

Жадность как подход к решению задач (greedy algorithms)

Принцип, примеры, реализация, ограничения

Задачка — 1368A — «C+=»

Лео создал новый язык программирования C+=. В C+= целочисленные переменные можно изменять только операцией «+=», которая прибавляет значение справа к переменной слева. Например, если выполнить « $a += b$ », когда $a = 2$, $b = 3$, значение a станет равно 5 (значение b при этом не изменится).

Лео создал программу-прототип с двумя целочисленными переменными a и b , исходно содержащими некоторые положительные значения. Он может выполнить некоторое количество операций « $a += b$ » или « $b += a$ ». Лео хочет протестировать обработку больших целых чисел, поэтому ему нужно сделать значение a либо b **строго больше**, чем некоторое данное число n . Какое наименьшее количество операций ему необходимо выполнить?

Задачка – 1368A – «C+=»

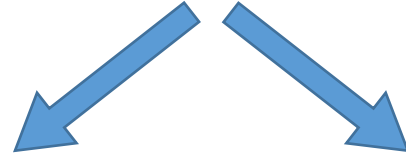
Лео создал новый язык программирования C+=. В C+= целочисленные переменные можно изменять только операцией «+=», которая прибавляет значение справа к переменной слева. Например, если выполнить «a += b», когда a = 2, b = 3, значение a станет равно 5 (значение b при этом не изменится).

Лео создал программу-прототип с двумя целочисленными переменными a и b, исходно содержащими некоторые положительные значения. Он может выполнить некоторое количество операций «a += b» или «b += a». Лео хочет протестировать обработку больших целых чисел, поэтому ему нужно сделать значение a либо b **строго больше**, чем некоторое данное число n. Какое наименьшее количество операций ему необходимо выполнить?

Перед нами типичная **задача оптимизации**, хоть и очень простая:

- Может быть много возможных решений
- Качество решения определяется некоторым параметром
- Требуется выбрать среди них одно оптимальное решение

Возможны всего 2 операции

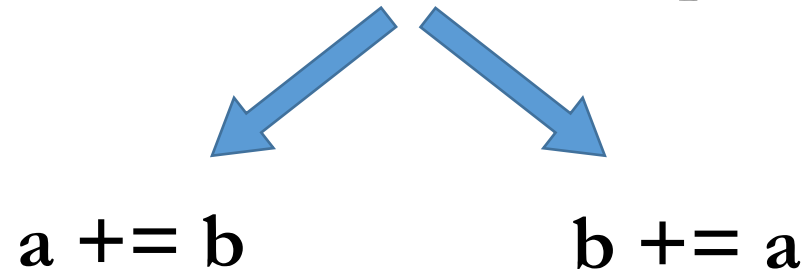


a += b

b += a

Какую из них нужно выбрать на каждом шаге, чтобы в итоге получить оптимальное решение?

Возможны всего 2 операции



Какую из них нужно выбрать на каждом шаге, чтобы в итоге получить оптимальное решение?

Ответ интуитивно понятен: увеличивать наименьшее число в паре.

Такая тактика имеет формальное название:

«Локально-оптимальный выбор»

Итак, на каждом шаге жадного алгоритма из всех возможных вариантов выбирается самый оптимальный на данном конкретном шаге.

При этом:

- Прошрое можем помнить, но откатиться не можем
- О будущем не задумываемся

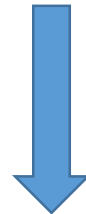
Итак, на каждом шаге жадного алгоритма из всех возможных вариантов выбирается самый оптимальный на данном конкретном шаге.

При этом:

- Прошрое можем помнить, но откатиться не можем
- О будущем не задумываемся

Принцип жадного выбора имеет место быть, когда:

Имеем последовательность локально-оптимальных выборов



Получаем глобально-оптимальное решение

Данная логика работает только при условии, что решаемая задача обладает **свойством оптимальности для подзадач**, т.е. оптимальное решение всей задачи обязательно содержит в себе оптимальные решения подзадач, из которых она состоит.

