Лекция 10. Эффективные алгоритмы

Методы решения *NP*-полных задач

- В соответствии с представлением алгоритма решения *NP*–полных задач с помощью алгоритма угадывания и алгоритма проверки программы, реализующие *NP*–полные задачи, требуют полного перебора вариантов и решаются рекурсивно, так, что алгоритм поиска решения на каждом их шаге рассматривает все возможные варианты решений на глубину 1 и оставшуюся задачу меньшего размера.
-

Пример

Пусть имеется произвольное клеточное поле и плитки размером 1×2. Необходимо покрыть данное поле такими плитками. Очевидно, что произвольное клеточное поле можно представить матрицей, в которой клетки заданного поля имеют следующие значения:

а) 0 — свободны,
б) число $n > 0$ — клетка занята плиткой с номером n,
в) число -1 — не принадлежащие полю клетки.

Пример

Сначала все свободные клетки заняты нулями, а все клетки, не подлежащие заполнению, числом -1. Приведем пример подлежащего заполнению поля:

0	0	-1	-1	-1
-1	0	-1	-1	-1
-1	0	0	0	0
-1	-1	0	0	0

1	1	-1	-1	-1	-1
-1	2	-1	-1	-1	-1
-1	2	3	4	4	-1
-1	-1	3	5	5	-1
-1	-1	-1	-1	-1	-1

Если полный перебор укладки плиток не привел к решению, следовательно задача решения не имеет.

Типы рекурсивных алгоритмов

Обычно рекурсивный алгоритм целесообразно разрабатывать при наличии одного из следующих условий:

1. При необходимости обработки данных, имеющих рекурсивную структуру.

2. Если алгоритм, обрабатывающий набор некоторых данных, можно построить, разбивая эти данные на части и получая из этих частичных решений решение на всей совокупности данных. Данный прием получил название "разделяй и властвуй".

Типы рекурсивных алгоритмов

3. Если задача поставлена так, что ее решением является выбор какого-то варианта из некоторого множества возможных решений. Решение задачи определяется после некоторого конечного числа шагов так, что выбирая на каждом шаге вариант решения, мы удаляем часть информации из всей подлежащей обработке информации и пытаемся решить задачу на меньшем объеме данных. Поиск решения завершается в двух случаях: либо когда кончатся данные, либо когда находится решение на текущем наборе данных. В частности, таким методом обычно решаются *NP*-полные задачи.

Типы рекурсивных алгоритмов

4. Если имеется рекурсивная схема некоторой функции. Существуют некоторые функции, которые легко можно определить рекурсивно, но которые нельзя определить в терминах обычных алгебраических выражений. Примером такой функции является функция Аккермана.

Рекурсия

- **прямая** (функция содержит вызов самой себя);
- **косвенная** (функция f1 вызывает функцию f2, которая вызывает исходную f1)

Рекурсия

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.

Пример. Вычисление факториала

Формула факториала:

$n! = n(n-1)(n-2)...1$

управляющий параметр - текущее значение числа n.

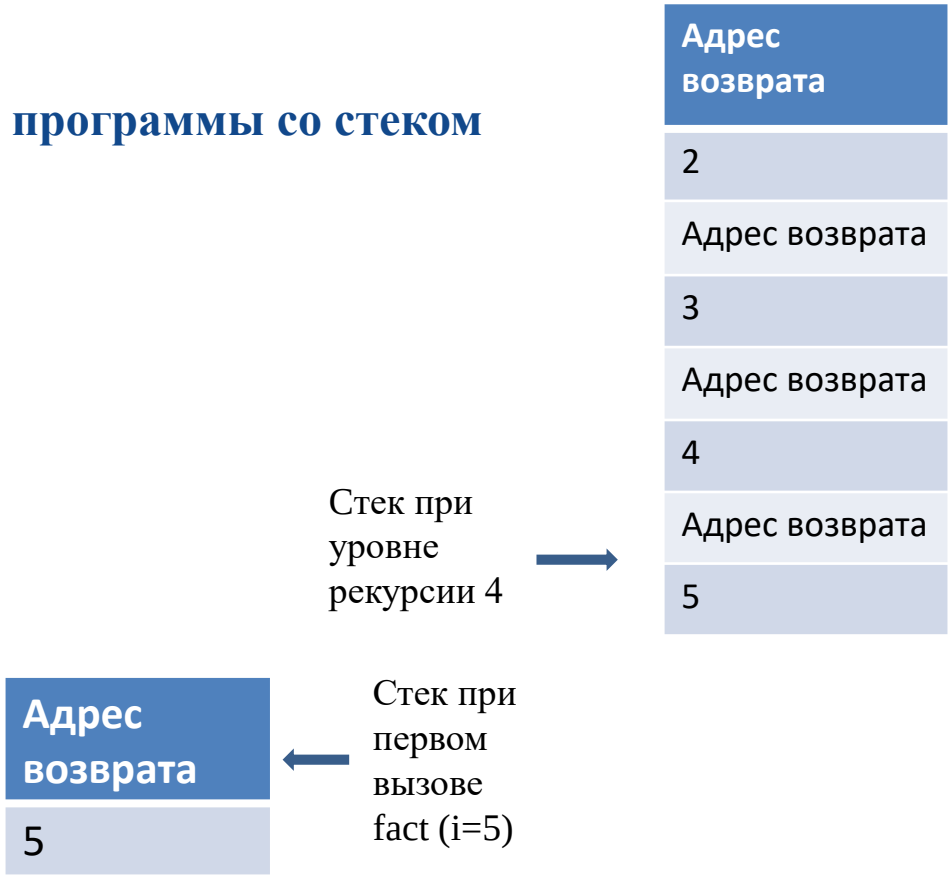
Тривиальный случай представляет собой $0! = 1$

