Динамическое программирование

Жадные алгоритмы

Код Хаффмана

- **Динамическое программирование (ДП)** - способ решения сложных задач путём разбиения их на более простые подзадачи.
- *Подход динамического программирования состоит* в том, чтобы решить каждую подзадачу только один раз, сократив тем самым количество вычислений.
- Термин «динамическое программирование» также встречается в теории управления в смысле «динамической оптимизации». Основным методом ДП является сведение общей задачи к ряду более простых экстремальных задач.

Лит.: Беллман Р., Динамическое программирование, пер. с англ., М., 1960.

- Ричард Беллман - американский математик, один из ведущих специалистов в области математики и вычислительной техники.

Пример. Числа Фибоначчи. $F_n = F_{n-1} + F_{n-2}$. $F_1 = 1, F_0 = 1$.

Можно вычислять рекурсивно:

$F_5 = F_4 + F_3$,

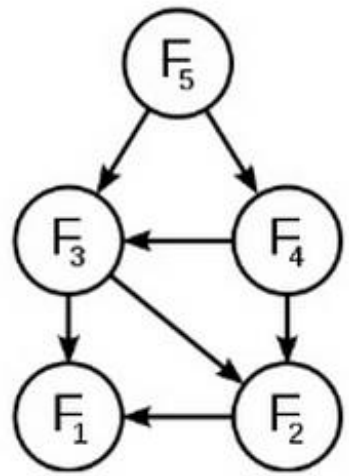
$F_4 = F_3 + F_2$,

$F_3 = F_2 + F_1$,

Многие значения могут вычисляться несколько раз.

Решение – сохранять результаты решения подзадач.

Этот подход называется **кэшированием**.

