Модуль 4.
Рекурсия и сложность алгоритмов

Рекурсия и ее реализация

Рекурсия реализуется на основе подпрограммы, вызывающей саму себя. Первый вызов рекурсивной подпрограммы выполняется из внешней программы.

Типы рекурсии:

- прямая (функция содержит вызов самой себя);
- косвенная (функция F1 вызывает функцию F2, которая вызывает исходную F1)

Этапы разработки рекурсивного алгоритма решения задачи

- 1) *параметризация задачи*, т.е. выявление различных элементов, от которых зависит ее решение, с целью нахождения управляющего параметра. При этом размерность управляющего параметра должна убывать после каждого рекурсивного вызова по мере нахождения решения;
- 2) *поиск тривиального случая и его решения*. На этом этапе должно быть найдено решение задачи без рекурсии;
- 3) *декомпозиция общего случая*, требующая привести его к одной или нескольким более простым задачам меньшей размерности.

Рекурсивный алгоритм вычисления факториала

Формула факториала:

$n! = n*(n-1)*(n-2)*\ldots*1$

- управляющий параметр -текущее значение числа *n*.
- Тривиальный случай представляет собой 0! = 1
- декомпозиция общего случая

  $n! = n*(n-1)!$

Рекурсивная программа вычисления факториала

```
#include "stdafx.h"
#include <conio.h>
long int fact(int i)
{
  if (i==0) return 1;
  else return i*fact(i-1);
}
int _tmain(int argc, _TCHAR* argv[])
{
  long int f;
  int n;
  printf("\nInput n");
  scanf("%d",&n);
  f=fact(n);
  printf("%i",f);
  getch();
  return 0;
}
```

Работа рекурсивной программы со стеком

Текущий уровень рекурсии	Рекурсивный спуск	Рекурсивный возврат
0	Ввод n=5. f=fact(5)	Вывод 120
1	i=5 fact(5)=5*fact(4)	fact(5)=5*24=120
2	i=4 fact(4)=4*fact(3)	fact(4)=4*6=24
3	i=3 fact(3)=3*fact(2)	fact(3)=3*2=6
4	i=2 fact(2)=2*fact(1)	fact(2)=2*1*1=2
5	i=1 fact(1)=1*fact(0)	fact(1)=1*1=1
6	i=0 fact=1	

Адрес возврата
5

Стек при первом вызове fact (i=5)

Стек при уровне рекурсии 4

Адрес возврата
2
Адрес возврата
3
Адрес возврата
4
Адрес возврата
5

Роль стека при вызове подпрограммы B из A

1. В вершину стека помещается фрагмент нужного размера. В него входят следующие данные:

(а) указатели фактических параметров при вызове процедуры B;

(б) пустые ячейки для локальных переменных, определенных в процедуре B;

(в) адрес возврата, т.е. адрес команды в процедуре A, которую следует выполнить после того, как процедура B закончит свою работу.

Если B - функция, то во фрагмент стека для B помещается указатель ячейки во фрагменте стека для A, в которую надлежит поместить значение этой функции (адрес значения).

2. Управление передается первому оператору процедуры B.
3. При завершении работы процедуры B управление передается процедуре A с помощью следующей последовательности шагов:

(а) адрес возврата извлекается из вершины стека

(б) если B функция, то ее значение запоминается в ячейке, предписанной указателем на адрес значения;

(в) фрагмент стека процедуры B извлекается из стека, в вершину ставится фрагмент процедуры A;

(г) выполнение процедуры A возобновляется *с* команды, указанной в адресе возврата. При вызове подпрограммой самой себя, т.е. в рекурсивном случае, выполняется та же самая последовательность действий.

Рекурсивное вычисление чисел Фибоначчи

F(1)=1,F(2)=1,для любого n>2 F(n)=F(n-1)+F(n-2)

```
#include "stdafx.h"
#include <conio.h>
long int fib(long int i)
{
if ((i==1)||(i==2)) return 1;
 else return fib(i-1)+fib(i-2);
}
```

```
int _tmain(int argc, _TCHAR*
argv[])
{
  int n;
  long int ch;
  printf("Input number:");
  scanf("%d",&n);
  ch=fib(n);
  printf("\nFibonachi with number
     %d is %i\n",n,ch);
  getch();
return 0;
}
```

Задача «Ханойские башни»

Существует легенда: «В храме Бендареса находится бронзовая плита с тремя алмазными стержнями. На один из стержней бог при сотворении мира нанизал 64 диска разного диаметра из чистого золота. Наибольший диск лежит на бронзовой плите и с остальными образует пирамиду, сужающуюся кверху. Это – башня Брамы. Работая день и ночь, жрецы переносят диски с одного стержня на другой, следуя законам Брамы:

- 1) диски можно перемещать с одного стержня на другой только по одному;
- 2) нельзя класть больший диск на меньший;
- 3)при переносе дисков с одного стержня на другой можно использовать промежуточный третий стержень, на котором диски тоже могут находиться тоже только в виде пирамиды.
- Когда все 64 диска будут перенесены с одного стержня на другой, наступит конец света».