Данная логика работает только при условии, что решаемая задача обладает **свойством оптимальности для подзадач**, т.е. оптимальное решение всей задачи обязательно содержит в себе оптимальные решения подзадач, из которых она состоит.

Получаем два условия для использования жадного алгоритма:

- **Принцип жадного выбора**
- **Оптимальность для подзадач**

Данная логика работает только при условии, что решаемая задача обладает **свойством оптимальности для подзадач**, т.е. оптимальное решение всей задачи обязательно содержит в себе оптимальные решения подзадач, из которых она состоит.

Получаем два условия для использования жадного алгоритма:

- **Принцип жадного выбора**
 - **Оптимальность для подзадач**
-
- Выполняются ли данные условия в рассматриваемой задаче?
 - Правомерно ли решать эту задачу жадным алгоритмом?

```

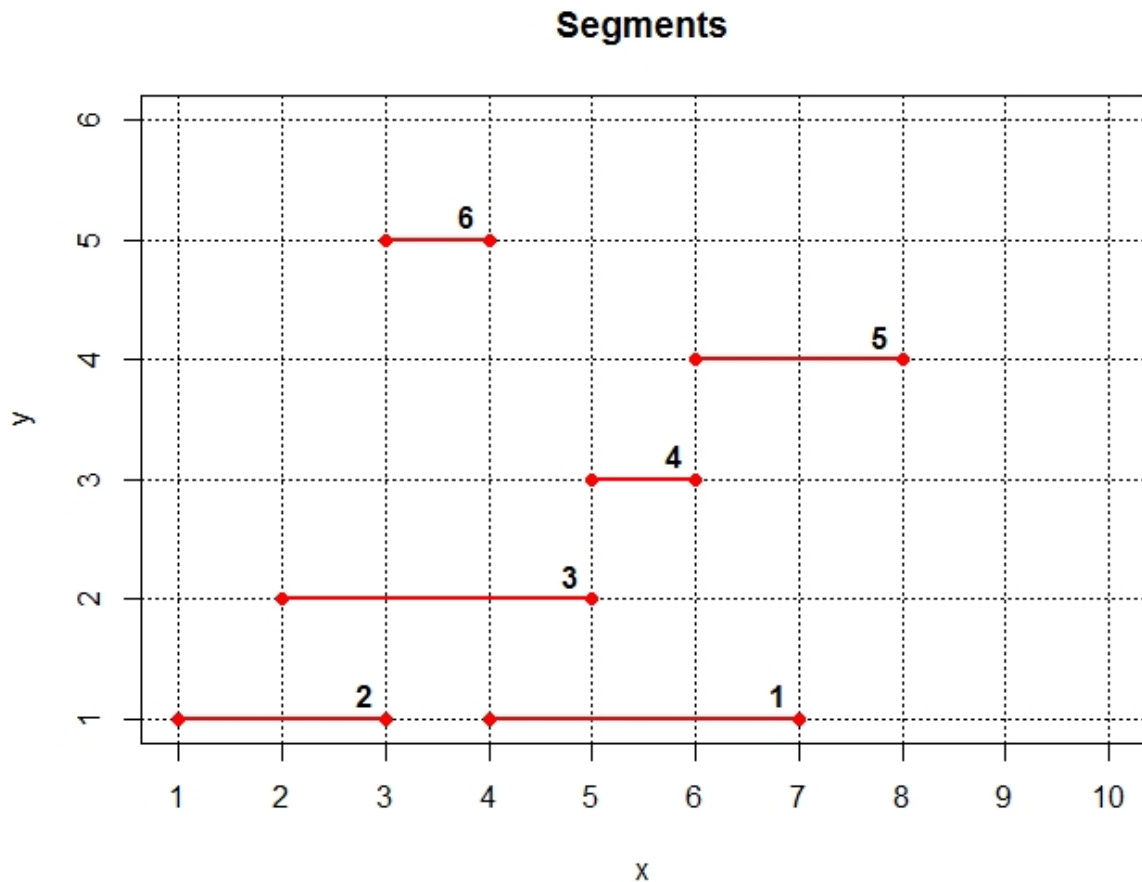
1  #include <iostream>
2  #include <vector>
3
4  using integer = long long unsigned int;
5
6  bool isNumbersOverBorder(integer a, integer b, integer border) {
7      if (a > border || b > border) {
8          return true;
9      }
10     return false;
11 }
12
13 int main() {
14     int tests;
15     std::cin >> tests;
16     for (auto i = 0; i < tests; i++) {
17         integer a, b, border;
18         std::cin >> a >> b >> border;
19
20         integer summ = 0;
21         int count = 0;
22         while (!isNumbersOverBorder(a, b, border)) {
23             if (a < b) {
24                 a += b;
25             } else {
26                 b += a;
27             }
28             count++;
29         }
30         std::cout << count << "\n";
31     }
32 }

