cdot n + c_2 \cdot n \cdot \log_2 n$.

Так как $n \cdot \log_2 n$ растет быстрее, чем n, то $T(n) = O(n \cdot \log_2 n)$.

Аналогично можно определить нижнюю оценку сложности, взяв левые части неравенств при раскрытии асимптотических обозначений. $T(n) = \Omega(n \cdot \log_2 n)$.

Поскольку нижние и верхние оценки совпадают, то можно говорить о порядке роста $T(n) = \Theta(n \cdot \log_2 n)$.

5.3. Анализ быстрой сортировки (сортировки Хоара)

Проведем анализ временной сложности быстрой сортировки (сортировки Хоара).

Основная процедура сортировки имеет вид:

```
Procedure QuickSort(l, r : Integer);
Var x, i, j : Integer;
Begin
 If l<r Then Begin
   x:=A[(l+r) Div 2]; {выбор значения
барьерного элемента}
   i:=l; r:=j;
   Repeat
    {перестановка элементов: слева от
барьерного - элементы, меньшие его, а справа
- большие}
    While (A[i]<x) Do Inc(i);
    While (A[j]>x) Do Dec(j);
    If i<=j Do Begin
     Swap(A[i], A[j]);
     Inc(i); Dec(j)
    End
   Until i>j;
```

Действия на входе в рекурсию

```
  QuickSort(l, j); {рекурсивный вызов для
левой части}
  QuickSort(i, r); {рекурсивный вызов для
правой части}
```

Два рекурсивных вызова

```
End
End;
```

Время перестановки элементов в рассматриваемой части пропорционально длине этой части, т. е. в общем виде для части из n элементов $D(n) = \Theta(n)$.

Действий на выходе из рекурсии алгоритмом не предусмотрено, т. е. время соединения полученных решений $S(n) = \Theta(1)$.

В процедуре присутствует два рекурсивных вызова, для выполнения аналогичных действий для двух полученных в результате перестановки элементов частей массива (не обязательно равной длины), т. е. $T(n_1)$ и $T(n_2)$. Заметим, что $n_2 \approx n - n_1$.

Если в массиве один элемент, то происходит выход из процедуры сразу после проверки условия ($l<r$), т. е. $T(1) = \Theta(1)$.

Таким образом, получили рекуррентное соотношение вида:

$$T(n) = \begin{cases} T(n_1) + T(n_2) + \Theta(n) + \Theta(1), & n > 1 \\ \Theta(1), & n = 1 \end{cases}.$$

В общем случае $n_2 = n - n_1$.

Упростим данное рекуррентное соотношение с учетом свойств асимптотических обозначений:

$$T(n) = \begin{cases} T(n_1) + T(n - n_1) + \Theta(n), & n > 1 \\ \Theta(1), & n = 1 \end{cases}.$$

Решить данное рекуррентное соотношение рассмотренными выше методами не удается, так как в соотношении присутствует формально введенный параметр n_1. Для того чтобы «избавиться» от него, рассмотрим работу алгоритма в лучшем и в худшем случаях.

Лучший случай

Характеризуется делением рассматриваемой части массива при каждом рекурсивном вызове пополам. Таким образом, $n_1 = n/2$, $n - n_1 = n/2$, и

рекуррентное соотношение с учетом свойств асимптотических обозначений принимает вид:

$$T_{\text{лучший случай}}(n) = \begin{cases} 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n), & n > 1 \\ \Theta(1), & n = 1 \end{cases}.$$

Таким образом, получено рекуррентное соотношение, аналогичное рекуррентному соотношению сортировки слиянием, т. е. $T_{\text{лучший случай}}(n) = \mathrm{O}(n \cdot \log_2 n)$.

Худший случай

Худший случай для времени работы алгоритма характеризуется выбором в качестве барьерного элемента при каждом рекурсивном вызове максимального (или минимального) элемента рассматриваемой части. Тогда сортируемое множество элементов каждый раз разбивается на две части таким образом, что в одной части оказывается один элемент, а в другой – все остальные. Таким образом, $n_1 = 1$, $n - n_1 = n - 1$, и рекуррентное соотношение с учетом свойств асимптотических обозначений принимает вид:

$$T_{\text{худший случай}}(n) = \begin{cases} T(n-1) + \Theta(n), & n > 1 \\ \Theta(1), & n = 1 \end{cases}.$$

Найдем методом итераций значение верхней оценки, предварительно раскрыв асимптотические обозначения и взяв лишь их правые части.