декомпозиция общего случая $n! = n(n-1)!$

```
long int fact(int i)
{
if (i==0) return 1;
else return i*fact(i-1);
}
```

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*6=24	fact(4)=4*6=24
3	i=3 fact(3)=3*2=6	fact(3)=3*2=6
4	i=2 fact(2)=2*1*1=2	fact(2)=2*1*1=2
5	i=1 fact(1)=1*1=1	fact(1)=1*1=1
6	i=0 fact=1	



Пример. Поиск минимального

Дано множество *S*, содержащее *n* целых чисел (*n* > 2). Найти минимальный элемент в этом множестве.

Для простоты будем считать, что *n* есть степень числа 2. Применяя метод "разделяй и властвуй" разобьем множество *S* на два подмножества из *n*/2 элементов в каждом. Тогда достаточно найти минимальный элемент в каждом из полученных подмножеств и выбрать минимальное число из этих двух полученных:

```
int MinEl(Vector S, int i, int n)
// i - начальный элемент; n - число элементов
{
int ndiv2;
ndiv2=n/2;
if (n==2) then return min(S[i],S[i+1])
else return min( MinEl(S,i,ndiv2), MinEl(S,i+ndiv2,ndiv2) );
}
```

Этот метод деления заданного множества на две равные части широко применяется для сокращения числа попарных сравнений.

Пример. Ханойские башни

Легенда гласит, что в Великом храме города Бенарес, под собором, отмечающим середину мира, находится бронзовый диск, на котором укреплены 3 алмазных стержня, высотой в один локоть и толщиной с пчелу. Давным-давно, в самом начале времён, монахи этого монастыря провинились перед богом Брахмой. Разгневанный Брахма воздвиг три высоких стержня и на один из них возложил 64 диска, сделанных из чистого золота. Причём так, что каждый меньший диск лежит на большем. Как только все 64 диска будут переложены со стержня, на который Брахма сложил их при создании мира, на другой стержень, башня вместе с храмом обратятся в пыль и под громовые раскаты погибнет мир.

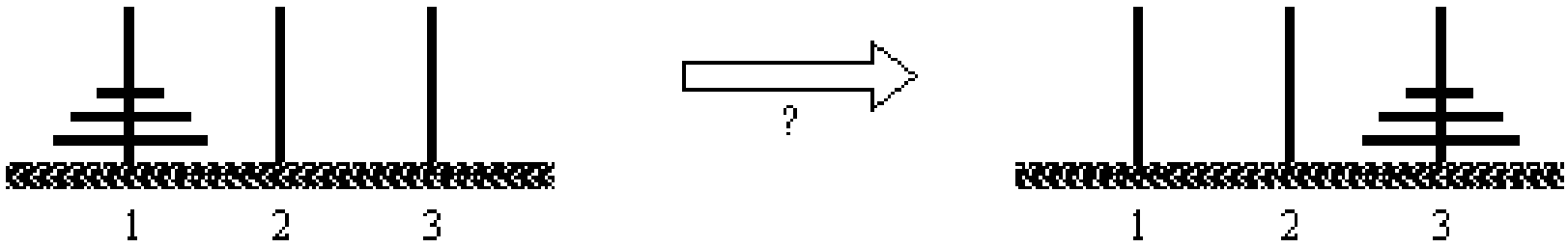
Пример. Ханойские башни

Количество перекладываний в зависимости от количества колец вычисляется по формуле $2^n - 1$

Число перемещений дисков, которые должны совершить монахи, равно

18 446 744 073 709 551 615.