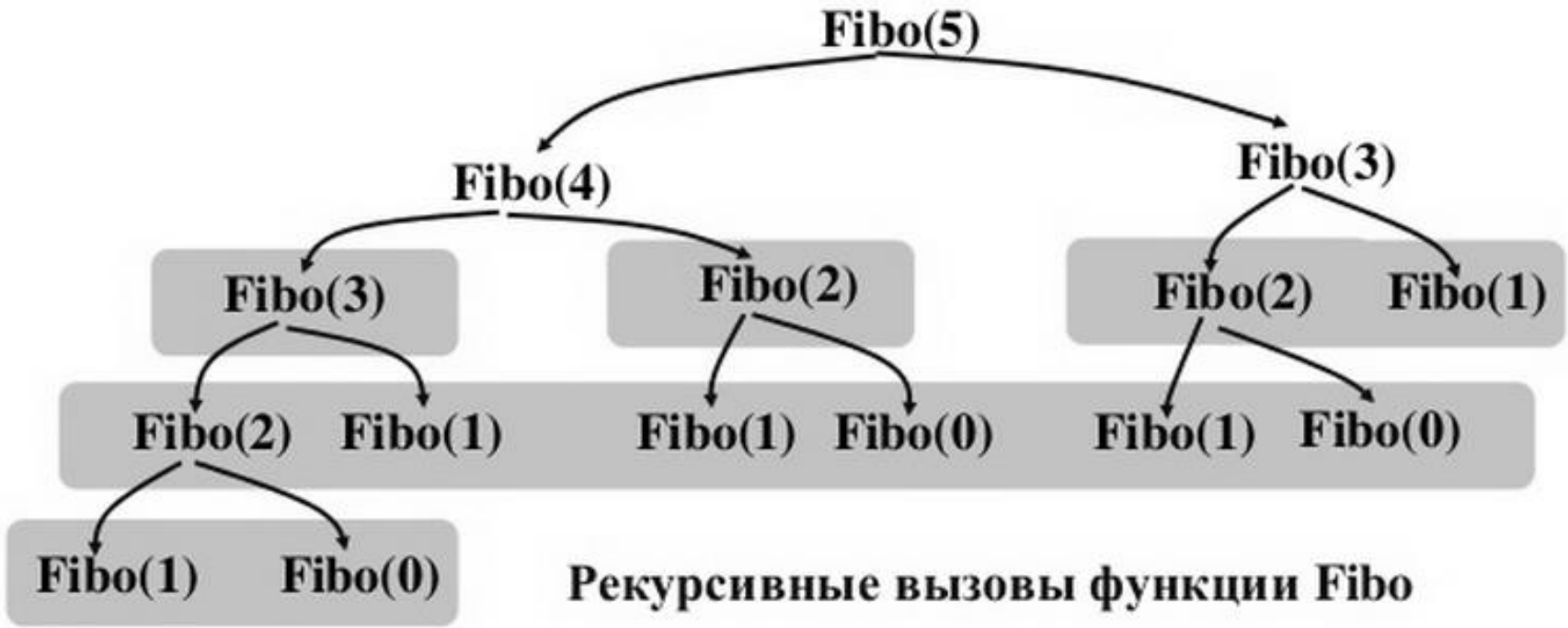


Последовательность Фибоначчи

- **0,1,1,2,3,5,8,13,21,34,…**
- $F(n) = F(n - 1) + F(n - 2)$, при $n > 1$ F(0) = 0, F(l) = 1
- Задача. Вычислить *n*-й член последовательности Фибоначчи.

```
function Fibo(n)
        if n <= 1 then
                return n
        end if
        return Fibo(n - 1) + Fibo(n -2)
end function
```

В динамическом программировании используются таблицы, в которых сохраняются решения подзадач (жертвуем памятью ради времени)



Рекурсивные вызовы функции Fibo

Пример. Вычисление рекуррентных функций нескольких аргументов.

$$F(x,y) = 3 \cdot F(x-1,y) - 2 \cdot F^2(x,y-1),$$
$$F(x,0) = x, F(0,y) = 0.$$

Вычисление $F(x,y)$ сводится к вычислению двух $F(\cdot,\cdot)$ от меньших аргументов.

Есть **перекрывающиеся подзадачи**.

$F(x-1,y-1)$ в рекурсивном решении вычисляется дважды.

$F(x-2,y-1)$ в рекурсивном решении вычисляется три раза.

$F(x-n,y-m)$ в рекурсивном решении вычисляется C_{n+m}^{n} раз.

Два подхода динамического программирования:

1. *Нисходящее динамическое программирование*, задача разбивается на подзадачи меньшего размера, они решаются и затем комбинируются для решения исходной задачи. Используется запоминание для решений часто встречающихся подзадач.

2. *Восходящее динамическое программирование*, все подзадачи, которые впоследствии понадобятся для решения исходной задачи просчитываются заранее и затем используются для построения решения исходной задачи.
Этот способ лучше нисходящего программирования в смысле размера необходимого стека и количества вызова функций, но иногда бывает нелегко заранее выяснить, решение каких подзадач нам потребуется в дальнейшем.

Принципы динамического программирования

1. Разбить задачу на подзадачи.
2. Кэшировать результаты решения подзадач.
3. Удалять более неиспользуемые результаты решения подзадач (опционально).

«Жадные» алгоритмы

- **«Жадный» алгоритм (Greedy algorithms)** - алгоритм, принимающий на каждом шаге локально-оптимальное решение, предполагая, что конечное решение окажется оптимальным.

Примеры «жадных» алгоритмов:

- алгоритм Хаффмана (кодирования)
- алгоритм Прима
- алгоритм Курскала
- алгоритм Дейкстры

Задача о размене (change-making problem)

Задача. Имеется неограниченное количество монет номиналом (достоинством) $a_1 < o_2 < \text{-…-} < a_п$.

Требуется выдать сумму *S* наименьшим количеством монет.

Жадный алгоритм решения этой задачи:

Берётся наибольшее возможное количество монет

достоинства a_n : $x_n = [S/a_n]$.

Также получаем, сколько нужно монет меньшего номинала.

Пример. Имеются монеты достоинством 1, 2, 5 и 10 рублей.

Выдать сумму 27 рублей.

Пример. Имеются монеты достоинством 1, 2, 5 и 10 рублей. Выдать сумму 27 рублей.

«Жадное» решение: 2 по 10, 1 по 5, 1 по 2.

На каждом шаге берётся наибольшее возможное количество монет достоинства a_n.

Жадные алгоритмы не всегда дают оптимальные решения.

Пример.

Имеются монеты достоинством 1, 5 и 7 рублей.

Выдать сумму 24 рубля.

Пример.