$$T_{\text{худший случай}}(n) \le T(n-1) + c_1 \cdot n \le T(n-2) + c_1 \cdot (n-1) + c_1 \cdot n \le$$
$$\le T(n-3) + c_1 \cdot (n-2) + c_1 \cdot (n-1) + c_1 \cdot n \le \ldots$$
$$\ldots \le T(1) + c_1 \cdot 2 + \ldots + c_1 \cdot (n-2) + c_1 \cdot (n-1) + c_1 \cdot n \le$$
$$\le c_2 + c_1 \cdot 2 + \ldots + c_1 \cdot (n-2) + c_1 \cdot (n-1) + c_1 \cdot n =$$
$$= c_2 + c_1 \cdot (2 + \ldots + (n-2) + (n-1) + n) = c_2 + c_1 \cdot \frac{(2+n) \cdot (n-1)}{2} =$$
$$= c_2 + c_1 \frac{n^2 + n - 2}{2} \quad - \quad \textit{квадратичная сложность}$$

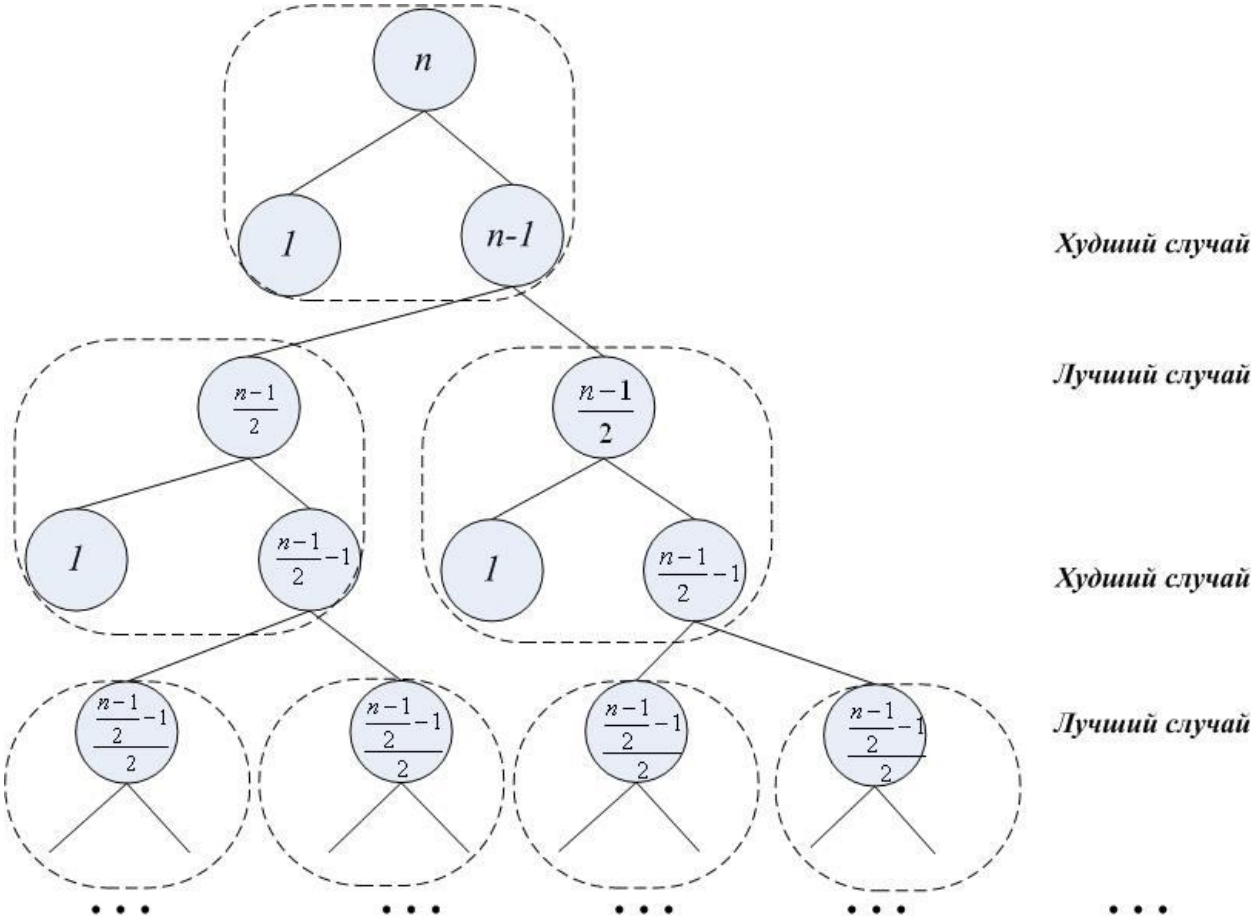
Таким образом, $T_{\text{худший случай}}(n) = \mathrm{O}(n^2)$.