Алгоритм решения задачи «Ханойские башни»

Назовем стержни левым (left), средним (middle) и правым (right). Задача состоит в переносе m дисков с левого стержня на правый. Задача может быть решена одним перемещением только для одного (m = 1) диска.

Построим рекурсивное решение задачи, состоящее из трех этапов:

- a) перенести башню, состоящую из m – 1 диска, с левого стержня на средний;
- b) перенести **один** оставшийся диск с левого стержня на правый;
- c) перенести башню, состоящую из m – 1 диска, со среднего стержня на правый.

Программа «Ханойские башни»: обозначения

- Обозначим тот стержень, с которого следует снять диски, через **s1**, на который надеть – через **sk**, а вспомогательный стержень через **sw**.
- Оформим алгоритм решения задачи о переносе башни из n дисков с s1 на sk в виде процедуры **move** с 4-мя параметрами: n, s1, sw, sk;
- алгоритм для n = 1 выделим в отдельную процедуру **step**, которая перемещает один диск со стержня s1 на sk.

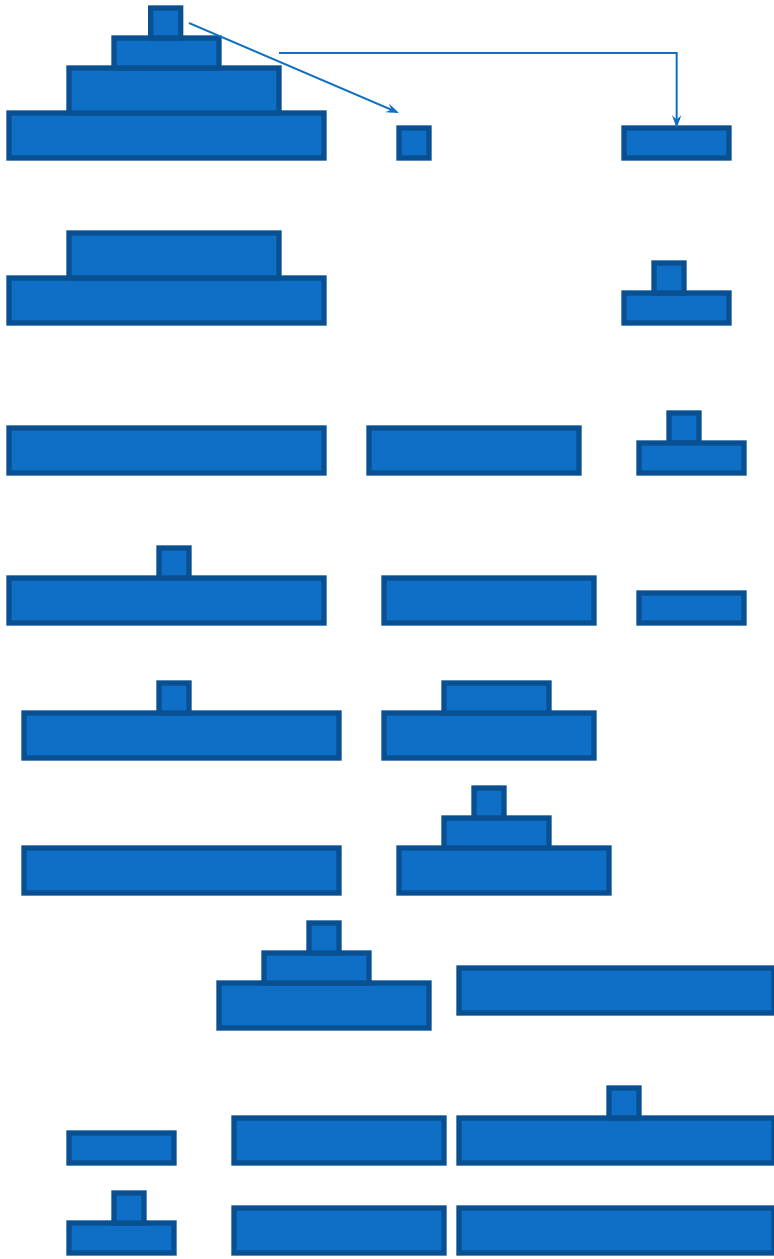
```
#include "stdafx.h"
#include <conio.h>
enum st {left,middle,right};
void name(st ster)
{
  switch (ster){
  case 0: printf(" left ");break;
  case 1: printf(" middle ");break;
  case 2: printf(" right ");break;
  };
}
void step(st si,st sk)
{

  printf("\ntake disk from");
  name(si);
  printf(",put to");
  name(sk);
}
```

```
void move(int n,st si,st sw,st sk)
{
 if (n==1) step(si,sk);
 else
 {
        move(n-1,si,sk,sw);
        step(si,sk);
        move(n-1,sw,si,sk);
 }
}

int _tmain(int argc, _TCHAR*
argv[])
{
int n;
printf("\nInput n:");
scanf("%d",&n);
move(n,left,middle,right);
getch();
return 0;
}
```