Имеются монеты достоинством 1, 5 и 7 рублей.

Выдать сумму 24 рубля.

- Решение жадным алгоритмом: 3 по 7, 3 по 1 = 6 монет

- Оптимальное решение: 2 по 7, 2 по 5 = 4 монеты

Жадный алгоритм

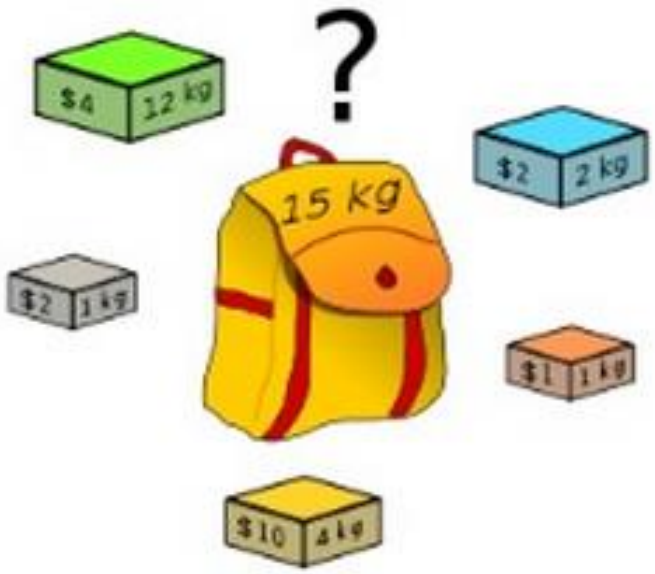
- заключается в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

Доказательство оптимальности часто следует такой схеме:

1. Доказывается, что жадный выбор на первом шаге не закрывает пути к оптимальному решению: для всякого решения есть другое, согласованное с жадным выбором и не хуже первого.
2. Показывается, что подзадача, возникающая после жадного выбора на первом шаге, аналогична исходной.
3. Рассуждение завершается по индукции.

Задача о рюкзаке

- **Задача о рюкзаке** (англ. Knapsack problem) — одна из задач комбинаторной оптимизации.
- Название своё получила от максимизационной задачи укладки как можно большего числа нужных вещей в рюкзак при условии, что общий объём (или вес) всех предметов, способных поместиться в рюкзак, ограничен.
- Имеется N грузов. Для каждого i-го груза определён вес *W i* и ценность *C i.*
- Нужно упаковать в рюкзак ограниченной грузоподъёмности G те грузы, при которых суммарная ценность упакованного была бы максимальной.



- Жадный алгоритм.
- Предметы сортируются по убыванию стоимости единицы каждого. Помещаем в рюкзак то, что помещается, и одновременно самое дорогое, т.е с максимальным отношением цены к весу.
- В оставшееся место помещаем следующий предмет аналогично.
- Точное решение можно получить не всегда.

Опр. В *непрерывной задаче о рюкзаке* (fractional knapsack problem), в отличие от общей задачи о рюкзаке, в рюкзак разрешать класть части предметов.

Замечания

- Нетрудно видеть, что жадный алгоритм находит оптимальное решение для непрерывной задачи о рюкзаке: на каждом шаге добавляем максимальное количество предмета максимальной удельной стоимости (стоимость/объём).
- Аналогичный алгоритм для общей задачи может и не найти оптимального решения: сразу положив в рюкзак самый дорогой предмет, мы можем потерять возможность полностью заполнить рюкзак.
- В первом случае выполняется принцип жадного выбора, во втором — нет.
- Поэтому непрерывная задача решается жадным алгоритмом, общая — динамическим программированием.

- Пример.

Пусть вместимость рюкзака 90. Предметы уже отсортированы. Применяем к ним жадный алгоритм.

	вес	цена	цена/вес
1	20	60	3
2	30	90	3
3	50	100	2

- Кладём в рюкзак первый, а за ним второй предметы. Третий предмет в рюкзак не влезет.
- Суммарная ценность поместившегося равна 150. Если бы были взяты второй и третий предметы, то суммарная ценность составила бы 190.
- Видно, что жадный алгоритм не обеспечивает оптимального решения, поэтому относится к приближенным.