Возникает вопрос: правомерно ли называть сортировку Хоара быстрой сортировкой? Для ответа на этот вопрос рассмотрим работу алгоритма в

среднем случае. Если оценка среднего случая окажется ближе к оценке лучшего случая, то сортировку можно будет считать быстрой.

Средний случай

Рассмотрим гипотетический средний случай, когда происходит чередование худшего и лучшего случаев. Дерево делений массива будет иметь следующий вид:



Данное дерево является аналогом двоичного дерева с составными вершинами, каждая из которых имеет сложность $\Theta(n)$, где n – длина рассматриваемой части массива. Как видно из рисунка, каждый раз массив делится на две равные части, т. е. пополам, таким образом, рекуррентное соотношение времени работы алгоритма в среднем случае имеет вид:

$$T_{средний\,случай}(n) = \begin{cases} 2 \cdot T\left(\dfrac{n}{2}\right) + \Theta(n) + \Theta(n), & n > 1 \\ \Theta(1), & n = 1 \end{cases}.$$

Найдем методом итераций значение верхней оценки, предварительно раскрыв асимптотические обозначения и взяв лишь их правые части.

$$T_{средний\ \ случай}(n) \le 2 \cdot T\left(\frac{n}{2}\right) + c_1 \cdot n + c_3 \cdot n = 2 \cdot T\left(\frac{n}{2}\right) + (c_1 + c_3) \cdot n \le$$

$$\le 2 \cdot \left(2 \cdot T\left(\frac{n}{4}\right) + c_1 \cdot \frac{n}{2} + c_3 \cdot \frac{n}{2}\right) + (c_1 + c_3) \cdot n = 4 \cdot T\left(\frac{n}{4}\right) + 2 \cdot (c_1 + c_3) \cdot n \le \ldots$$

$$\ldots \le 2^i \cdot T\left(\frac{n}{2^i}\right) + i \cdot (c_1 + c_3) \cdot n \le$$

$$\left\langle предположи\ м, что\ \frac{n}{2^i} = 1, т.е.\ i = \log_2 n \right\rangle$$

$$\le 2^{\log_2 n} \cdot c_2 + (c_1 + c_3) \cdot n \cdot \log_2 n =$$

$$= (c_1 + c_3) \cdot n \cdot \log_2 n + c_2 \cdot n.$$

Таким образом, $T_{средний\ \ случай}(n) = O(n \cdot \log_2 n)$, т. е. средний и лучший случаи асимптотически эквивалентны, и сортировку Хоара можно считать быстрой.

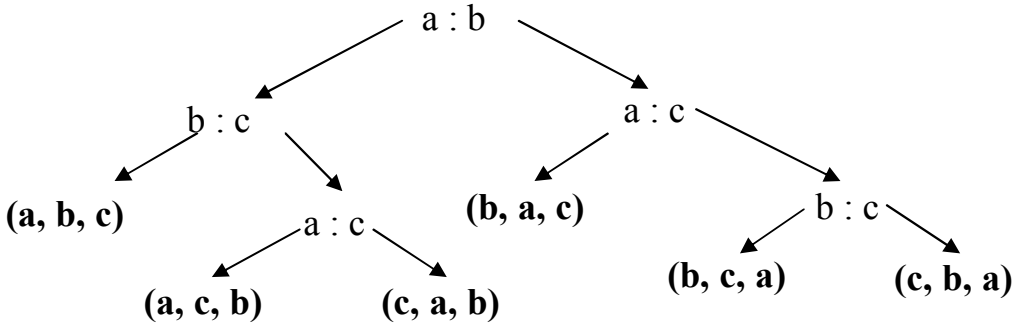
5.4. Нижние оценки для сортировки сравнением

Приведем верхние и нижние оценки разных методов сортировки информации, в основе которых лежит сравнение элементов друг с другом.

Сортировка	Асимптотические оценки		
	Нижняя оценка (Ω)	Верхняя оценка (O)	Порядок роста (Θ)
Метод простого включения	n	n^2	–
Метод простого обмена	n^2	n^2	n^2
Метод простого выбора	n^2	n^2	n^2
Слиянием	$n \cdot \log_2 n$	$n \cdot \log_2 n$	$n \cdot \log_2 n$
Хоара	$n \cdot \log_2 n$	n^2	–

Возникает вопрос: можно ли построить алгоритм, имеющий оценки лучше оценок, приведенных в таблице?

Для ответа на этот вопрос построим разрешающее дерево, сортирующее по возрастанию, например, три элемента (a, b, c).



Разрешающее дерево является двоичным деревом. Количество листьев равно $n!$ – количеству всевозможных перестановок из n элементов.