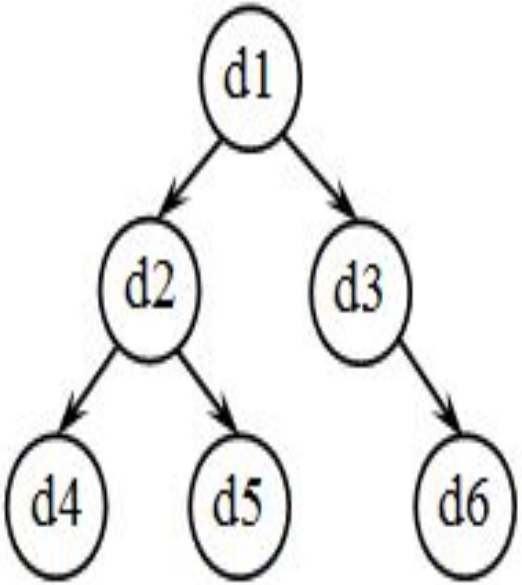
E:\Учитель\бакалавриатИБ\математичес...

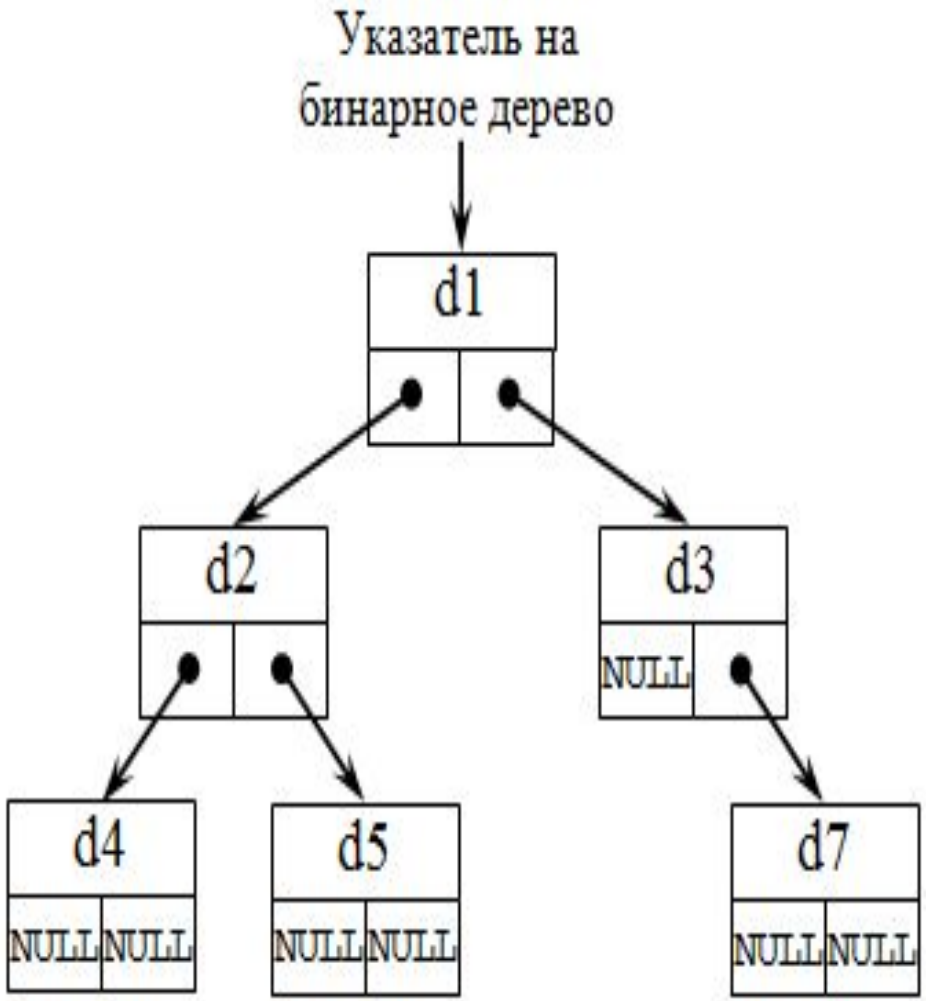
E:\Учитель\бакалавриат\ИБ

```
Input n:4

take disk from left ,put to middle
take disk from left ,put to right
take disk from middle ,put to right
take disk from left ,put to middle
take disk from right ,put to left
take disk from right ,put to middle
take disk from left ,put to middle
take disk from left ,put to right
take disk from middle ,put to right
take disk from middle ,put to left
take disk from right ,put to left
take disk from middle ,put to right
take disk from left ,put to middle
take disk from left ,put to right
take disk from middle ,put to right
```