Отличительные особенности жадных алгоритмов:

- Принцип жадного выбора: последовательность локально оптимальных (жадных) выборов дает глобально оптимальное решение.
  Для установления этого свойства, как правило, нужно показать, что жадный выбор согласован с некоторым оптимальным решением и что после выбора получается аналогичная подзадача.

- Свойство оптимальности для подзадач: оптимальное решение для задачи содержит оптимальные решения для подзадач.

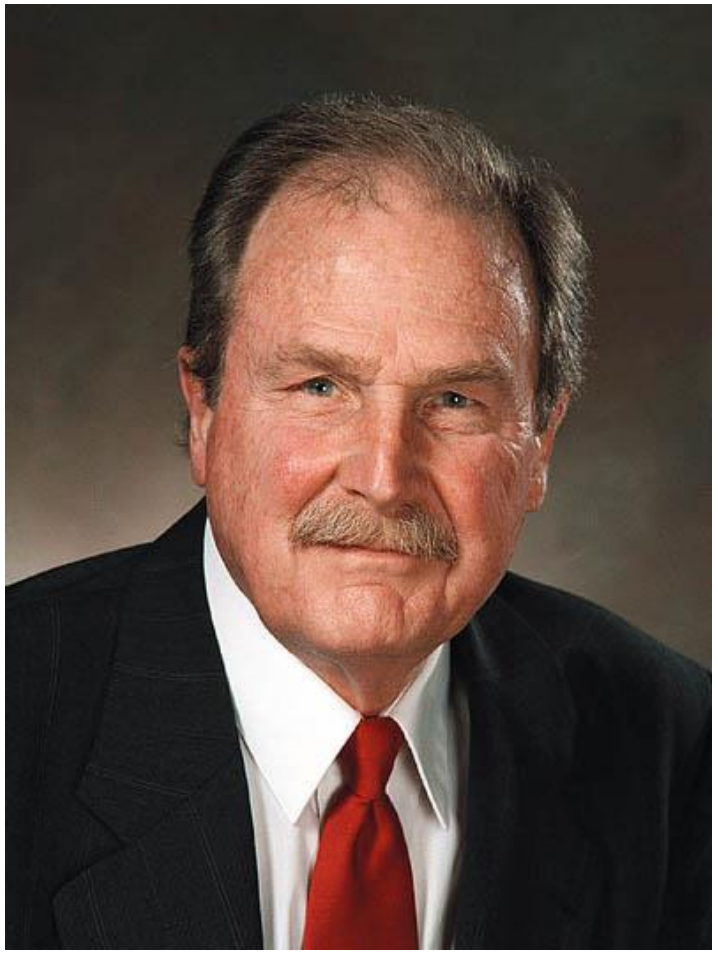
Код Хаффмана

- Деревья Хаффмана (Huffman) и коды Хаффмана используются для сжатая информации путем кодирования часто встречающихся символов короткими последовательностями битов.
- Предложен Д. А. Хаффманом в 1952 году (США, MIT).

Дэвид Хаффман

- первопроходец в сфере теории информации.
- Родился: 9 августа 1925 г., Огайо, США
- Умер: 7 октября 1999 г., Санта-Круз, Калифорния, США

- В 1952 году создал алгоритм префиксного кодирования с минимальной избыточностью. В 1999 году получил медаль Ричарда Хэмминга за исключительный вклад в теорию информации. Википедия
- Образование: Массачусетский технологический институт,Университет штата Огайо



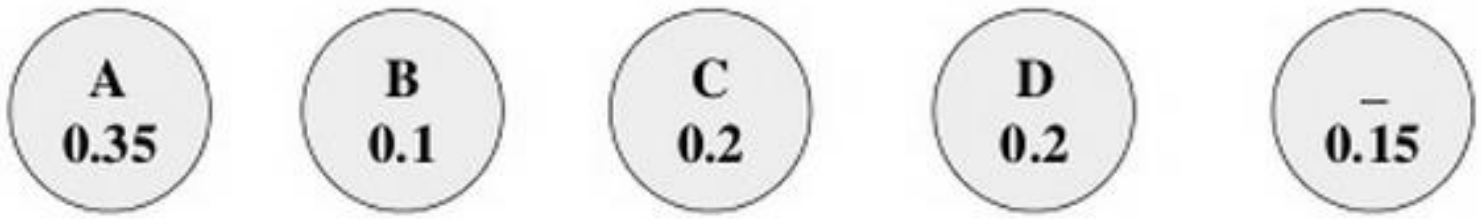
- Идея основана на частоте появления символа в последовательности.
- Символ, который встречается в последовательности чаще всего, получает новый очень маленький код, а символ, который встречается реже всего, получает, наоборот, очень длинный код.
- Таким образом, после обработки всего кода, самые частотные символы заняли меньше всего места (и меньше, чем они занимали в оригинале), а самые редкие - больше.

