Министерство образования и науки РФ
Государственное образовательное учреждение
Высшего и профессионального образования
Иркутский Государственный университет
Институт математики, экономики и информатики

Анализ алгоритмов с примерами на языке Pascal

учебное пособие

2015

Учебное пособие предназначено для студентов обучающихся по направлению подготовки бакалавров «Математика» и «Прикладная математика и информатика». Может быть использовано в курсах «Основы информатики», «Компьютерные науки».

Составитель: Мезенцев А.В.

© Иркутский Государственный университет, 2015 г.

Оглавление.

1. Анализ не рекурсивных алгоритмов 4
2. Оценка алгоритмов количество итераций которых не задано явно 12
3. Анализ рекурсивных алгоритмов 17
4. P, NP и NP-полные задачи 27
5. Алгоритмы генерации комбинаторных объектов 38

1. Анализ не рекурсивных алгоритмов .

Анализ алгоритма можно определить как поиск оценок затрачиваемых на выполнение алгоритма памяти и времени. Наиболее часто анализ алгоритмов используется либо для того, чтобы сравнить два различных алгоритма решения одной и той же задачи, либо для того , чтобы выявить практическую применимость алгоритма.

Оценка алгоритма по памяти достаточно прозрачна, и здесь мы ее рассматривать не будем.

Сосредоточимся на оценке алгоритмов по времени выполнения. Один из подходов к оценке алгоритмов по времени выполнения заключается в том, чтобы просто запускать алгоритм на компьютере и засекать тем или иным образом время его выполнения.

У этого подхода есть много недостатков. Во-первых, время выполнения сильно зависит от компьютера, на котором выполняется алгоритм. Во-вторых, такая оценка дает только одно значение для конкретной размерности входных данных. Даже если у нас есть целая таблица оценок для различных размерностей, получить по ней функциональную зависимость времени выполнения от размерности входных данных весьма проблематично (то есть мы не сможем получить оценку для произвольной размерности). В-третьих, такая оценка зависит от реализации (например, от того какой программист будет кодировать этот алгоритм).

Поэтому для оценки алгоритма по времени выполнения стараются найти функциональную зависимость количества выполняемых элементарных операций от размерности входных данных.

Рассмотрим оценку алгоритма на примере.

Пусть у нас есть следующий алгоритм, вычисляющий сумму значений всех компонентов одномерного массива:

```
(1) s:=0;
(2) for i:=1 to n do
(3)       s:= s+a[i];
```

Алгоритм надо записать таким образом, чтобы в одной строке был один оператор. Далее, рядом с каждым выполняемым оператором надо записать выражение, зависящее от размерности входных данных указывающее то количество раз, которое этот оператор будет выполняться. Эта оценка может быть более или менее точной, главное чтобы вы выполняли ее однообразно. Например, можно считать, что каждый оператор выполняется за одну абстрактную единицу времени. Или разбивать выполнение каждого оператора на последовательность выполнений элементарных операций: прочитать из памяти, записать в память, выполнить арифметическую операцию.

При первом подходе мы получим следующие оценки. Первый оператор будет выполнен один раз и это не зависит от размерности входных данных. Количество выполнений второго оператора зависит от размерности входных данных (конкретно от длины массива n). В нашем случае это n+1 (не забываем о том, что заголовок цикла `for` выполняется на единицу больше раз, чем его тело). Соответственно третий оператор будет выполняться за n абстрактных единиц времени. Таким образом, имеем:

(1) `s:=0;`	1
(2) `for i:=1 to n do`	n+1
(3) `s:= s+a[i];`	n

Осталось просуммировать оценки всех операторов и получить оценку алгоритма $2n$+2.

При втором подходе имеем:

(1) `s:=0;`	1

```
(2) for i:=1 to n do          6(n+1)
(3)     s:= s+a[i];           7n
```

Здесь оценка первого оператора осталась такой же, так как надо выполнить только одну элементарную операцию: записать в память `s` значение константы 0. Второй оператор разбивается на следующие элементарные операции: прочитать из памяти `i`; прочитать из памяти значение шага; выполнить сложение `i` и шага; записать в память `i`; прочитать из памяти `n`; сравнить значения `i` и `n`. Итого 6 операций. Третий оператор разбивается на семь операций: прочитать из памяти `i`; прочитать из памяти адрес начала массива `a`; сложить эти два адреса; прочитать по этому адресу `a[i]`; прочитать из памяти `s`; сложить; записать в память `s`. Итого 7 операций. Оценка алгоритма – $13n+7$.

Кажется, что эти два подхода дают нам две различные оценки одного и того же алгоритма. На самом деле в анализе алгоритмов в большинстве случаев используется еще более грубый подход. В качестве оценки берется выражение $O(f(n))$. Где в качестве $f(n)$ берется одна из стандартных функций используемых в анализе алгоритмов, n – размерность входных данных. Напомним, что $g(n) = O(f(n))$, когда $\lim_{n\to\infty}\frac{f(n)}{g(n)} = C$, где C произвольная константа. Поэтому и первый, и второй подход дают нам одну и ту же оценку $O(n)$. Более точные оценки используются только когда сравнивают различные алгоритмы решения одной и той же задачи.

Наиболее часто в оценке алгоритмов используются следующие функции: $\log_2 n$, n, $n\cdot\log_2 n$, n^2, n^3, 2^n, 10^n, $n!$. Алгоритмы, имеющие оценку $O(\log(n))$,неважно по какому основанию, называются *очень быстрыми алгоритмами*. Таких алгоритмов существует не очень много. На самом деле в литературе упоминается обычно только один алгоритм, имеющий оценку $O(\log_2 n)$ – это алгоритм бинарного поиска. Мы рассмотрим его позже. Алгоритмы, имеющие оценку $O(n)$ и $O(n\cdot\log n)$ называются *быстрыми алгоритмами*.

Алгоритмы имеющие оценку O(n^2), O(n^3), или в общем случае O(n^C), где C произвольная положительная константа, называются *полиномиальными алгоритмами*. Наконец, алгоритмы, имеющие оценку O(2^n), O(10^n), O(n!) называются *неполиномиальными* алгоритмами.

Если алгоритм, который вы разработали, является быстрым или очень быстрым, то вам крупно повезло. Это очень эффективный алгоритм и его можно применять к данным практически неограниченной размерности.

Если алгоритм является полиномиальным, тоже не плохо. Такие алгоритмы широко используются и могут применяться к данным с практически важными размерностями.

Если же алгоритм является неполиномиальным, то это значит, что решение практически важных задач с помощью такого алгоритма найти будет невозможно.

В следующей таблице приводятся значения этих функций для некоторых аргументов:

n	1	2	5	10	20	30	50
$log_{10} n$	0	0,30103	0,69897	1	1,30103	1,477121	1,69897
$ln\ n$	0	0,693147	1,609438	2,302585	2,995732	3,401197	3,912023
$log_2 n$	0	1	2,321928	3,321928	4,321928	4,906891	5,643856
n	1	2	5	10	20	30	50
$n\ ln\ n$	0	1,386294	8,04719	23,02585	59,91465	102,0359	195,6012
n^2	1	4	25	100	400	900	2500
n^3	1	8	125	1000	8000	27000	125000
2^n	2	4	32	1024	1048576	$1{,}07 \cdot 10^9$	$1{,}13 \cdot 10^{15}$
10^n	10	100	100000	10^{10}	10^{20}	10^{30}	10^{50}
$n!$	1	2	120	3628800	$2{,}43 \cdot 10^{18}$	$2{,}65 \cdot 10^{32}$	$3{,}04 \cdot 10^{64}$
n^n	1	4	3125	10^{10}	10^{26}	$2 \cdot 10^{44}$	$8{,}88 \cdot 10^{84}$

Для того чтобы было понятно, что скрывается за этими абстрактными числами типа 10^{10} или 10^{30} рассмотрим еще одну таблицу.