Хранение бинарных деревьев



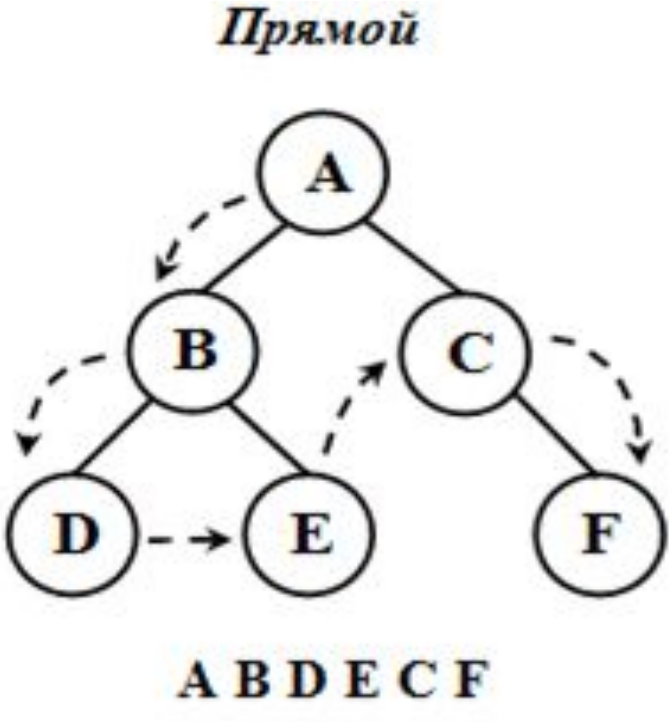


Обходы бинарных деревьев

- Прямой (корень – левое поддерево - правое поддерево);
- обратный (левое поддерево – правое поддерево – корень) ;
- симметричный (левое – корень – правое)

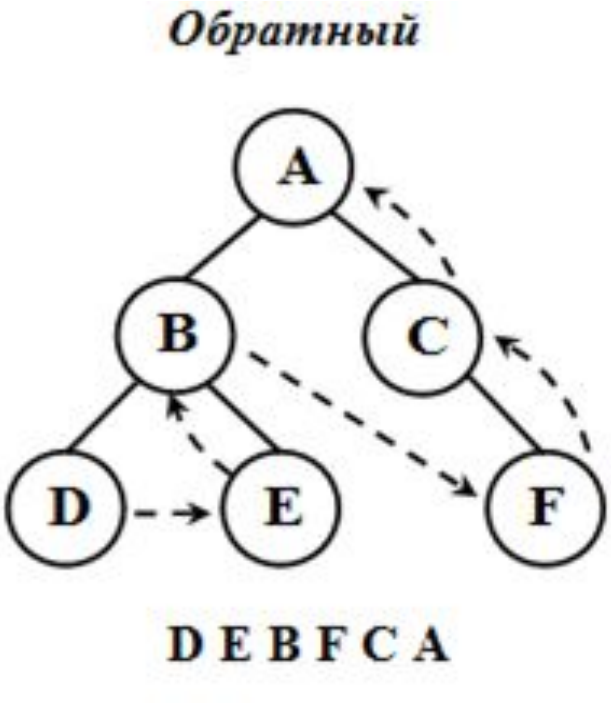
```
struct tree {
  char info;
  struct tree *left;
  struct tree *right;
};
```

Прямой обход



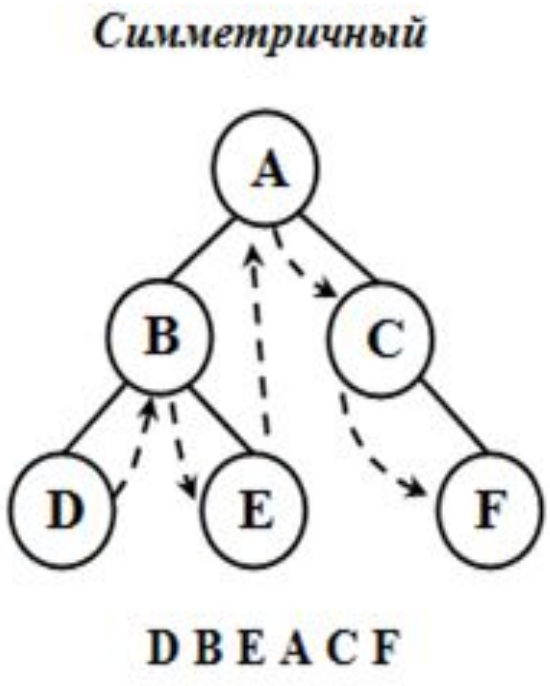

```
void pr(struct tree *root)
{
  if(!root) return;

  if(root->info) printf("%c ",
                        root->info);
  pr(root->left);
  pr(root->right);
}
```