Задано множество символов и известны вероятности их появления в тексте (сообщении, файле).

- $A = \{a_1, a_2, \ldots, a_n\}$ множество символов (алфавит),
- $P = \{p_1, p_2, \ldots, p_n\}$ вероятности появления символов.
- **Требуется** каждому символу сопоставить код - битовую строку (последовательность, codeword).

$$C(A, P) = \{c_1, c_2, \ldots, c_n\}$$

- Пример



- Интуитивно ясно, что хочется придумать кодирование, при котором символ *A* кодировался бы одним битом (возможно, ценой того, что *B* бы кодировался тремя).
- Однако при кодировании символов последовательностями битов разной длины может возникнуть проблема декодировки.
- Одним из способов решения такой проблемы является префиксное кодирование.
- При таком кодировании ни для каких двух символов код одного не является префиксом кода другого.
- Каждое такое кодирование может быть представлено полным (у каждой вершины либо ноль, либо два сына) бинарным деревом, на ребрах которого стоят 0 и 1, а в листьях — кодируемые символы.

Оптимальное дерево

- Наша задача состоит в нахождении такого полного бинарного дерева, листья которого помечены символами и которое минимизирует *стоимость дерева*, определяемую как

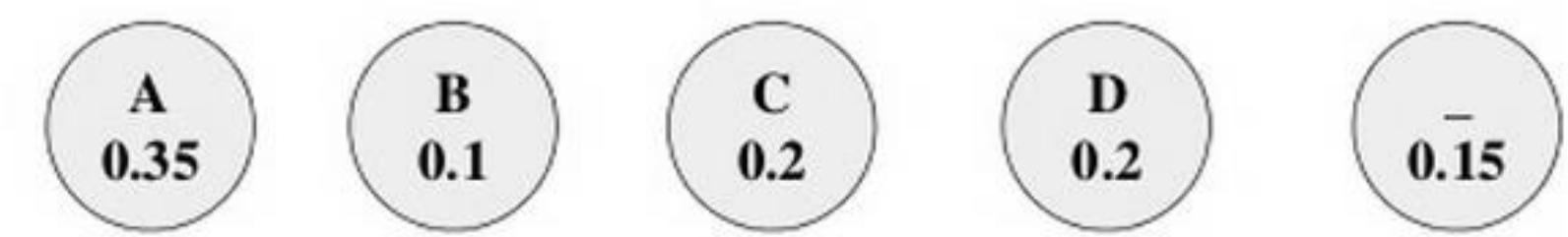
  Σfi *(*i=1,…n; глубина i-го символа в дереве).

- Определим частоту внутренней вершины дерева как сумму частот её сыновей. Нетрудно видеть, что при таком определении частота каждой вершины дерева равняется просто количеству посещений этой вершины при кодировании или декодировании.
- Стоимость дерева тогда равняется сумме частот всех вершин дерева кроме корня.

Идея жадного построения дерева

- Два самых редких символа должны висеть в самом низу оптимального дерева.
- Стоимость оптимального дерева для частот $f_1 .... f_n$, в котором f_1 и f_2 являются листьями-братьями, равняется сумме $(f_1 + f_2)$ и стоимости оптимального дерева для частот $(f_1 + f_2), ... f_n$.

Шаг 1. Создается *n* одноузловых деревьев.
В каждом узле записан символ алфавита и вероятность его появления в тексте.



- **Шаг** 2. Находим два дерева с наименьшими вероятностями и делаем их левым и правым поддеревьями нового дерева - создаем родительский узел.
- В созданном узле записываем сумму вероятностей поддеревьев.
- Повторяем шаг 2 пока не получим одно дерево.