Будем считать, что у нас имеются два компьютера, первый из которых может выполнить 10^6 операций в секунду, а второй 10^9 операций в секунду. Заметим, что под операциями здесь мы понимаем операции, время выполнение которых в оценке алгоритмов считается за абстрактную единицу. То есть быстродействие наших абстрактных компьютеров несравнимо с тактовой частотой современных компьютеров. Вряд ли самый современный компьютер может быть быстрее, чем наш второй абстрактный.

	Первый компьютер		Второй компьютер	
10^{10}	10^4 сек.	2,7 часа	10 сек.	10 сек.
10^{12}	10^6 сек.	10 дней	10^3 сек.	17 мин.
10^{15}	10^9 сек.	31 год	10^6 сек.	10 дней
10^{18}	10^{12} сек.	31710 лет	10^9 сек.	31 год
10^{20}	10^{14} сек.	3 млн. лет	10^{11} сек.	3171 год
10^{30}	10^{24} сек.	31 000 000 млрд. лет	10^{21} сек.	31 000 млрд. лет

В первом столбце таблицы приводятся различные значения количества операций, которые необходимо выполнить для завершения алгоритма. Мы видим что, начиная с 10^{18} для первого компьютера и 10^{20} для второго, время выполнения алгоритмов становятся совершенно нереальными. А ведь такие числа получаются для алгоритмов с оценкой $O(n!)$ всего-навсего при n=20. Размерности практически важных задач обычно гораздо больше.

Рассмотрим теперь примеры алгоритмов, оценка которых зависит не только от размерности входных данных. Это широко распространенный алгоритм нахождения наибольшего (наименьшего) значения среди компонентов одномерного массива.

```
(1) max:=a[i];                 1
(2) for i:=2 to n do           n
(3)   if max<a[i] then         n-1
```

(4) `max:=a[i];` от 0 до n-1

Нахождение оценок 1, 2 и 3 операторов не должно вызвать затруднений. А вот количество выполнений 4 оператора зависит от конкретных значений компонентов массива, поэтому мы не можем дать однозначной оценки. В этом случае поступают следующим образом. Дают не одну оценку, а три: *наилучшую*, *наихудшую* и *среднюю*. Из этих трех оценок сложнее всего найти среднюю (даже сформулировать что значит средняя), хотя она с практической точки зрения и является наиболее важной. Для студентов первого курса это, пожалуй, является трудно разрешимой задачей, поэтому здесь мы ее рассматривать не будем.

Что касается наилучшей и наихудшей оценок, то их искать проще. Надо представить себе такие входные данные, для которых соответствующий оператор будет выполняться наименьшее и наибольшее количество раз соответственно.

Для нашего примера наилучшими входными данными будет такой массив, в котором максимум стоит на первом месте. В этом случае 4 оператор ни разу не выполнится, так как условие в 3 операторе будет все время `false`. Наихудшими входными данными будет такой массив, в котором компоненты упорядочены по возрастанию. В этом случае условие в 3 операторе каждый раз будет `true`, и оператор 4 будет выполняться каждый раз.

Таким образом, лучшая оценка нашего алгоритма равна $2n$, а худшая – $3n$-1.

В качестве примера оценки более сложных алгоритмов содержащих вложенные циклы рассмотрим два самых простых, и обычно приходящих в голову первыми, алгоритма сортировки.

Условие задачи: дан массив целых чисел $a_1, a_2, \ldots, a_n$. Упорядочить компоненты массива по неубыванию.

Первый алгоритм называется *алгоритмом «сортировки методом выбора»*.

Суть его в следующем. Во всем массиве ищется минимум и ставится на первое место, в оставшейся части массива ищется минимум и ставится на второе место и так далее, пока не рассмотренным не останется последний элемент массива. Этот алгоритм используют люди в повседневной жизни. Априори этот алгоритм кажется достаточно «плохим», то есть не эффективным. Рассмотрим этот алгоритм подробнее:

(1)	`for i:=1 to n-1 do begin`	n
(2)	`  min:=a[i];`	n-1
(3)	`  k:=i;`	n-1
(4)	`  for j:=i+1 to n do`	$(n^2+n-2)/2$
(5)	`    if min>a[j] then begin`	$(n^2-n)/2$
(6)	`      min:=a[j];`	от 0 до $(n^2-n)/2$
(7)	`      k:=j`	от 0 до $(n^2-n)/2$
(8)	`    end;`	
(9)	`  a[k]:=a[i];`	$n-1$
(10)	`  a[i]:=min`	$n-1$
(11)	`end`	

Оценка 4 оператора получена следующим образом. Этот заголовок цикла выполняется для каждого `i`, причем для `i` =1 он выполнится n раз, для `i` =2 – $n-1$ раз, и так далее, при `i`=$n-2$ – 3 раза, при `i` = $n-1$ – 2 раза. То есть мы имеем арифметическую прогрессию $a_1, a_2, \ldots, a_m$, сумма которой

$$s_m = \frac{(a_1 + a_m) \cdot m}{2}.$$

Здесь $m=n$-1, $a_1=n$, $a_{n-1}=2$ и сумма равна $(n+2)(n-1)/2 = (n^2+n-2)/2$.

Оценка 5 оператора получена подобным же образом, только не надо забывать о том, что заголовок цикла `for` всегда выполняется на один раз больше чем его тело. Здесь m=n-1, a_1=n-1, a_{n-1}=1 и сумма равна (n-1+1)(n-1)/2 = (n^2-n)/2.

Что касается оценки 5 и 6 операторов, то с этим мы уже сталкивались ранее. В зависимости от входных данных эти операторы могут выполняться различное количество раз. Для «хороших» данных (массив уже упорядочен) эти операторы не выполнятся ни разу. Для «плохих» данных эти операторы будут выполняться при каждой проверке `min>a[j]`. Для алгоритмов сортировки «плохие» данные представляют собой массив, упорядоченный в обратном порядке.

Таким образом, лучшая оценка алгоритма «сортировки методом выбора» равна n^2+5n-5, худшая оценка равна $2n^2+4n-5$. Плохо это, или хорошо? Пока можно заметить только, что алгоритм тратит достаточно много времени, когда массив уже упорядочен. Это несомненный минус этого алгоритма.

Рассмотрим теперь другой алгоритм, который называется *алгоритмом «сортировки методом пузырька»*. Суть его заключается в том, что сравнивается пара расположенных рядом элементов массива и, если они стоят не месте, они меняются местами. Проблема заключается в том, что за один просмотр всех пар массив можно и не упорядочить. Поэтому требуется еще один как бы «холостой» прогон. Кажется, что этот алгоритм намного лучше предыдущего, но давайте не будем торопиться. Оценим алгоритм «сортировки методом пузырька»:

		лучший	худший
(1)	`do`		
(2)	`  f:=false;`	1	n
(3)	`  for i:=1 to n-1 do`	n	n^2
(4)	`    if a[i]>a[i+1] then begin`	n-1	$n(n-1)$
(5)	`      t:=a[i];`	0	$n(n-1)$

(6)	`a[i]:=a[i+1];`	0	$n(n-1)$
(7)	`a[i+1]:=t;`	0	$n(n-1)$
(8)	`f:=true`	0	$n(n-1)$
(9)	`end`		
(10)	`until not(f);`	1	n

Лучшая оценка алгоритма «сортировки методом пузырька» равна $2n+1$. Худшая оценка – $6n^2-4n$. Теперь мы можем сравнить эти два алгоритма решающие одну и ту же задачу. Метод пузырька для «хороших» данных работает лучше, но для «плохих» данных он работает почти в три раза дольше, чем метод выбора.