Обратный обход




```
void obr(struct tree *root)
{
  if(!root) return;

  obr(root->left);
  obr(root->right);
  if(root->info) printf("%c ",
root->info);
}
```

Симметричный обход




```
void sim(struct tree
*root)
{
  if(!root) return;

  sim(root->left);
  if(root->info)
printf("%c ",
root->info);
  sim(root->right);
}
```

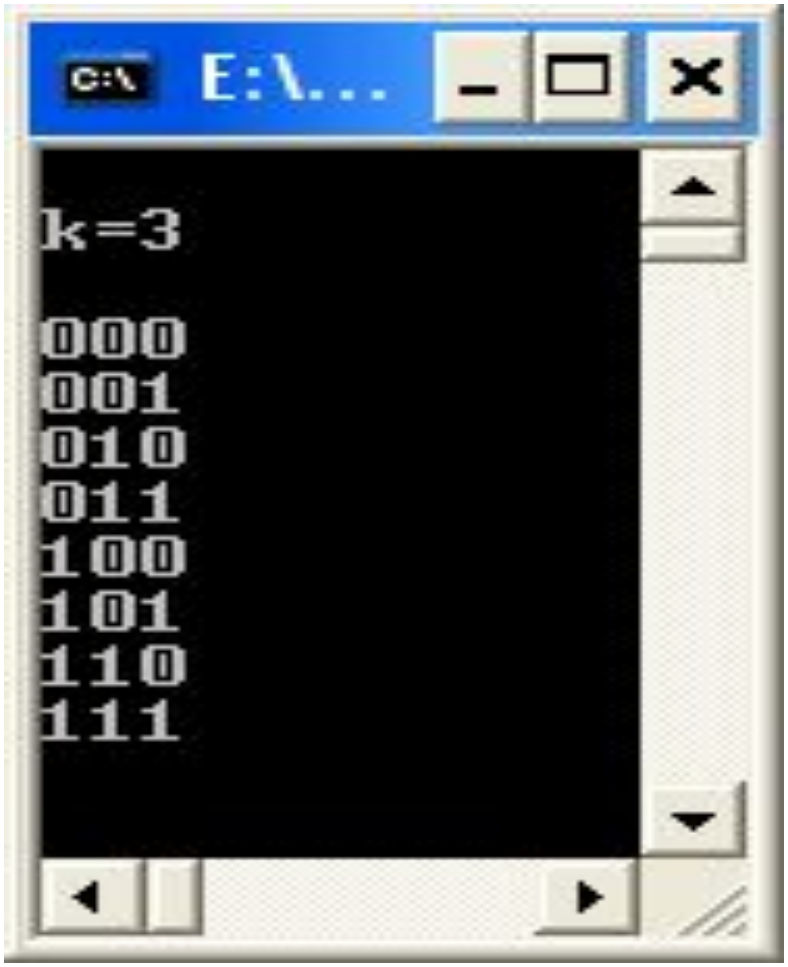
Переборные задачи.

Рассмотрим программу, осуществляющую вывод всех k-значных двоичных чисел (их всего, очевидно, 2^k)

```
#include "stdafx.h"
#include <conio.h>
int a[20];
int k;
void write_number()
{
    printf("\n");
    for (int j=0;j<k;j++)
    printf("%d",a[j]);
}
```

```
void find(int i)
{
 if (i==k) write_number();
 else
 {
        a[i]=0;
        find(i+1);
        a[i]=1;
        find(i+1);
 }
}
int _tmain(int argc, _TCHAR*
argv[])
{
        printf("\nk=");
        scanf("%d",&k);
        find(0);
        getch();
        return 0;
}
```

Пример работы программы для k=3



Если **i=k+1**, то find(k+1) запускает write_number для вывода и завершает работу, так как вызов find(k+1) обозначает, что первые k элементов массива уже заполнены.