На каждом шаге осуществляется "жадный выбор“

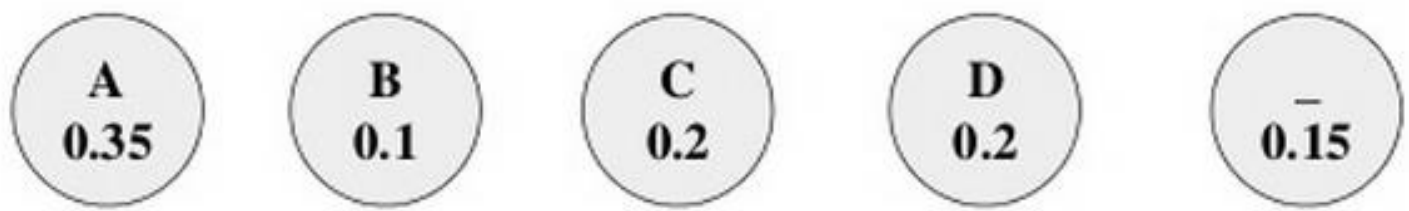
(два дерева с наименьшими вероятностями)

Алгоритм

```
HUFFMAN(f)
 1  ▷ Вход: массив частот f[1..n]
 2  ▷ Выход: кодирующее дерево с n листами
 3  создать очередь с приоритетами H, упорядоченную по f
 4  for i ← 1 to n
 5        do Insert(H, i)
 6  for k ← n + 1 to 2n − 1
 7        do i ← Extract-Min(H)
 8           j ← Extract-Min(H)
 9           создать вершину с номером k и сыновьями i, j
10           f[k] ← f[i] + f[j]
11           Insert(H, k)
```