Таким образом, выбор того или иного алгоритма сортировки зависит от того, какие входные данные вы ожидаете для своего алгоритма. Если предполагается, что массивы будут отсортированы или почти отсортированы, то следует отдать предпочтение методу пузырька. Если предполагается, что массивы будут отсортированы или почти отсортированы в обратном порядке, то следует отдать предпочтение методу выбора. Следует заметить, что для любых самых быстрых алгоритмов сортировки оценка в худшем случае $O(n^2)$.

2. Оценка алгоритмов, количество итераций которых не задано явно

Рассмотрим следующую задачу:

Вычислить сумму бесконечного сходящегося ряда $1+\frac{x}{1!}+\frac{x^2}{2!}+...+\frac{x^n}{n!}+...$.

Легко видеть, что это разложение в ряд широко известной экспоненциальной функции (e^x). Ряд сходящийся, то есть каждое очередное слагаемое должно быть меньше по абсолютной величине предыдущего. Это значит, что всегда найдется такое слагаемое, начиная с которого все остальные будут добавлять

к сумме пренебрежимо малую величину. Таким образом, мы можем написать алгоритм, вычисляющий искомую сумму с любой заданной точностью.

Например, если заданная точность 0,01, то все слагаемые, значения которых меньше 0,01, мы отбросим, и наша вычисленная сумма будет отличаться от истинной не больше чем на 0,01.

Для решения этой задачи можно составить два алгоритма. Первый представляет собой «лобовое» решение. Степень и факториал вычисляются явно:

```
y:=0;
p:=1;
n:=1;
while Abs(p)<=eps do begin
    y:=y+p;
    f:=1;
    for i:=1 to n do
        f:=f*I;
    p:=Power(x, n)/f;
    n:=n+1;
end;
```

Второй алгоритм более «хитрый». Он учитывает тот факт, что для того чтобы получить второе слагаемое, достаточно первое слагаемое умножить на x и разделить на 1. Для того чтобы получить третье слагаемое достаточно второе слагаемое умножить на x и разделить на 2, и так далее.

```
y:=0;
p:=1;
n:=1;
while Abs(p)<=eps do begin
    y:=y+p;
```

```
    p:=p*x/n;
    n:=n+1;
end;
```

Даже на первый взгляд второй алгоритм кажется более эффективным, чем первый. Давайте оценим это точно.

При оценке подобных алгоритмов мы сталкиваемся с одной проблемой. Дело в том, что мы не можем знать заранее, сколько раз будет выполняться тело цикла в этом алгоритме. Мы не можем вычислить это по заданной точности `eps`, все зависит от конкретного ряда. Проблема кажется трудно разрешимой, однако заметим, что в обоих алгоритмах для одинаковых входных данных циклы выполняются одинаковое количество раз. А так как нам надо лишь выяснить во сколько раз один алгоритм эффективнее другого, то можно просто задать какое-либо абстрактное количество итераций цикла. Тогда алгоритмы будут выглядеть так.

Первый алгоритм:

```
y:=0;
p:=2;
for n:=1 to m do begin
    y:=y+p;
    f:=1;
    for i:=1 to n do
        f:=f*I;
    p:=Power(x, n)/f;
end;
```

Второй алгоритм:

```
y:=0;
p:=2;
for n:=1 to m do begin
```

```
    y:=y+p;
    p:=p*x/n;
end;
```

Оценим первый алгоритм:

`y:=0;`	1
`p:=1;`	1
`for n:=1 to m do begin`	$m+1$
`  y:=y+p;`	m
`  f:=1;`	m
`  for i:=1 to n do`	$(m^2+3m)/2$
`    f:=f*I;`	$(m^2+m)/2$
`  p:=Power(x, n)/f;`	m
`end;`	

Суммарная оценка: m^2+6m+3.

Оценка достаточно грубая, в частности мы не учитываем тот факт, что умножение требует больше времени, чем сложение, а возведение в степень – больше чем умножение.

Оценим второй алгоритм:

`y:=0;`	1
`p:=1;`	1
`for n:=1 to m do begin`	$m+1$
`  y:=y+p;`	m
`  p:=p*x/n;`	m
`end;`	

Суммарная оценка: $3m+3$.

Даже с учетом того, что мы проигнорировали разницу в выполнении операций возведения в степень (первый алгоритм) и умножения (второй алгоритм), второй алгоритм является гораздо более эффективным. Например, при m=10 оценка первого алгоритма 163, а второго 33. То есть, второй алгоритм выполнится почти в пять раз быстрее. При m=20 эти оценки равны 523 и 63 соответственно. Второй алгоритм быстрее почти в девять раз.

Умножение двух матриц

Рассмотрим задачу, алгоритм решения которой имеет оценку $O(n^3)$. Это задача умножения двух матриц. Требуется вычислить произведение двух матриц: a размерности $n \times k$ и b размерности $k \times m$. Результатом будет матрица c размерности $n \times m$.

```
for i:=1 to n do                              n+1
    for j:=1 to m do begin                    n·(m+1)
        c[i,j]:=0;                            n·m
        for ii:=1 to k do                     n·m·(k+1)
            c[i,j]:=a[i,ii]*b[ii,j];          n·m·k
end;
```

Суммарной оценкой будет - $2 \cdot n \cdot m \cdot k + 3 \cdot n \cdot m + 2 \cdot n + 1$, т.е. $O(n \cdot m \cdot k)$.

Если же обе исходные матрицы будут квадратными, т.е. $n=m=k$, то суммарная оценка алгоритма будет - $2 \cdot n^3 + 3 \cdot n^2 + 2 \cdot n + 1$, или более грубо - $O(n^3)$.

3. Анализ рекурсивных алгоритмов

Рассмотрим рекурсивный алгоритм, вычисляющий сумму компонентов одномерного массива.

```
function sum(a: array of integer, n: integer): integer;
    begin
        if n<1 then
            sum:=0
        else
            sum:=sum(a, n-1)+a[n]
    end
```

Основная операция – суммирование. При изменении аргумента на 1 количество выполненных основных операций равно 1. При значении аргумента равным 0 (`n<1`), количество выполненных основных операций равно 0. Таким образом, мы получаем следующее рекуррентное соотношение

$S(n) = S(n-1) + 1$, с начальным условием

$S(0) = 0$.

Решаем это рекуррентное соотношение методом обратной подстановки.

Прямая подстановка:

$S(n-1) = S(n-2) + 1$

$S(n-2) = S(n-3) + 1$

$S(n-3) = S(n-4) + 1$

… и т. д.

Теперь подставляем найденные значения обратно в исходное соотношение:

$S(n) = (S(n-2) + 1) + 1 =$

$((S(n-3) + 1) + 1) + 1 =$

$$(((S(n-4) + 1) + 1) + 1) + 1) = S(n-4) + 4$$

Или в общем виде, для произвольного k - $S(n) = S(n-k) + k$.

Для того, чтобы воспользоваться начальными условиями, аргумент у S в правой части должен быть равным 0. То есть, $n-k = 0$. Отсюда $k = n$.

$$S(n) = S(n-k) + k = S(n-n) + n = S(0) + n = 0 + n = n$$

Оценка эффективности нашего алгоритма по времени – $O(n)$.

Рекурсивный алгоритм нахождения наибольшего значения среди компонентов одномерного массива.

```
int max(int [] a, int n){
    if(n<=1)
        return a[n];
    else
        if(max(a, n-1)>a[n])
            return max(a, n-1);
        else
            return a[n];
}
```

Основная операция – сравнение (`max(a, n-1)>a[n]`). Заметим, что время работы нашего алгоритма зависит не только от размерности, поэтому будем строить худшую оценку C_w.