Задача о расстановке n ферзей

```
for i1 := 1  to n  do begin
    ферзь[1] := i1;
    for i2 := 1  to n  do begin
        ферзь[2] := i2;
        for i3 := 1  to n  do begin
            ферзь[3] := i3;
            ...
                        <проверка построенной позиции>
        end;
    end;
end;
```

Отсечения при переборе

```
for i1 := 1  to n  do begin
    ферзь[1] := i1;
    for i2 := 1  to n  do begin
      ферзь[2] := i2;
            { проверка частично построенной позиции }
      if <ферзь[2] не бьёт ферзь[1]>  then
        for i3 := 1  to n  do begin
          ферзь[3] := i3;
         { проверка частично построенной позиции }
          if <ферзь[3] не бьёт предыдущих>  then
            for i4 := 1  to n  do begin
              ...
            if <ферзь[n] не бьёт предыдущих> then
                    <решение найдено>
            end;
        end;
    end;

end;
```

Простые примеры задач на динамическое программирование

```
Var D : Array [1..50] of LongInt;
    Function F(X : integer) : LongInt;
    Begin
     if D[X] = 0
        then
            if (X = 1) or (X = 2)
                then D[X] := 1
                else D[X] := F(x - 1) + F(x - 2);
            F := D[X]
    end;
```

Без рекурсии:

```
D[1] := 1; D[2] := 1;
For i := 3 to X do D[i] := D[i-1] + D[i-2];
```

Жадные алгоритмы

Жадный алгоритм (greedy algorithm) – это метод решения оптимизационных задач, основанный на том, что процесс решения задачи можно разбить на шаги, на каждом из которых принимается решение.

Решение, принимаемое на каждом шаге, должно быть оптимальным только на текущем шаге, и должно приниматься вне зависимости от решений на других шагах.

На каждом шаге «жадный» алгоритм выбирает локально оптимальное в том или ином смысле решение.

Дискретная и непрерывная задача о рюкзаке

Постановка дискретной задачи.

В рюкзак загружаются предметы n различных типов (количество предметов каждого типа не ограничено). Максимальный вес рюкзака W. Каждый предмет типа i имеет вес w_i и стоимость v_i (i=1,2, …, n). Требуется определить максимальную стоимость груза, вес которого не превышает W.

Обозначим количество предметов типа i через k_i, тогда требуется максимизировать $v_1*k_1+v_2*k_2+...+v_n*k_n$ при ограничениях $w_1*k_1+w_2*k_2+...+w_n*k_n \le W$, где k_i - целые ($0 \le k_i \le [W/w_i]$), квадратные скобки означают целую часть числа.

Непрерывная задача о рюкзаке

Жадный алгоритм:

1. Вычислим цены всех предметов - стоимость предмета разделим на массу (Price[i]/Weight[i]).
2. Сначала берем самый дорогой предмет и наполняем им рюкзак, если предмет закончился, а рюкзак не заполнен, берем следующий по цене, и так далее, пока не наберем максимально возможную суммарную массу предметов.

Для сортировки понадобится O(N·log(N)) операций.

В цикле по i от 1 до N и попробовать взять предмет i — займет еще O(n) операций. Итого, общая сложность — O(N·log(N) + N), что в общем случае эквивалентно O(N·log(N)).

Неоптимальность жадного алгоритма для дискретной задачи

Объем рюкзака= 50

60 у.е. в целом, 6 за единицу веса

100 у.е. в целом, 5 за единицу веса

120 у.е. в целом, 4 за единицу веса

Результат «жадного выбора» в дискретной задаче – 2 предмета – 10 и 20 кг, сумма – 160 у.е.

На самом деле оптимум – 2 и 3 предметы! Сумма – 220 у.е.

Размер задачи, временная сложность

Алгоритмическая временная сложность — это зависимость времени исполнения алгоритма от длины или количества входных данных, называемых **размером задачи**.

Измеряется время числом элементарных шагов. Каждый элементарный шаг связан с операцией, выполняемой в алгоритме.

Оценка временной сложности операторов программы

$T_{if} = T_v + \max(T_{then}+1, T_{else})$.

1 операция по ветке Then соответствует операции безусловного перехода.

$T_{for} = T_{v1} + T_{v2} + 1 + k(T_{тело} + 4)$,

где k – количество итераций цикла, 1 до цикла соответствует начальному присваиванию, 4 операции в цикле – сравнение параметра цикла с выражением V_2, условный переход на тело цикла, увеличение счетчика цикла и безусловный переход на начало цикла.

ПРИМЕР:

```
for i:=1 to 100 do
    for j:=1 to 3 do s:=s+a[i,j];
```

Его временная сложность равна 1+100(4+1+3*(4+2))=2301 операций.

Класс и порядок сложности

Определение. Множество вычислительных проблем, для которых существуют алгоритмы, схожие по сложности, называется **классом сложности.**

При оценке эффективности алгоритма вычисление точного значения функции временной сложности f(n) может быть трудно, поэтому используются методы аппроксимации для определения верхней границы функции.

Пусть имеется функция g(n) и константа K такая, что K*g(n) превышает f(n) по мере того, как n значительно возрастает.

Определение. Функция f(n) имеет порядок O(g(n)), если имеется константа K и счетчик n_0, такие, что

$f(n) < K*g(n)$, для $n > n_0$.