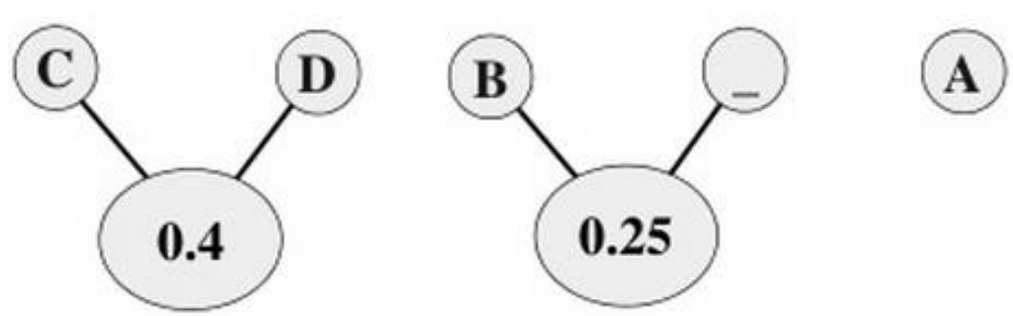
Пример

Символ	A	B	C	D	_
Код (биты)	11	100	00	01	101

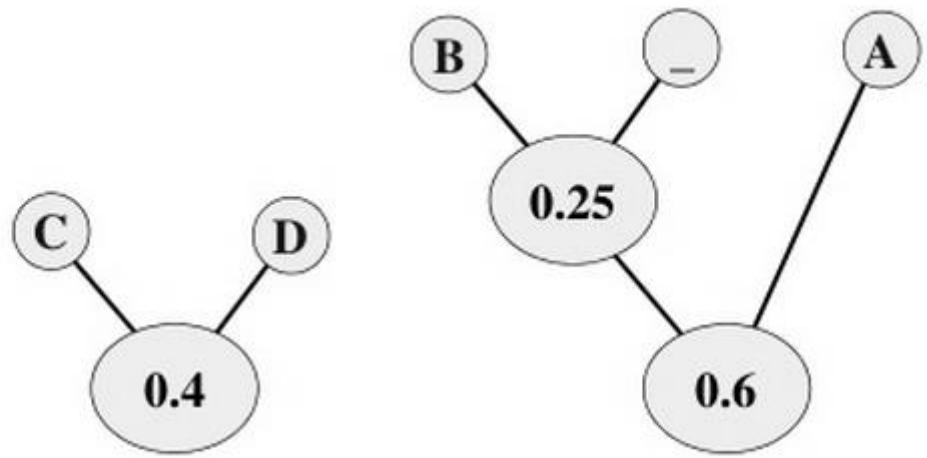


Символ	B	_	C	D	A
Вероятность	0.1	0.15	0.2	0.2	0.35

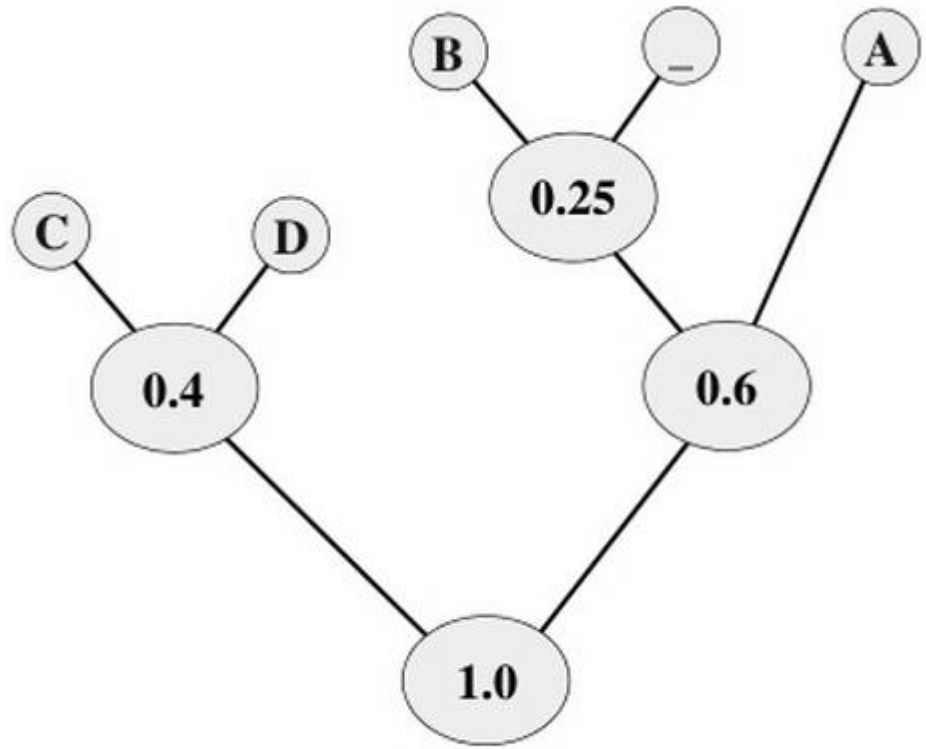
Символ	B	_	C	D	A
Вероятность	0.25		0.2	0.2	0.35