В худшем случае, при изменении значения размерности на 1, задача распадается на две, меньшей размерности, плюс одно сравнение. Получаем следующее рекуррентное соотношение:

$C_w(n) = 2 \cdot C_w(n-1) + 1$, с начальным условием:

$C_w(1) = 0$.

Решаем рекуррентное соотношение методом обратной подстановки.

Прямая подстановка:

$$C_w(n-1) = 2 \cdot C_w(n-2) + 1$$

$$C_w(n-2) = 2 \cdot C_w(n-3) + 1$$

$$C_w(n-3) = 2 \cdot C_w(n-4) + 1$$

… и т. д.

Теперь подставляем найденные значения обратно в исходное соотношение:

$$C_w(n) = 2 \cdot (2 \cdot C_w(n-2) + 1) + 1 =$$

$$= 2 \cdot (2 \cdot (2 \cdot C_w(n-3) + 1) + 1) + 1 =$$

$$= 2 \cdot (2 \cdot (2 \cdot (2 \cdot C_w(n-4) + 1) + 1) + 1) + 1 =$$

$$= 2^4 \cdot C_w(n-4) + 2^3 + 2^2 + 2^1 + 2^0$$

Или в общем виде, для произвольного k:

$$C_w(n) = 2^k \cdot C_w(n-k) + 2^{k-1} + 2^{k-2} + \ldots + 2^1 + 2^0$$

Для того, чтобы воспользоваться начальными условиями, аргумент у C_w в правой части должен быть равным 1. То есть, $n-k = 1$. Отсюда $k = n-1$.

$$C_w(n) = 2^k \cdot C_w(1) + 2^{n-2} + 2^{n-3} + \ldots + 2^1 + 2^0$$

$C_w(1) = 0$, таким образом

$$C_w(n) = 2^{n-2} + 2^{n-3} + \ldots + 2^1 + 2^0$$

В правой части – сумма степеней 2:

$$2^{n-2} + 2^{n-3} + \ldots + 2^1 + 2^0 = 2^{n-1} - 1$$

$$C_w(n) = 2^{n-1} - 1$$

Оценка эффективности нашего алгоритма по времени - $O(2^n)$. Это очень «плохая» оценка. Но почему оценка именно такая? Дело в том, что в худшем случае мы два раза рекурсивно вызываем экземпляр нашего алгоритма с

одним и тем же аргументом! Если его вызывать один раз, и сохранять результат, можно значительно улучшить наш алгоритм.

```
int max(int [] a, int n){
    int m;
    if(n<=1)
        return a[n];
    else{
        m= max(a, n-1);
        if(m>a[n])
            return m;
        else
            return a[n];
    }
}
```

Основная операция по прежнему – сравнение (`m>a[n]`).

В худшем случае, при изменении значения размерности на 1, требуется ***один раз*** решить задачу меньшей размерности и выполнить одно сравнение. Получаем следующее рекуррентное соотношение:

$C_w(n) = C_w(n-1) + 1$, с начальным условием:

$C_w(1) = 0$.

Такое рекуррентное соотношение мы уже решали выше и знаем ответ:

$C_w(n) = n$

Оценка эффективности улучшенного алгоритма по времени - $O(n)$.

Задача поиска.

Рассмотрим алгоритмы решения задачи поиска. Дан массив ключей $a_1, a_2, \ldots, a_n$ и искомое значения ключа b, требуется выяснить, существует ли

в массиве ключей ключ с таким значением, и, если он есть, вычислить его индекс в массиве ключей.

Первый алгоритм называется ***алгоритмом последовательным поиском***. Поочередно сравниваем b с элементами массива a_1, a_2, …, a_n. Обнаружив совпадение, мы возвращаем индекс найденного элемента; если же элемент не найден, то возвращаем –1.

```
ind:=-1;
for i:=1 to n do
    if a[i]=b then begin
        ind:=I;
        break
    end;
```

Достаточно понятно, что придется проверить в худшем случае n элементов, в среднем $n/2$. Таким образом, сложность (эффективность) алгоритма последовательного поиска в худшем случае – $O(n)$.

Второй алгоритм называется ***алгоритмом бинарного (двоичного, дихотомического) поиска***.

Предполагается, что массив ключей отсортирован по возрастанию. При этом условии можно значительно повысить эффективность алгоритма. На каждой итерации алгоритма мы будем вычислять средний индекс в той части массива ключей, в которой предполагается искомое значение. Сравниваем этот средний элемент массива с искомым значением. Может быть три варианта. Первый – они равны; искомый индекс найден. Второй – элемент массива меньше искомого значения; необходимо продолжить поиск в правой половине массива. Третий – элемент массива больше искомого значения; необходимо продолжить поиск в левой половине массива. Так как на каждой итерации длина той части массива, в которой осуществляется поиск, уменьшается в два раза, поиск достаточно быстро завершится.

```
l:=1; r:=n; und:=-1;
while l<=r do begin
    m:=(l+r) div 2;
    if a[m]=b then begin
        ind:=m;
        break
    end;
    if b<a[m] then r:=m-1 else l:=m+1
end;
```

Для оценки эффективности этого алгоритма не годится метод, который мы до этого использовали так как, мы не сможем для каждого из операторов алгоритма указать количество раз выполнения.

Мы будем использовать метод, который разработан в основном для оценки рекурсивных алгоритмов. Хотя наш алгоритм не рекурсивный, в принципе, он имеет рекурсивный характер.

План анализа эффективности рекурсивных алгоритмов

1. Выберите параметр (параметры), по которым будет оцениваться размер входных данных алгоритма.
2. Определите основную операцию алгоритма.
3. Проверьте, зависит ли число выполняемых основных операций только от размера входных данных, Если оно зависит и от других факторов, рассмотрите при необходимости, как меняется эффективность алгоритма для наихудшего, среднего и наилучшего случаев.
4. Составьте рекуррентное соотношение, выражающее количество выполняемых основных операций алгоритма, и укажите соответствующие начальные условия.
5. Найдите решение рекуррентного уравнения или, если это невозможно, определите хотя бы его порядок роста.

Оценим количество сравнений искомого ключа с элементами массива. Из соображений простоты, мы будем считать, что одно сравнение `b` и `a[m]` позволяет определить `b < a[m]`, `b = a[m]`, или `b > a[m]`.

Обозначим через $C_w(n)$ – количество сравнений в наихудшем случае. Вообще говоря, существуют два варианта окончания работы алгоритма в худшем случае: первый – массив не содержит искомое значение; второй - `b=a[m]` и `l=r=m`.

Поскольку после одного сравнения алгоритм попадает в ту же ситуацию, что и до сравнения, только область поиска становится вдвое меньше, можно записать следующее рекуррентное соотношение (уравнение):

$$C_w(n) = C_w(\lfloor n/2 \rfloor)+1, \text{ если } n>1 \qquad C_w(1) = 1$$

(здесь $\lfloor x \rfloor$ - округление в меньшую сторону).

Рекуррентные соотношения для оценки сложности алгоритма выводятся непосредственно из вида алгоритма, однако с их помощью нельзя быстро вычислить эту оценку.

Для этого следует привести рекуррентное соотношение к так называемому ***замкнутому виду***, отказавшись от их рекурсивной природы.

Производится такое приведение посредством последовательных подстановок, позволяющих уловить общий принцип.

Рекуррентные соотношения.

Рекуррентные соотношения для оценки сложности алгоритма выводятся непосредственно из вида алгоритма, однако с их помощью нельзя быстро вычислить эту оценку.

Для этого следует привести рекуррентное соотношение к так называемому ***замкнутому виду***, отказавшись от их рекурсивной природы.

Производится такое приведение посредством последовательных подстановок, позволяющих уловить общий принцип.

Рекуррентное соотношение задается в одной из двух форм:

1) Когда простых случаев немного:

$T(n) = 2 \times T(n-2) - 15;$

$T(2) = 40;$

$T(n) = 40.$

2) Когда количество простых случаев относительно велико:

$C_w(n)$ – количество сравнений искомого ключа с элементами массива длины n в худшем случае.