Порядок сложности нерекурсивных и рекурсивных функций

```
s=0;
for i:=1 to n do
  for j:=1 to m do s=s+1;
```

Временная сложность O(n*m)

Временная сложность для рекурсивных программ также является рекурсивной функцией:

$T(n_0)$=**const** - нет рекурсивного хода

$T(n)=f(T(g(n))$ – при рекурсивном вызове.

Грубая оценка временной сложности рекурсивной программы

Если верны соотношения

$T(1)=d;$

$T(n)=a*T(n/c)+b*n,$

то в зависимости от констант a и c

выражение для сложности имеет вид:

$$T(n)=\begin{cases} O(n) & \text{при } a<c \\ O(n \log_2 n) & \text{при } a=c \\ O(n^{\log_c a}) & \text{при } a>c \end{cases}$$

Классы сложности алгоритмов

O(1) - количество шагов алгоритма не зависит от количества входных данных. Обычно это алгоритмы, использующие определённую часть данных и игнорирующие все остальные данные. Например, анализ одного элемента массива вне зависимости от количества элементов;

$O(\log_2 n)$ – алгоритм двоичного поиска;

O(**n**) - алгоритм линейной сложности, когда для каждого входного объекта выполняется только одно действие;

$O(n \log_2 n)$ - такую сложность имеет алгоритм быстрой сортировки;

$O(n^2)$ - квадратичные алгоритмы. В основном, все простейшие алгоритмы сортировки.

$O(n^x)$ - полиномиальные алгоритмы.

$O(x^n)$ - экспоненциальные алгоритмы.

O(n!) - факториальные алгоритмы.

Временная сложность рекурсивного алгоритма для задачи «Ханойские башни»

Решение задачи для n дисков сводится к решению двух подзадач и одному ходу. Обе подзадачи имеют не половинный размер, а размер, лишь на единицу меньший исходного. Подсчитаем требуемое число ходов T(n). С учетом структуры решения:

$$T(n) = 2T(n-1) + T_{ход}$$

a=2, c=n/(n-1), следовательно a>c, и задача имеет экспоненциальный порядок сложности. По индукции можно доказать, что

$$T(n)=2^{n-1}+2^{n-2}+\ldots 2+1=2^{n}-1.$$

Зависимость сложности от n. P-задачи

n	$\log_2 n$	$N \log_2 n$	n^2	n^3	2^n
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	4294967296
128	7	896	16384	2097152	$3.4*10^{38}$
1024	10	10240	1048576	1073741824	$1.8*10^{308}$
65536	16	1048576	4294967296	$2.8*10^{14}$	Нам не дождаться!

Алгоритм, для которого $t(n)=O(n^x)$, где x – неотрицательная константа, называют **полиномиальными**

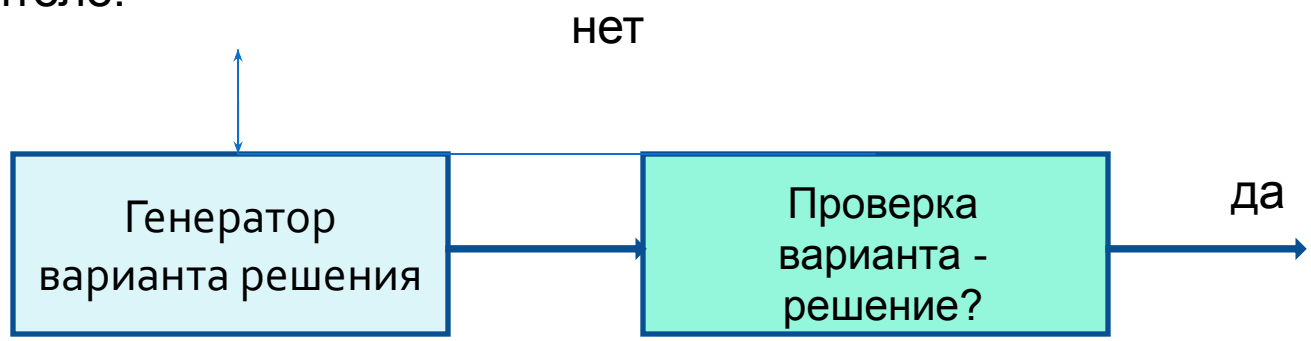
Определение. Задача принадлежит классу **P-задач**, если для нее существует детерминированный алгоритм, решающий ее за полиномиальное время.

Если для решения задачи не существует полиномиального алгоритма, будем называть ее **труднорешаемой**.