Если бы монахи, работая день и ночь, делали каждую секунду одно перемещение диска, их работа продолжалась бы почти **585 миллиардов лет**.



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.

Алгоритм решения задачи Ханойские башни

- Обозначим тот стержень, с которого следует снять диски, через **si**, 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 четырьмя параметрами: **n, si, sw, sk**;
- алгоритм для $n = 1$ выделим в отдельную процедуру **step**, которая перемещает один диск со стержня **si** на **sk**.

```
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);
system("pause");
return 0;
}
```

Результат работы

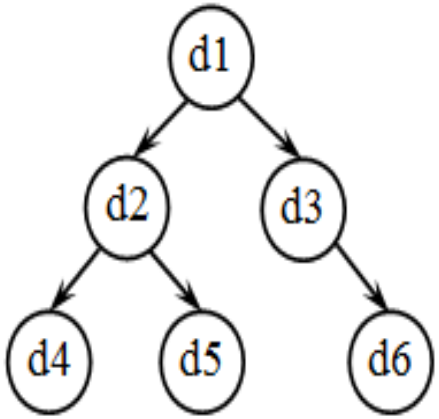
```
Input n:2

take disk from  left , put to middle
take disk from  left , put to right
take disk from  middle , put to right
Для продолжения нажмите любую клавишу . . .
```

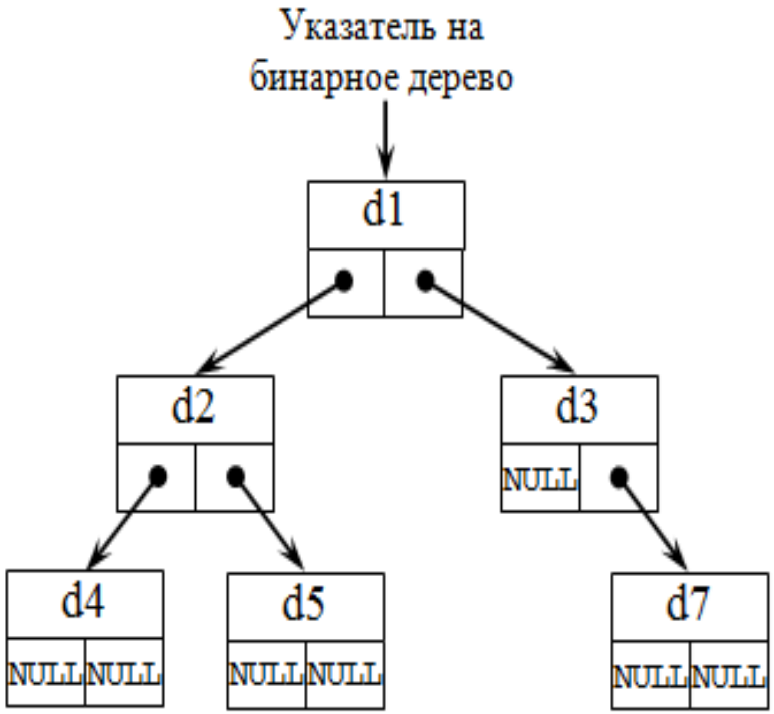
```
Input n:3

take disk from  left , put to right
take disk from  left , put to middle
take disk from  right , put to middle
take disk from  left , put to right
take disk from  middle , put to left
take disk from  middle , put to right
take disk from  left , put to right
Для продолжения нажмите любую клавишу . . .
```

Бинарные деревья

Дерево – это граф, имеющий следующую структуру:

- Начальный узел (вершина) дерева называется *корнем дерева*
- Каждая вершина имеет не более двух потомков.
- Вершины дерева соединены направленными дугами, которые называют *ветвями дерева*.
- *Листьями дерева* называют вершины, в которые входит одна ветвь и не выходит ни одной ветви.