(1) $C_w(n) = C_w(\lfloor n/2 \rfloor)+1$, если $n>1$ $\quad C_w(1) = 1$

Положим, сначала, что $n = 2^k$ и решаем (1) методом обратной подстановки. Так как $n = 2^k$, то при делении n/2 будем получать всегда целое число (надобность в функции $\lfloor n/2 \rfloor$ отпадает):

$C_w(n) = C_w(n/2)+1$, если $n>1$ $\quad C_w(1) = 1$

Осуществляем прямую подстановку:

$C_w(n/2) = C_w(n/4)+1$

$C_w(n/4) = C_w(n/8)+1$

$C_w(n/8) = C_w(n/16)+1$

$C_w(n/16) = C_w(n/32)+1$

…

Теперь делаем обратную подстановку в исходное соотношение:

$C_w(n) = (C_w(n/4)+1)+1 =$

$= ((C_w(n/8)+1)+1)+1 =$

$= (((C_w(n/16)+1)+1)+1)+1 =$

$= ((((C_w(n/32)+1) +1)+1)+1)+1 = C_w(n/32)+5 = C_w(n/32)+5 \cdot log_2 2$

$C_w(n) = C_w(n/32)+5 \cdot log_2 2 = C_w(n/32)+ log_2 2^5 = C_w(n/32)+ log_2 32$

Закончить понижение аргумента мы должны при его значении равным 1, то есть $n/n = 1$.

$C_w(n) = C_w(n/n)+ log_2 n$ и так как $C_w(1) = 1$ следовательно

(2) $C_w(n) = 1+ log_2 n$

Докажем теперь, что (2) с небольшими изменениями:

(3) $C_w(n) = 1+\lfloor log_2 n \rfloor$

Будет решением (1) для любого положительного целого n.

Убедимся с помощью непосредственной подстановки, что (3) удовлетворяет (1) для любого положительного четного $n = 2 \cdot i$ ($i>0$).

Левая часть (1):

$$C_w(n) = 1+\lfloor log_2 2 \cdot i \rfloor = 1+\lfloor log_2 2+log_2 i \rfloor = 2+\lfloor log_2 i \rfloor$$

Правая часть (1):

$$C_w(\lfloor n/2 \rfloor)+1 = C_w(\lfloor (2 \cdot i)/2 \rfloor)+1 = C_w(i)+1 = (1+\lfloor log_2 i \rfloor)+1 = 2+\lfloor log_2 i \rfloor$$

Оба выражения равны, что доказывает наше утверждение.

Докажем теперь, что (3) является решением (1) для любого положительного нечетного $n = 2 \cdot i - 1$ ($i>0$).

Левая часть (1):

$$C_w(n) = 1+\lfloor log_2(2 \cdot i - 1) \rfloor = log_2 2+\lfloor log_2(2 \cdot i - 1) \rfloor =$$

$$= \lfloor log_2 2 \cdot (2 \cdot i - 1) \rfloor = \lfloor log_2 (4 \cdot i - 1) \rfloor$$

Правая часть (1):

$$C_w(\lfloor n/2 \rfloor)+1 = C_w(\lfloor (2 \cdot i - 1)/2 \rfloor)+1 = (1+\lfloor log_2((2 \cdot i - 1)/2) \rfloor)+1 =$$

$$= \lfloor 2+log_2((2 \cdot i - 1)/2) \rfloor = \lfloor log_2 2^2+log_2((2 \cdot i - 1)/2) \rfloor =$$

$$= \lfloor log_2(4 \cdot (2 \cdot i - 1)/2) \rfloor = \lfloor log_2 (4 \cdot i - 1) \rfloor$$

Оба выражения равны, утверждение доказано.

Таким образом, $C_w(n)=1+\lfloor log_2 n \rfloor$ является решением исходного рекуррентного соотношения для любого целого $n>0$.

4. P, NP и NP-полные задачи.

При изучении вычислительной сложности задач первое, на что следует обратить внимание – может ли данная задача быть решена при помощи некоторого алгоритма за полиномиальное время.

Говорят, что алгоритм решает задачу за полиномиальное время, если его временная эффективность в наихудшем случае принадлежит классу O(p(n)), где p(n) – полином от размера входных данных n.

Задачу, которая может быть решена за полиномиальное время называют ***легкой***, а задачу, которая не может быть решена за полиномиальное время – ***трудной***.

Мы не можем решить трудные задачи за реальное время, за исключением очень малых размеров входных данных.

Хотя время работы может сильно отличаться для различных степеней полинома, имеется очень мало практичных полиномиальных алгоритмов со степенью полинома больше трех. Кроме того, полиномы, обычно не содержат очень больших коэффициентов.

Полиномиальные функции обладают многими удобными свойствами; в частности, как сумма, так и композиция двух полиномов всегда остаются полиномами.

Говоря не строго, задачи, решаемые за полиномиальное время, можно рассматривать как множество, которое в теории вычислительной сложности называется P.

Более формально, в множество P включаются только задачи ***принятия решения***, представляющие собой задачи, ответ на которые – «ДА» либо «НЕТ».

Класс P представляет собой класс задач принятия решения, которые могут быть решены (детерминистическим) алгоритмом за полиномиальное время. Этот класс задач называется ***полиномиальным***.

Многие важные задачи, не являющиеся задачами принятия решения в своей естественной формулировке, могут быть приведены к ряду задач принятия решения, которые проще изучать.

Например, вместо выяснения, какое наименьшее количество цветов надо для раскраски вершин графа так, чтобы никакие две смежные вершины не были одного цвета, мы можем выяснить, существует ли такая раскраска вершин графа не более чем m цветами при m=1,2,… .Первое значение m в этом ряду для которого задача раскраски имеет решение, дает ответ на оптимизационную задачу раскраски графа.

Возникает вопрос: каждая ли задача принятия решения может быть решена за полиномиальное время? Ответ – нет. Некоторые задачи принятия решения не могут быть решены в принципе, никакими алгоритмами (***неразрешимые задачи***). Пример – задача останова (Тьюринг, 1936 г.): для данной компьютерной программы и входных данных определить, завершится ли выполнение программы или она будет выполняться бесконечно?

Второй вопрос: существуют ли задачи разрешимые, но трудные? Ответ – да. Но количество известных примеров невелико, в особенности тех, которые возникают естественным путем, а не построены в качестве теоретического доказательства.

Имеется, однако, большое количество важных задач, для которых не найден алгоритм с полиномиальным временем работы, но и не доказана невозможность его существования (несколько сотен из различных областей информатики, математики и исследования операций).

Вот несколько из наиболее известных задач:

1. **Гамильтонов цикл**. Определить, имеется ли в данном графе гамильтонов цикл (путь, который начинается и заканчивается в одной и той же вершине и проходит по всем остальным вершинам ровно по одному разу).
2. **Задача коммивояжера**. Найти кратчайший маршрут по n городам с известными расстояниями между ними (найти кратчайший гамильтонов цикл в полном графе с положительными целыми весами).
3. **Задача о рюкзаке**. Найти подмножество с наибольшей стоимостью из n предметов с заданными положительными целыми весами и стоимостями, которое может быть помещено в рюкзак с заданной положительной целой емкость (грузоподъемностью).
4. **Задача о разделении**. Даны n положительных целых чисел; требуется определить, можно ли разделить их на два непересекающихся подмножества с одинаковыми суммами.
5. **Упаковка корзин**. Даны n предметов, размеры которых представляют собой положительные рациональные числа, не превышающие 1. Их надо разместить в наименьшее количество корзин размером 1.
6. **Раскраска графа (карты)**. Для данного графа найти его хроматическое число (наименьшее количество цветов, которыми можно раскрасить вершины графа так, чтобы никакие две смежные вершины не были окрашены в один и тот же цвет).
7. **Целочисленное линейное программирование**. Найти максимальное (минимальное) значение функции нескольких целочисленных переменных при условии выполнения конечного множества ограничений в виде линейных равенств и/или неравенств.