```

**Что можно
улучшить?**

Жадная задача: покрытие отрезка точками

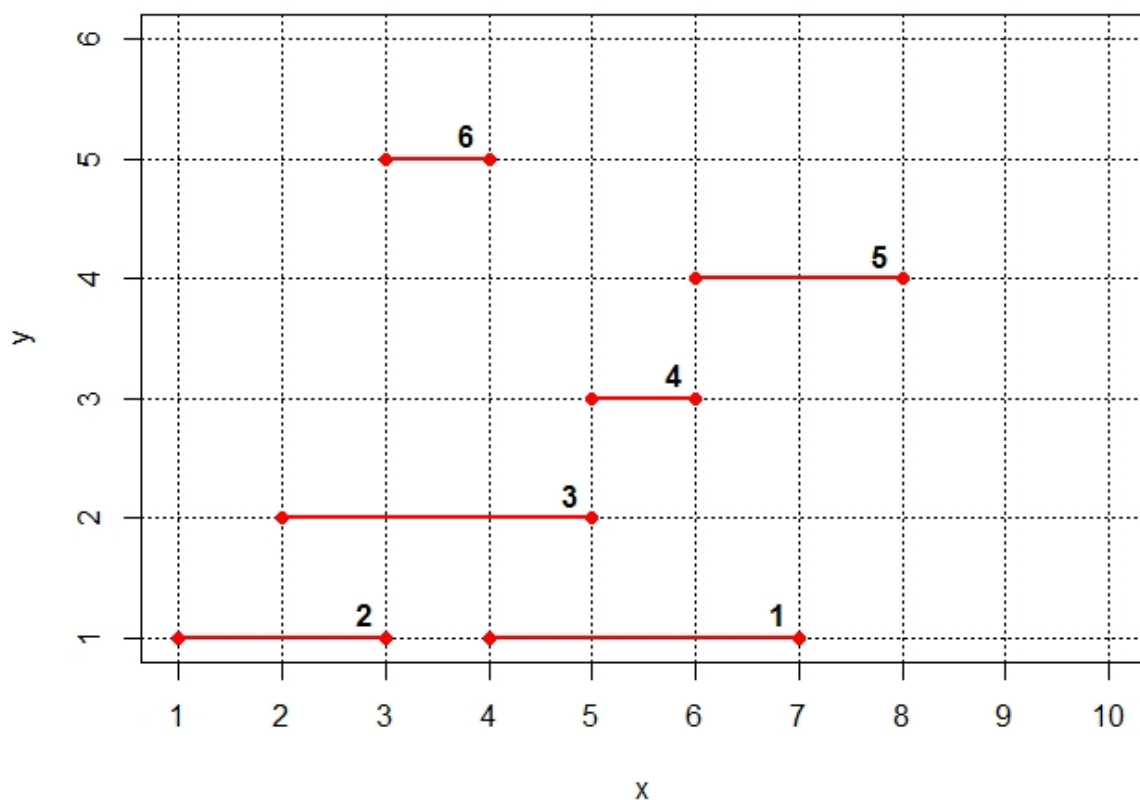
Условие: Дано n отрезков на прямой. Необходимо найти множество точек минимального размера, для которого каждый из отрезков содержит хотя бы одну из точек.