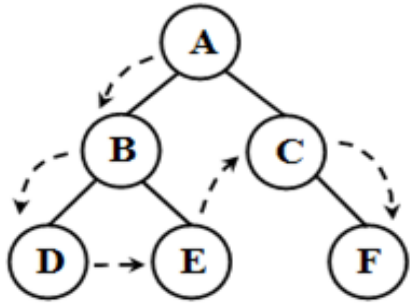


Рекурсивный алгоритм обхода бинарного дерева

- Прямой (корень – левое поддерево - правое поддерево);
- Обратный (левое поддерево – правое поддерево – корень) ;
- Симметричный (левое – корень – правое)

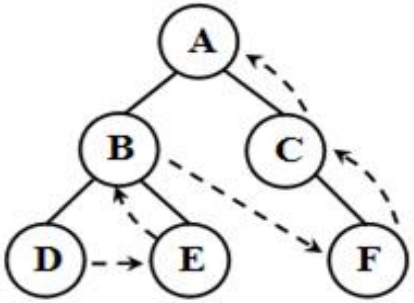
```
struct tree {
  char info;
  struct tree *left;
  struct tree *right;
};
```

Прямой



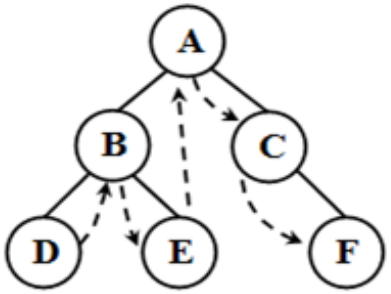
A B D E C F

Обратный



D E B F C A

Симметричный



D B E A C F

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

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

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

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

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

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

Методы отсечения

Самый прямолинейный подход при поиске решений методом полного перебора состоит в попытке перепробовать все различные ходы, пока не удастся получить решение.

Многие из прикладных задач имеют чрезвычайно большие пространства состояний, поэтому методы полного перебора всех вариантов практически неработоспособны из-за временных ограничений. Один из способов ускорения поиска решения состоит в использовании оценочных функций для упорядочивания перебора вариантов.

Методы отсечения

Оценочная функция должна обеспечивать возможность упорядочивания вершин — кандидатов на обработку — с тем, чтобы выделить ту вершину, которая с наибольшей вероятностью находится на лучшем пути к цели. Оценочные функции строятся на основе различных соображений и связаны с конкретной прикладной областью.

Например: «Крестики-нолики», «Пятнашки».

Игра «Восемь»

Выберем в качестве функции цели число позиций, на которых фишки стоят не на своих местах, и будем перебирать вершины в порядке неубывания функции цели.

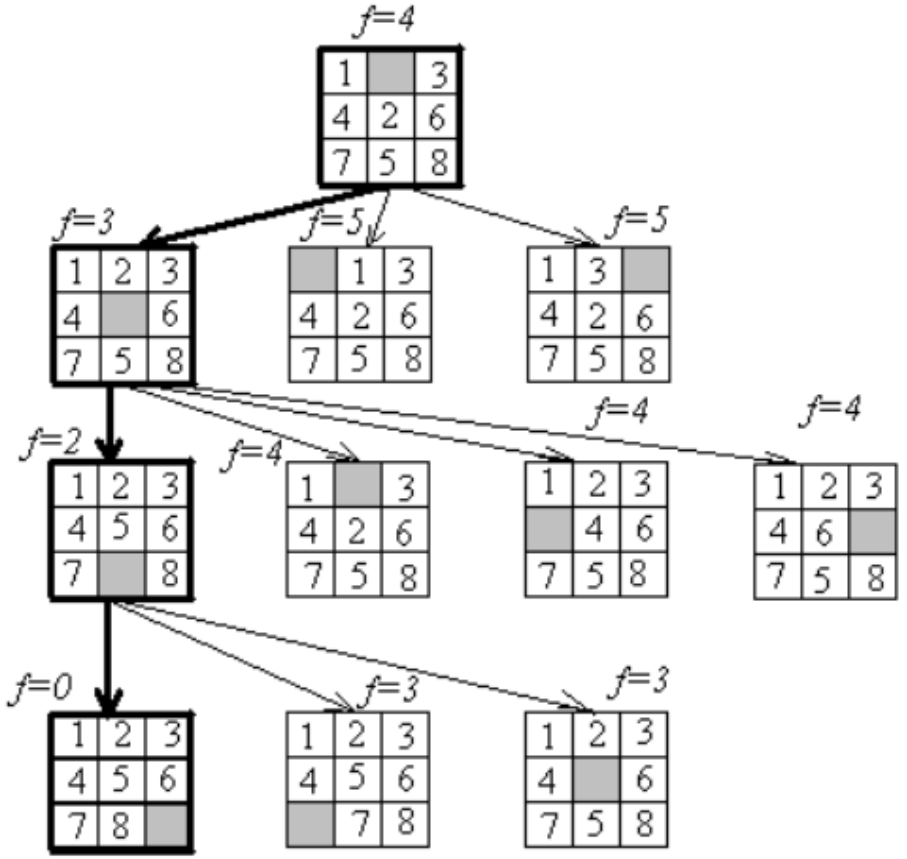


Рис. 7.5: Отсечение вариантов для игры в ВОСЕМЬ.