Некоторые из этих задач являются задачами принятия решения. Общее у всех этих задач то, что они обладают экспоненциальным (или еще более быстрым) ростом количества вариантов выбора решения с ростом размерности задачи n.

Заметим, что имеется ряд задач с очень похожими условиями, но решаемых за полиномиальное время. Например, **задача об Эйлеровом цикле**, то есть, о существовании цикла, который проходит по всем ребрам данного графа ровно по одному разу, может быть решена за время $O(n^2)$ путем проверки связности графа и все ли его вершины имеют четную степень (**Задача о мостах в Кенигсберге**).

Этот пример поразителен: интуитивно, совершенно не ожидаешь, что задача обхода всех ребер по одному разу (Эйлеров цикл) настолько проще кажущейся похожей задачи о цикле, обходящем по одному разу все вершины (гамильтонов цикл).

Еще одно общее свойство огромного большинства задач принятия решения заключается в том, что при том, что решение задач может быть вычислительно сложным, проверка предложенного решения обычно достаточно проста и может быть выполнена за полиномиальное время (такие решения можно представить как случайным образом генерируемые кем-то и предлагаемые нам для проверки их корректности).

Например, для того чтобы проверить, что предложенный список вершин является гамильтоновым циклом для данного графа с n вершинами, необходимо убедиться, что список содержит n+1 вершину графа, что первые n вершин в списке различны, а последняя совпадает с первой, и что каждая пара соседних вершин в списке соединяется ребром.

Это свойство приводит нас к понятию недетерминистического алгоритма. ***Недетерминистическим алгоритмом*** называется двух этапная процедура, которая получает в качестве входа экземпляр I задачи принятия решения и делает следующее:

- недетерминистический этап («угадывание»): генерируется произвольная строка S, которая может рассматриваться как кандидат в решение данной задачи I (но может оказаться и полной ерундой);

– детерминистический этап («проверка»): получает *I* и *S* в качестве входных данных и выдает «да», если *S* является решением *I*. (Если *S* не является решением *I*, алгоритм либо возвращает «нет», либо может вообще не завершить работу).

Недетерминистический алгоритм является недетерминистическим полиномиальным, если временная эффективность этапа проверки полиномиальная.

Класс NP – это класс задач принятия решения, которые могут быть решены недетерминистическим полиномиальным алгоритмом (nondeterministic polynomial).

P $\subseteq$ NP (для P на этапе проверки просто игнорируем строку *S*, генерируемую на этапе угадывания).

Большинство задач принятия решения принадлежат классу NP. Задача останова не принадлежит классу NP.

Важный открытый вопрос теоретической информатики: является ли класс P собственным подмножеством NP или эти классы совпадают?

Задача принятия решения D_1 называется ***полиномиально приводимой*** к задаче принятия решения D_2, если имеется функция *t*, которая преобразует экземпляры D_1 в экземпляры D_2 так, что

1. *t* отображает все экземпляры D_1 с положительным ответом на экземпляры D_2 с положительным ответом, и все экземпляры D_1 с отрицательным ответом на экземпляры D_2 с отрицательным ответом.
2. *t* – вычислима при помощи алгоритма с полиномиальным временем работы.

Если задача D_1 полиномиально приводима к некоторой задаче D_2, которая может быть решена за полиномиальное время, то задача D_1 также может быть решена за полиномиальное время.

Задача принятия решения D называется NP-полной, если:

1. она принадлежит классу NP;
2. любая задача в NP полиномиально приводима к D.

Понятие NP-полноты требует полиномиальной приводимости ***всех*** задач в NP, как известных, так и неизвестных, к рассматриваемой задаче.

При огромном количестве задач принятия решения вызывает изумление тот факт, что были найдены конкретные примеры NP-полных задач.

Этого добились независимо друг от друга Стивен Кук (в 1971 году) и Леонид Левин (в 1973 году). Они показали, что задача КНФ-выполнимости является NP-полной. (КНФ (CNF) – конъюнктивная нормальная форма булева выражения). Каждое булево выражение может быть представлено в КНФ. Спрашивается, можно ли назначить в булевом выражении переменным (*true* или *false*) так, чтобы результат вычисления всего выражения был *true*.

С того времени найдены сотни примеров NP-полных задач. Некоторые из них были приведены выше. Известно, однако, что если P≠NP, то должны существовать NP-задачи, которые не являются ни P-задачами, ни NP-полными задачами.

Непосредственно из определения NP-полноты следует, что если будет найден детерминистический алгоритм решения одной из NP-полных задач, то все задачи в NP могут быть решены за полиномиальное время при помощи детерминистического алгоритма, следовательно, P=NP.

Исчерпывающий перебор.

Исчерпывающим перебором называется поиск элемента со специфическими свойствами в области экспоненциально (или еще быстрее) растущей с увеличением размера задачи. Обычно такие задачи возникают в случаях, когда присутствуют – явно или не явно – комбинаторные объекты: перестановки, сочетания, подмножества.

Многие подобные задачи являются задачами оптимизации: в них требуется найти элемент, который максимизирует (минимизирует) некоторую целевую функцию как, например, длина пути или стоимость.

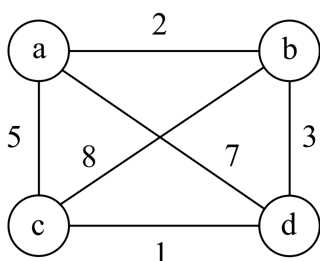
Исчерпывающий перебор представляет собой подход к комбинаторным задачам с позиции грубой силы. Он предполагает генерацию всех возможных элементов из области определения задачи, и последующий поиск нужного элемента (например, оптимизирующего значение целевой функции задачи).

Идея исчерпывающего перебора весьма проста, ее реализация обычно требует алгоритма для генерации определенных комбинаторных объектов.

Задача коммивояжера.

Под задачей коммивояжера понимается поиск кратчайшего гамильтонова цикла неориентированного графа.

Гамильтонов цикл в графе содержащем ровно n вершин можно определить как последовательность $n+1$ смежных вершин v_{i0}, v_{i1}, …, v_{in-1}, v_{i0}, где первая вершина в последовательности совпадает с последней, в то время как все остальные $n–1$ вершин различны. Далее без потери общности можно предположить, что все циклы начинаются и заканчиваются в одной конкретной вершине (в конце концов, все они циклы). Значит, можно получить все возможные маршруты, генерируя все перестановки $n–1$ промежуточных городов, вычисляя длину соответствующих путей и находя кратчайший из них.



Путь	Длина
a → b → c → d → a	2 + 8 + 1 + 7 = 18

$a \to b \to d \to c \to a$ $2 + 3 + 1 + 5 = 11$

$a \to c \to b \to d \to a$ $5 + 8 + 3 + 7 = 23$

$a \to c \to d \to b \to a$ $5 + 1 + 3 + 2 = 11$

$a \to d \to b \to c \to a$ $7 + 3 + 8 + 5 = 23$

$a \to d \to c \to b \to a$ $7 + 1 + 8 + 2 = 18$

При внимательном рассмотрении можно выявить три пары обходов, которые отличаются друг от друга только направлением. Таким образом, можно снизить количество перестановок вершин вдвое.

Можно, например, выбрать две промежуточные вершины, скажем b и c, и рассматривать только те перестановки, в которых b предшествует c (это неявно определяет направление обхода).