Число сравнений в худшем случае равно высоте разрешающего дерева, т. е. максимальной длине простого пути в этом дереве от корня до листа.

Теорема. Высота любого разрешающего дерева, сортирующего n элементов, есть $\Omega(n \cdot \log_2 n)$.

Смысл этой теоремы заключается в следующем: нельзя построить алгоритм сортировки сравнением, имеющий верхнюю оценку лучше, чем $O(n \cdot \log_2 n)$. В этом смысле сортировка слиянием является асимптотически оптимальной.

6. Вопросы и задания для самоконтроля

1. Что является объективной характеристикой временной сложности алгоритма?

2. Что представляет собой функция времени работы алгоритма?

3. Что дает оценка времени работы алгоритма в худшем, лучшем и среднем случаях? Какова их связь с асимптотическими обозначениями?

4. Из чего складывается время работы рекурсивного алгоритма?

5. В чем заключаются сложности каждого из способов решения рекуррентных соотношений?

6. В каких случаях не применима теорема о рекуррентных соотношениях?

7. Какие методы сортировки информации являются асимптотически оптимальными? Почему?

8. Выяснить, обладают ли функции $\Theta(g(n))$, $O(g(n))$ и $\Omega(g(n))$ следующими свойствами:

– *транзитивность*:
если $f(n) = \Theta(g(n))$ и $g(n) = \Theta(h(n))$, то $f(n) = \Theta(h(n))$;
если $f(n) = O(g(n))$ и $g(n) = O(h(n))$, то $f(n) = O(h(n))$;
если $f(n) = \Omega(g(n))$ и $g(n) = \Omega(h(n))$, то $f(n) = \Omega(h(n))$;

– *рефлексивность:*
$f(n) = \Theta(f(n))$; $f(n) = O(f(n))$; $f(n) = \Omega(f(n))$;

– *симметричность:*
$f(n) = \Theta(g(n))$ тогда и только тогда, когда $g(n) = \Theta(f(n))$;
$f(n) = O(g(n))$ тогда и только тогда, когда $g(n) = O(f(n))$;

– *обращение:*
$f(n) = O(g(n))$ тогда и только тогда, когда $g(n) = \Omega(f(n))$.

9. Даны следующие функции от n:

$f_1(n) = n^2$;
$f_2(n) = n^2+1000\cdot n$;

$$f_3(n)=\begin{cases} n, & \text{если } n \text{ нечетно} \\ n^3, & \text{если } n \text{ четно} \end{cases};$$

$$f_4(n)=\begin{cases} n, & \text{если } n > 100 \\ n^3, & \text{если } n \le 100 \end{cases}.$$

Указать для каждой пары функций, когда $f_i(n)$ имеет порядок роста $O(f_j(n))$ и когда $f_i(n)$ есть $\Omega(f_j(n))$.

10. Можно ли утверждать, что $2^{n+1} = O(2^{n+1})$; $2^{2\cdot n} = O(2^{\cdot n})$?

11. Доказать по определению, что следующие утверждения истинны:

- 17 имеет порядок $O(1)$;
- $n\cdot(n-1)/2$ имеет порядок $O(n^2)$;
- $\max(n^3, 10\cdot n^2)$ имеет порядок $O(n^3)$.

12. Пусть $T_1(n)$ есть $\Omega(f(n))$ и $T_2(n)$ есть $\Omega(g(n))$. Какие из следующих утверждений истинны? Доказать:

- $T_1(n) + T_2(n)$ есть $\Omega(\max(f(n), g(n)))$;
- $T_1(n) \cdot T_2(n)$ есть $\Omega(f(n) \cdot g(n))$.

13. Выполнить анализ временной сложности следующих алгоритмов:
– линейного поиска с барьером;
– сортировка методом простого обмена;
– сортировка методом простого выбора.

14. Выполнить анализ временной сложности рекурсивной реализации бинарного поиска.

15. Найти порядок $T(n)$, если $T(1) = 1$:

- $T(n) = T(n/2)+n$;
- $T(n) = T(n-1)+n^2$;
- $T(n) = n\cdot T(n-1)$.

16. Оценить временную сложность рекурсивной процедуры.

```
Procedure Soch (i : Integer);
Var k : Integer;
Begin
 If i>n Then Print(a)
 Else For k:=1 To n Do
  Begin
   a[i]:=k;
   Soch(i+1);
  End;
End;
```

Литература

1. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы. – М.: Вильямс, 2000.

2. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. – М.: Мир, 1982.

3. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы. Построение и анализ. – М.: МЦНМО, 2001.

4. Рейнгольд Э., Нивергельт Ю., Део Н. Комбинаторные алгоритмы. Теория и практика. – М.: Мир, 1980.