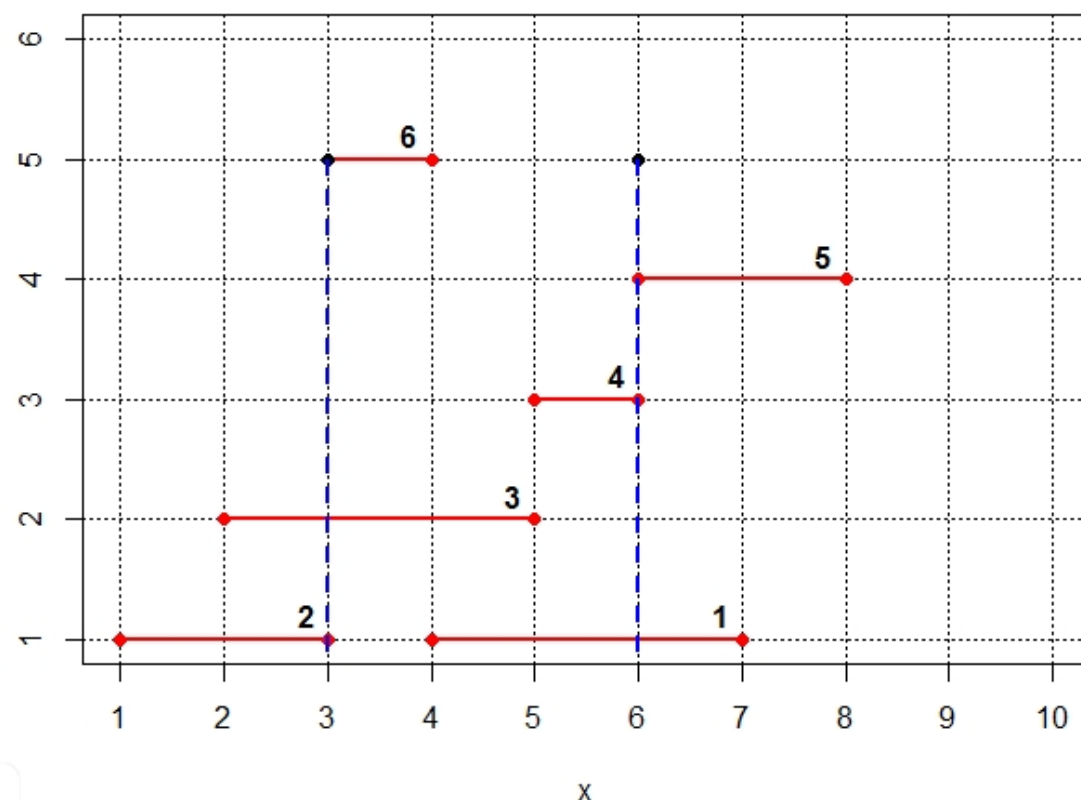
Жадная задача: покрытие отрезка точками

Условие: Дано n отрезков на прямой. Необходимо найти множество точек минимального размера, для которого каждый из отрезков содержит хотя бы одну из точек.

Segments



Segments



Алгоритм в общем виде:

Пока не покрыты все отрезки:

- находим минимальную правую границу

- добавляем эту точку-границу в ответ

- избавляемся от отрезков, в которые входит добавленная точка

Алгоритм в общем виде:

Пока не покрыты все отрезки:

находим минимальную правую границу

добавляем эту точку-границу в ответ

избавляемся от отрезков, в которые входит добавленная точка

Время работы: $O(n^2)$. Как оценим?