Однако это улучшение алгоритма не дает заметного эффекта. Общее количество перестановок остается равным $(n-1)!/2$, что делает исчерпывающий перебор неприемлемым для всех значений n, кроме самых малых.

Заметим, что если бы не ограничение, чтобы все пути начинались в одной и той же вершине, то количество перестановок было бы в n раз больше.

Задача о рюкзаке.

Дано n предметов весом $w_1, \ldots, w_n$ и ценой $v_1, \ldots, v_n$, а также рюкзак, выдерживающий вес W. Требуется найти подмножество предметов, которое можно разместить в рюкзаке, и которое имеет при этом максимальную стоимость.

Исчерпывающий перебор в этой задаче приводит к рассмотрению всех подмножеств данного множества из n элементов, вычислению общего веса каждого из них для того, чтобы выяснить, допустим ли такой набор предметов (не превосходит ли его общий вес возможности рюкзака), и выбору из допустимых подмножества с максимальным весом.

Общее количество подмножеств $\boldsymbol{n}$-элементного множества равно $2^{\boldsymbol{n}}$, исчерпывающий перебор приводит к алгоритму со временем работы $\mathbf{O}(2^{\boldsymbol{n}})$, вне зависимости от того, насколько эффективным методом генерируются рассматриваемые подмножества.

Предметы:	1)	w = 7	v = 42
	2)	w = 3	v = 12
	3)	w = 4	v = 40
	4)	w = 5	v = 25

Грузоподъемность рюкзака W = 10

Подмножество	Общий вес	Общая стоимость	
∅	0	0	
{1}	7	42	
{2}	3	12	
{3}	4	40	
{4}	5	25	
{1, 2}	10	36	
{1, 3}	11	недопустим	
{1, 4}	12	недопустим	
{2, 3}	7	52	
{2, 4}	8	37	
{3, 4}	**9**	**65**	**ответ**
{1, 2, 3}	14	недопустим	
{1, 2, 4}	15	недопустим	
{1, 3, 4}	16	недопустим	
{2, 3, 4}	12	недопустим	

{1, 2, 3, 4}	19	недопустим

Задача о назначениях.

Имеется n работников, которые должны выполнить n заданий, по одному заданию каждый. Стоимость выполнения i-ым работником j-го задания известна и равна C_{ij} для всех пар i и j.

Надо распределить задания между работниками таким образом, чтобы они были выполнены с наименьшей общей стоимостью.

Легко видеть, что экземпляр задачи с назначением заданий полностью определяется матрицей стоимости C.

В терминах данной матрицы требуется выбрать по одному элементу из каждой строки матрицы так, чтобы выбранные элементы находились в разных столбцах, а их общая сумма имела наименьшее возможное значение.

Заметим, что очевидной стратегии решения данной задачи не существует. Например, нельзя выбирать наименьшие элементы в каждой строке, так как они могут оказаться в одном и том же столбце. В действительности, наименьшие элементы матрицы могут вообще не входить в оптимальное решение. Так что исчерпывающий перебор для решения задачи может оказаться неизбежным.

Допустимое решение можно описать в виде кортежа из n значений $\langle j_1, \ldots, j_n \rangle$, в котором i-ый компонент указывает столбец, где находится выбранный в i-ой строке элемент матрицы.

	Задание 1	Задание 2	Задание 3	Задание 4
Работник 1	9	2	7	8
Работник 2	6	4	3	7
Работник 3	5	8	1	8
Работник 4	7	6	9	4

Кортеж $\langle 2, 3, 4, 1\rangle$ указывает допустимое назначение:

2 задание работнику 1

3 задание работнику 2

4 задание работнику 3

1 задание работнику 4

Из условия задачи вытекает, что имеется однозначное соответствие между допустимыми назначениями и перестановками первых ***n*** натуральных чисел.

Следовательно, исчерпывающий перебор потребует генерации всех перестановок натуральных чисел 1, 2, …, ***n***, вычисления общей стоимости и выбора назначения с минимальной стоимостью.

Перестановка	Стоимость	
$\langle 1, 2, 3, 4\rangle$	9 + 4 + 1 + 4 = 18	
$\langle 1, 2, 4, 3\rangle$	9 + 4 + 8 + 9 = 30	
$\langle 1, 3, 2, 4\rangle$	9 + 3 + 8 + 4 = 24	
$\langle 1, 3, 4, 2\rangle$	9 + 3 + 8 + 6 = 26	
$\langle 1, 4, 2, 3\rangle$	9 + 7 + 8 + 9 = 33	
$\langle 1, 4, 3, 2\rangle$	9 + 7 + 1 + 6 = 23	
$\langle 2, 1, 3, 4\rangle$	**2 + 6 + 1 + 4 = 13**	**ответ**
$\langle 2, 1, 4, 3\rangle$	2 + 6 + 8 + 9 = 25	
$\langle 2, 3, 1, 4\rangle$	2 + 3 + 5 + 4 = 14	
$\langle 2, 3, 4, 1\rangle$	2 + 3 + 8 + 7 = 20	
$\langle 2, 4, 1, 3\rangle$	2 + 7 + 5 + 9 = 23	
$\langle 2, 4, 3, 1\rangle$	2 + 7 + 1 + 7 = 17	
$\langle 3, 1, 2, 4\rangle$	7 + 6 + 8 + 4 = 25	
$\langle 3, 1, 4, 2\rangle$	7 + 6 + 8 + 6 = 27	
$\langle 3, 2, 1, 4\rangle$	7 + 4 + 5 + 4 = 20	
$\langle 3, 2, 4, 1\rangle$	7 + 4 + 8 + 7 = 26	

$\langle 3, 4, 1, 2\rangle$	7 + 7 + 5 + 6 = 25
$\langle 3, 4, 2, 1\rangle$	7 + 7 + 8 + 7 = 29
$\langle 4, 1, 2, 3\rangle$	8 + 6 + 8 + 9 = 31
$\langle 4, 1, 3, 2\rangle$	8 + 6 + 1 + 6 = 21
$\langle 4, 2, 1, 3\rangle$	8 + 4 + 5 + 9 = 26
$\langle 4, 2, 3, 1\rangle$	8 + 4 + 1 + 7 = 20
$\langle 4, 3, 1, 2\rangle$	8 + 3 + 5 + 6 = 22
$\langle 4, 3, 2, 1\rangle$	8 + 3 + 8 + 7 = 26

5. Алгоритмы генерации комбинаторных объектов.

Наиболее важные типы комбинаторных объектов – перестановки и подмножества данного множества. В математике, в первую очередь, интересуются формулами для подсчета количества таких объектов. Эти формулы предупреждают нас о том, что это количество с увеличением размерности задачи растет экспоненциально или еще быстрее.

Генерация перестановок.

Дано множество натуральных чисел от 1 до n. В общем случае их можно интерпретировать как индексы элементов n-элементного множества $\{a_1, \ldots, a_n\}$.

Нам нужно сгенерировать n! перестановок множества {1, 2, …, n}.

Применим метод уменьшения размерности задачи.

Задача меньшего на 1 размера состоит в генерации всех (n-1)! перестановок. Полагая, что задача меньшего размера успешно решена, мы можем получить решение большей задачи путем вставки n в каждую из n возможных позиций среди элементов каждой из перестановок n-1 элементов.

Мы можем вставлять n в ранее сгенерированные перестановки слева направо или справа налево. Оказывается, выгодно начинать вставку n в

последовательность 1 2 3 … (n-1) *справа налево* и изменять направление всякий раз при переходе к новой перестановке множества {1, 2, …, n-1}.

Пример:

Начало	1		
Вставка 2:			
справа налево	1 **2**	**2** 1	
Вставка 3:			
справа налево	1 2 **3**	1 **3** 2	**3** 1 2
слева направо	**3** 2 1	2 **3** 1	2 1 **3**

Преимущество такого порядка генерации – он удовлетворяет ***требованию минимальных изменений***: каждая перестановка получается из своей непосредственной предшественницы при помощи обмена местами только двух элементов.