Устранение рекурсии

В общем случае к каждому алгоритму надо подходить индивидуально, моделируя те действия, которые выполняет рекурсивная программа, полученная в результате трансляции. Это означает, что в нерекурсивном эквиваленте рекурсивного алгоритма необходимо описать "*магазин*", каждый элемент которого содержит:

1. данные, являющиеся исходной информацией для рекурсивной процедуры;
2. внутренние (локальные) данные рекурсивной процедуры, если они есть.

Каждое обращение к рекурсивной процедуре в нерекурсивной программе соответствует занесению информации в магазин. Каждый выход из рекурсии соответствует стиранию информации из магазина. Общая структура нерекурсивной программы соответствует циклу типа «while», который выполняется при условии, что магазин не пуст.

Динамическое программирование

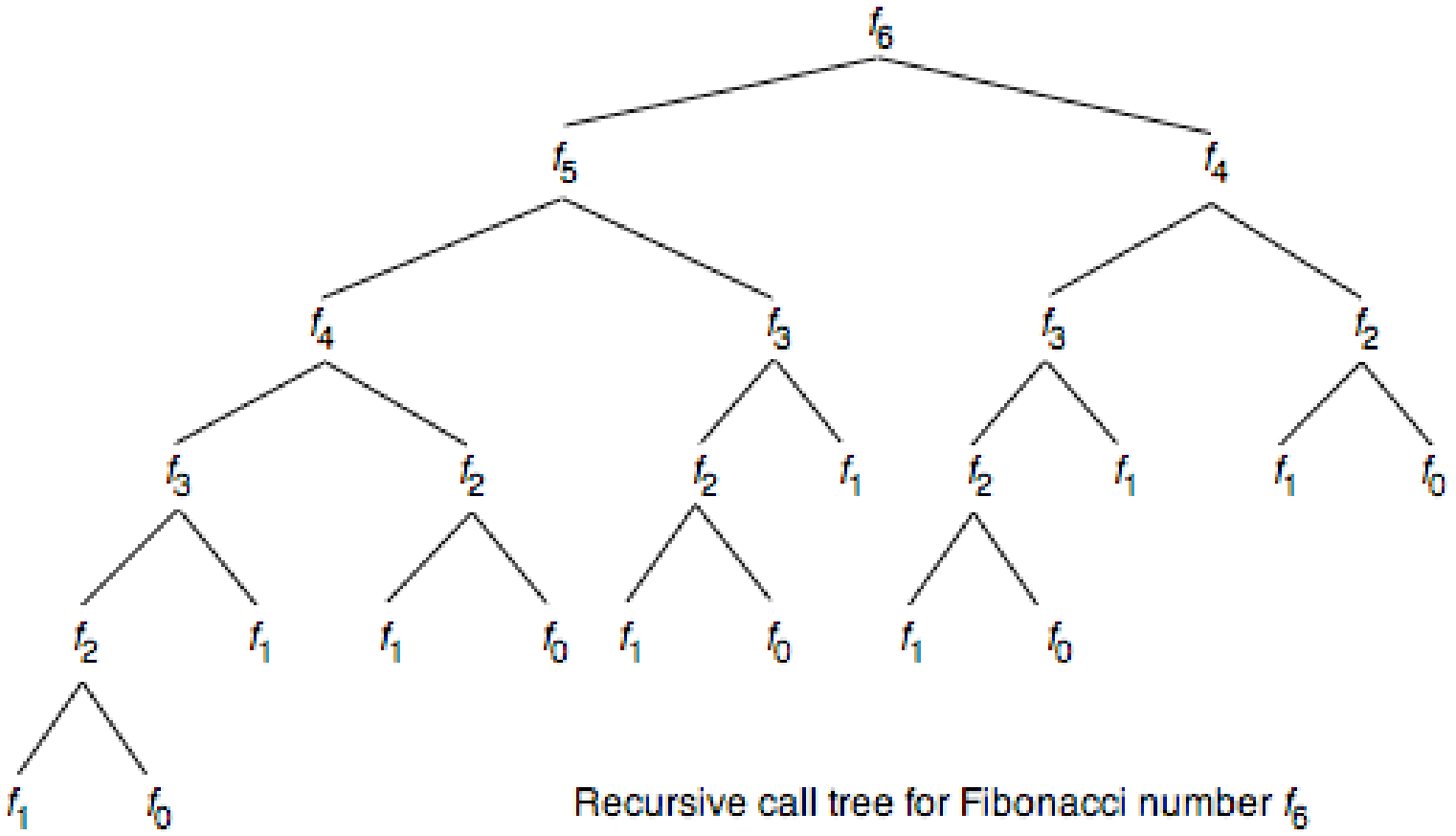
Рекурсивная техника полезна, если задачу можно разбить на подзадачи, каждая из которых решается за разумное время, т.е. суммарный размер задач будет небольшим.