Экспоненциальные и NP-задачи

Экспоненциальные	«Ханойские башни» O (2^n)	Задача «Выполнимость»
P-задачи	Умножение матриц O (n^3)	

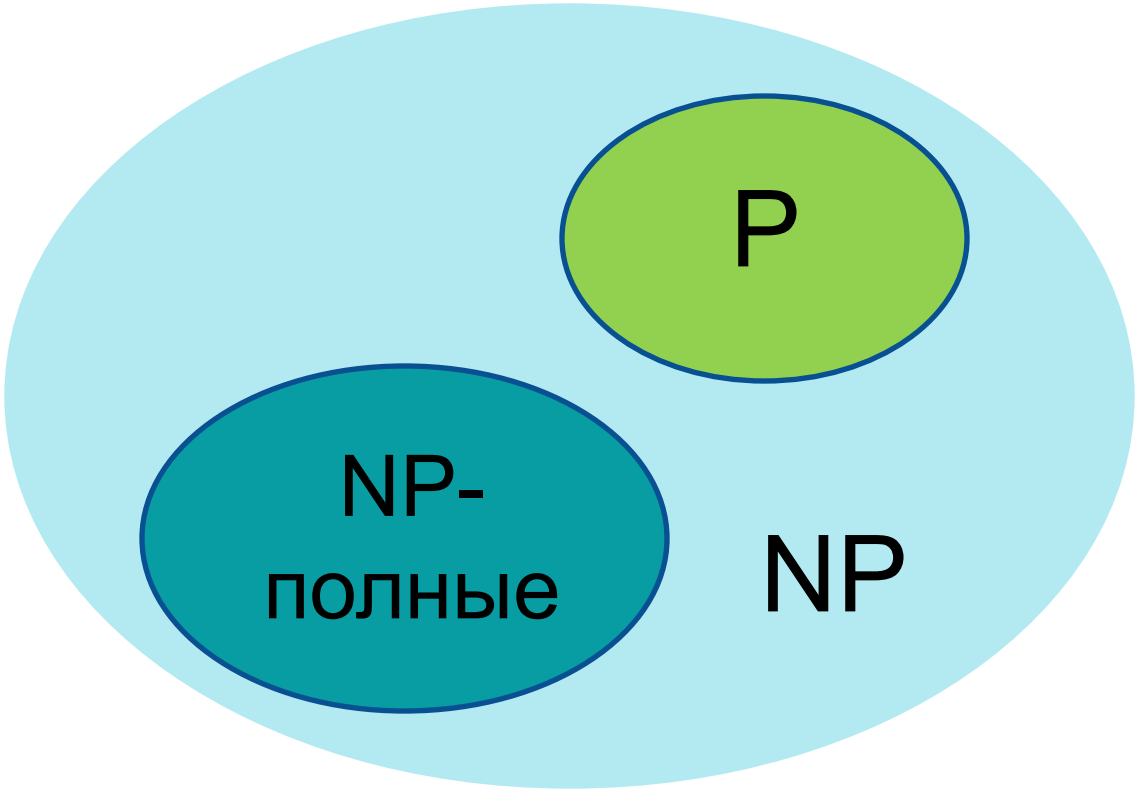
NP-задачи - это класс задач, которые можно решить за полиномиальное время (P), но на машине, более мощной, чем обычная однопроцессорная машина — на недетерминированном (N) вычислителе.



P ⊆ NP. Верно ли обратное (и тогда P = NP) или существуют задачи, решаемые за полиномиальное время на недетерминированной машине, но требующие экспоненциального времени на однопроцессорной машине (тогда P ⊂ NP, NP \ P≠∅) ?

NP-полные задачи

Основная теорема. Если некоторая NP – полная задача разрешима за полиномиальное время, то P = NP. Иначе говоря, если в классе NP есть задача, не разрешимая за полиномиальное время, то все NP – полные задачи таковы.



Задача «Выполнимость».Примеры NP-полных задач

ПРОБЛЕМА ВЫПОЛНИМОСТИ или задача о дизъюнкциях.

Дано конечное множество переменных $X_1,\ldots,X_N$. Дан набор дизъюнкций, составленных из этих переменных $D_1,\ldots,D_m$. Существует ли для $X_1,\ldots,X_N$ набор значений истинности, при котором выполняются (истинны) все дизъюнкции из множества D?

Примеры NP-полных задач:

1. Гамильтонов цикл.

Дан граф G с n вершинами. Существует ли в графе простой цикл, проходящий через все вершины графа? Простым называется цикл, в котором вершины не повторяются. Таким образом, гамильтонов цикл — это последовательность вершин и дуг (ребер) графа, содержащая все вершины графа G по одному разу, но, может быть, содержащая не все дуги.

2. Задача коммивояжера.

Дан граф G с n вершинами. Каждому ребру графа приписано положительное целое число di , задающее длину ребра. Кроме этого, задано некоторое положительное целое число L. Требуется ответить на вопрос: найдется ли в графе G маршрут, проходящий через все вершины графа G , такой, что его длина не превышает L?

Игры : пятнашки, тетрис, сапер.