Требование минимальных изменений выгодно как с точки зрения скорости работы алгоритма, так и для приложения использующего перестановки. Например, в задаче коммивояжера нам нужны перестановки городов. Если такие перестановки генерируются алгоритмом, удовлетворяющим требованию минимальных изменений, то мы можем вычислить длину нового маршрута на основании длины старого маршрута за *постоянное*, а не линейное время (как ?).

Можно получить тот же порядок перестановок n элементов и без явной генерации перестановок для меньших значений n. Это можно сделать, связав с каждым компонентом k перестановки так называемое ***направление***. Будем указывать это направление при помощи стрелки:

3241(→←→←)

Компонент k в такой перестановке с использованием стрелок называется ***мобильным***, если стрелка указывает на меньшее соседнее число. Например, в

числа 3 и 4 мобильны, а 2 и 1 – нет.

Воспользовавшись понятием мобильного компонента, мы получим следующий

алгоритм Джонсона-Троттера.

<u>Входные данные</u>: n– натуральное число

<u>Выходные данные</u>: список перестановок множества {1, 2, …, n}(

1) Инициализируем первую перестановку: 12 … n(←← … ←)

2) Пока имеется мобильное число k выполнять

3) Найти наибольшее мобильное число k

4) Поменять местами k и соседнее число, на которое указывает стрелка у k

5) Поменять направлении стрелок у всех чисел, больших k

Пример для n = 3

12**3**(←←←) 1**3**2(←←←) 31**2**(←←←) **3**21(→←←) 2**3**1(←→←) 213(←←→)

Этот алгоритм – один из наиболее эффективных для генерации всех перестановок (хотя все они имеют оценку O(n!)).

Порядок перестановок, генерируемых алгоритмом Джонсона-Троттера, не совсем естественный: было бы естественным ожидать, что последняя перестановка в списке будет иметь вид: n (n-1) … 2 1. Именно такая перестановка окажется последней, если перестановки будут упорядочены в соответствии с лексикографическим порядком.

1 2 3   1 3 2   2 1 3   2 3 1   3 1 2   3 2 1

Каким образом мы можем сгенерировать перестановку, следующую за $a_1, a_2, \ldots, a_{n-1}, a_n$ в лексикографическом порядке?

Если $a_{n-1} > a_n$, мы должны обратиться к элементу a_{n-2}. Если $a_{n-2} < a_{n-1}$, мы должны переставить последние три элемента, минимально увеличивая (n-2)-ой элемент, т.е. помещая на это место следующий больший a_{n-2} элемент, выбранный из a_{n-1} и a_n, и заполняя позиции (n-1) и (n) оставшимися двумя из трех элементов a_{n-2}, a_{n-1}, a_n в возрастающем порядке.

В общем случае мы просматриваем текущую перестановку справа налево в поисках первой пары соседних элементов a_i и a_{i+1} таких, что $a_i<a_{i+1}$ (и, следовательно, $a_{i+1}> \ldots >a_n$). Затем мы находим наименьший элемент из «хвоста», больший a_i, т.е. $min\{a_j|a_j>a_i,\ j>i\}$, и помещаем его в позицию i; позиции с $i+1$-ой по n-ую заполняются элементами a_i, a_{i+1}, …, a_n, из которых изъят элемент для вставки в позицию i, в возрастающем порядке.

Пример.

$1\ 2\ 3\ \underline{4\ 5}$

$1\ 2\ \underline{3\ 5}\ 4$

$1\ 2\ 4\ \underline{3\ 5}$

$1\ 2\ \underline{4\ 5}\ 3$

$1\ 2\ 5\ \underline{3\ 4}$

$1\ \underline{2\ 5}\ 4\ 3$

$1\ 3\ 2\ \underline{4\ 5}$

$1\ 3\ \underline{2\ 5}\ 4$

$1\ 3\ 4\ \underline{2\ 5}$

$1\ 3\ \underline{4\ 5}\ 2$

$1\ 3\ 5\ \underline{2\ 4}$

$1\ \underline{3\ 5}\ 4\ 2$

$1\ 4\ 2\ \underline{3\ 5}$

и т.д.

Генерация подмножеств.

Рассмотрим алгоритм для генерации всех 2^n подмножеств абстрактного множества $A = \{a_1, a_2, \ldots, a_n\}$.

К этой задаче также применим метод уменьшения размерности на единицу. Все подмножества множества $A = \{a_1, a_2, \ldots, a_n\}$ разделить на две группы – те, которые содержат элемент a_n, и те, которые не содержат его. Первая группа представляет собой не что иное, как все подмножества множества $\{a_1, a_2, \ldots, a_{n-1}\}$, в то время как все элементы второй группы можно получить путем добавления элемента a_n к каждому подмножеству множества $\{a_1, a_2, \ldots, a_{n-1}\}$. Следовательно, как только мы получим список всех подмножеств множества $\{a_1, a_2, \ldots, a_{n-1}\}$, мы можем получить все подмножества множества $\{a_1, a_2, \ldots, a_n\}$, просто добавляя к списку все его элементы с добавление к каждому из них элемента a_n.

Пример 1. Сгенерируем все подмножества множества $\{a_1, a_2, a_3\}$

n	Сгенерированные подмножества
0	$\varnothing$
1	$\varnothing$ $\{a_1\}$
2	$\varnothing$ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$
3	$\varnothing$ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$ $\{a_3\}$ $\{a_1, a_3\}$ $\{a_2, a_3\}$ $\{a_1, a_2, a_3\}$

Как и в случае перестановок, мы не обязаны генерировать все подмножества для множеств меньших размеров. Удобный способ решения основан на взаимно однозначном соответствии между всеми 2^n подмножествами n-элементного множества $A = \{a_1, a_2, \ldots, a_n\}$ и всеми 2^n битовыми строками $b_1b_2\ldots b_n$ длины n.

Простейший способ установить такое соответствие – назначить подмножеству битовую строку, в которой b_i=1, если a_i принадлежит данному подмножеству, и b_i=0 в противном случае.

Используя такое соответствие, мы можем сгенерировать все битовые строки длиной n, просто генерируя последовательно двоичные числа от 0 до 2^n-1, добавляя при необходимости соответствующее количество ведущих нулей.

Пример 2. n = 3

Битовые строки

000 001 010 011 100 101 110 111

Подмножества

$\varnothing$ $\{a_3\}$ $\{a_2\}$ $\{a_2,a_3\}$ $\{a_1\}$ $\{a_1,a_3\}$ $\{a_1,a_2\}$ $\{a_1,a_2,a_3\}$

Заметим, что в то время, как битовые строки, сгенерированные данных алгоритмом, находятся в лексикографическом порядке (для алфавита {0, 1}), порядок подмножеств выглядит далеко не естественно.

Например, мы можем захотеть получить так называемый ***плотный порядок***, когда подмножество, включающее a_j, может находиться в списке только после всех подмножеств, включающих элементы a_1, a_2, …, a_{j-1}, как, например, в примере 1.

Более интересный вопрос – о существовании алгоритма генерации битовых строк, соответствующих требованию минимальных изменений, так, чтобы каждая строка отличалась от непосредственно предшествующей ей только одним битом (каждое подмножество будет отличаться от своего предшественника в списке либо добавлением, либо удалением одного элемента, но не более).

Это код Грея. При n = 3:

000 001 011 010 110 111 101 100.

Код Грея легко получить:

```
function Gray(n: integer): integer;
  begin
    Gray := n xor (n shl 1)
```

```
end;
```

Здесь используются две побитовые операции: исключающее или и сдвиг вправо на один разряд.