Из формулы оценки временной сложности вытекает, что если сумма размеров подзадач задачи размера ***n*** равна ***an*** для некоторой постоянной $a > 1$, то рекурсивный алгоритм, вероятно, имеет полиномиальную временную сложность. Но если разбиение задачи размера n сводит ее к n задачам размера n-1, то рекурсивный алгоритм, вероятно, имеет экспоненциальную сложность. В этом случае часто можно получить более эффективные алгоритмы с помощью специальной техники, называемой динамическим программированием.

Суть динамического программирования основана на временном хранении в специальном массиве текущих решений на задачах предшествующих размеров.

Числа Фибоначчи

$$F_0 = 0, \qquad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2}, \quad n \geqslant 2, \quad n \in \mathbb{Z}.$$



Recursive call tree for Fibonacci number f_6

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.

Пример. Допустим, есть денежные купюры достоинством 10, 50, 100 и 500 рублей и нужно набрать 860 рублей.

Дискретная задача о рюкзаке

Постановка 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**.

Для решения задачи **жадным алгоритмом**, необходимо отсортировать вещи по их удельной ценности (то есть отношению ценности предмета к его весу), и поместить в рюкзак предметы с наибольшей удельной ценностью.

Дискретная задача о рюкзаке

10 кг, 60 у.е. в целом,
6 за единицу веса

20 кг, 100 у.е. в целом,
5 за единицу веса

30 кг, 120 у.е. в целом,
4 за единицу веса

Объем рюкзака= 50

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.

Относится к классу *NP*-полных, и для неё нет полиномиального алгоритма, решающего её за разумное время. Поэтому при решении необходимо выбирать между точными алгоритмами, которые неприменимы для «больших» рюкзаков, и приближенными, которые работают быстро, но не гарантируют оптимального решения задачи.

При выборе правильного метода решения задачи необходимо ответить на вопросы:

1. Насколько хорошо Вы понимаете проблему?
   - Что представляют собой входные данные? Что должен представлять собой результат?
   - Можно ли решить пример задачи на ограниченном малом объёме входных данных вручную?
   - Каков реальный типовой размер задачи на практике (N=?).
   - Какие условия и ограничения накладываются на время решения задачи?
   - Что приемлемо: прямое решение простым алгоритмом или разработка специального эффективного алгоритма в течение какого-то времени.
   - Определение класса к которому можно отнести задачу (комбинаторная, задача принятия решений, на графах и др.).
2. Можно ли построить простой алгоритм или эвристику?
   - Если ДА, то будет ли такой алгоритм корректным и какова его вычислительная сложность $T(N)$?

При выборе правильного метода решения задачи необходимо ответить на вопросы:

3. Существует ли типовой алгоритм решения задачи данного класса?

4. Существует ли какой-либо упрощенный вариант задачи, который можно решить сразу, если игнорировать некоторые входные параметры и ограничения задачи, допуская упрощенное представление форматов данных? Можно ли обобщить алгоритм решения упрощенного варианта задачи на всю задачу?

5. Какой из стандартных методов разработки эффективных алгоритмов наиболее соответствует задаче?

Наиболее часто используются следующие методы:

1. *«Разделяй и властвуй»* - метод декомпозиции, метод сведения задачи к подзадачам, метод частных целей.
2. *Динамическое программирование*.
3. *«Жадный» алгоритм*.
4. Метод отсечений (программирование *«с отходом назад»,* программирование с отслеживанием).
5. Метод *локального поиска* (*подъема)*.
6. Метод *эвристики*.
7. Метод *рекурсии*.