Символ	C	D	B, _	A
Вероятность	0.4		0.25	0.35

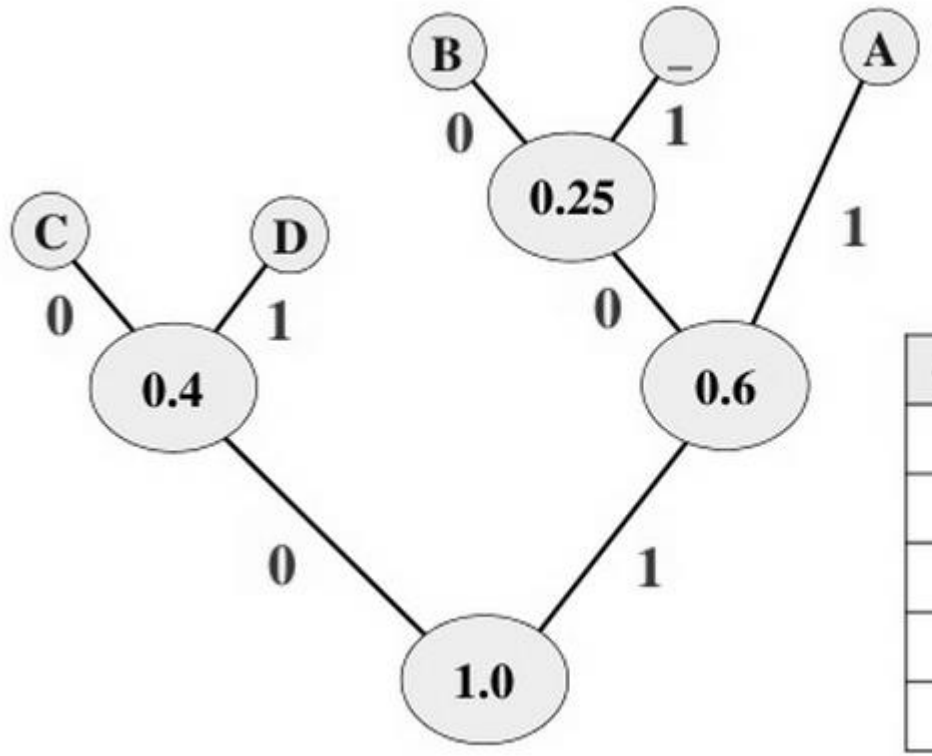


Символ	C	D	B, _	A
Вероятность	0.4		0.6	



Дерево Хаффмана

Теперь, чтобы получить код для каждого символа, надо пройти по дереву, и для каждого перехода добавлять 0, если идём влево, и 1 — если направо:



Символ	Код
A	11
B	100
C	00
D	01
_	101

Задание

Оценить трудоемкость построения дерева Хаффмана.

- На практике, при реализации данного алгоритма сразу после построения дерева строится таблица Хаффмана.
- Данная таблица - это связный список или массив, который содержит каждый символ и его код, это делает кодирование более эффективным.
- Затратно каждый раз искать символ и одновременно вычислять его код, так как мы не знаем, где он находится, и придётся обходить всё дерево целиком.
- Как правило, для кодирования используется таблица Хаффмана, а для декодирования — дерево Хаффмана.

Пример

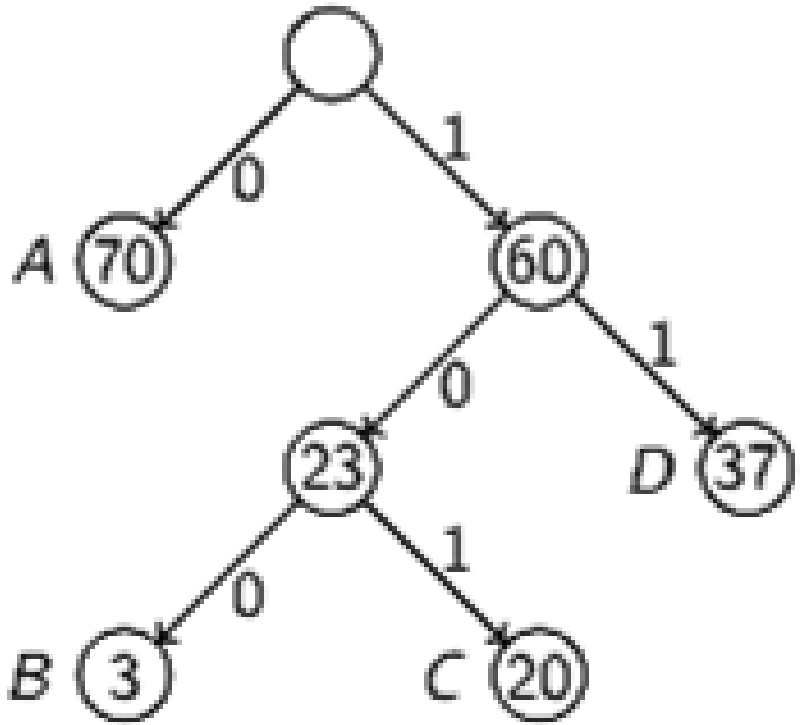
- Рассмотрим строчку длины 130, состоящую только

из символов *A, B, C, D*. Более того, допустим, что частота каждого символа известна:

A	*B*	*C*	*D*
70	3	20	37

- Наша задача состоит в том, чтобы каждому символу присвоить битовый код так, чтобы соответственно закодированная строчка имела как можно меньшую длину.

- Один из вариантов кодирования: *A* = 00, *B* = 01, C = 10, *D* = 1.
- В коде тогда будет 260 бит.



Кодирование

A	*B*	*C*	*D*
70	3	20	37
0	100	101	11

Закодированная строчка содержит 213 битов.

Спасибо за внимание!