Алгоритм в общем виде:

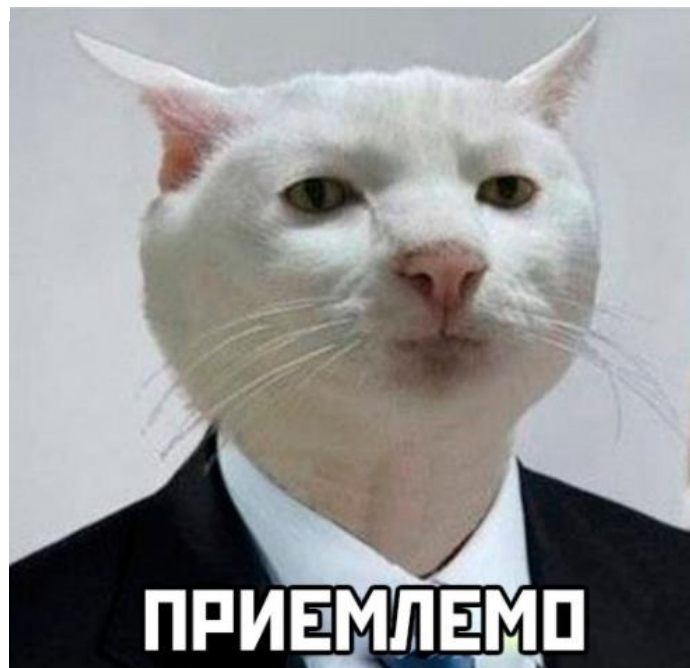
Пока не покрыты все отрезки:

находим минимальную правую границу

добавляем эту точку-границу в ответ

избавляемся от отрезков, в которые входит добавленная точка

Время работы: $O(n^2)$. Как оценим?



Алгоритм в общем виде:

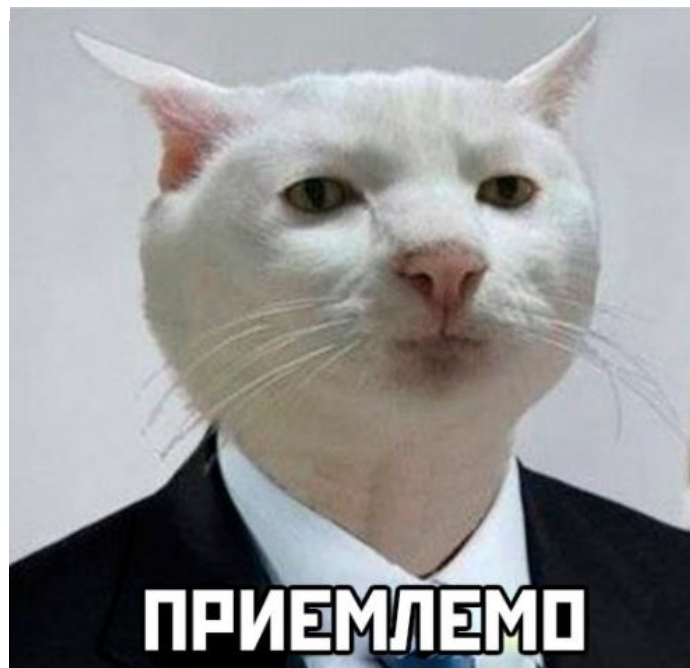
Пока не покрыты все отрезки:

находим минимальную правую границу

добавляем эту точку-границу в ответ

избавляемся от отрезков, в которые входит добавленная точка

Время работы: $O(n^2)$. Как оценим?



Но можно ли лучше?

Более эффективный алгоритм:

Более эффективный алгоритм:

Сортируем отрезки по правому краю

Пока не покрыты все отрезки:

находим минимальную правую границу

добавляем эту точку-границу в ответ

избавляемся от отрезков, в которые входит добавленная точка

Время работы: $O(n^2)$.

Более эффективный алгоритм:

Сортируем отрезки по правому краю

Пока не покрыты все отрезки:

 находим минимальную правую границу

 добавляем эту точку-границу в ответ

 избавляемся от отрезков, в которые входит добавленная точка

Время работы: $O(n^2)$. Но константы будут значительно лучше.

Более эффективный алгоритм:

Сортируем отрезки по правому краю

Пока не покрыты все отрезки:

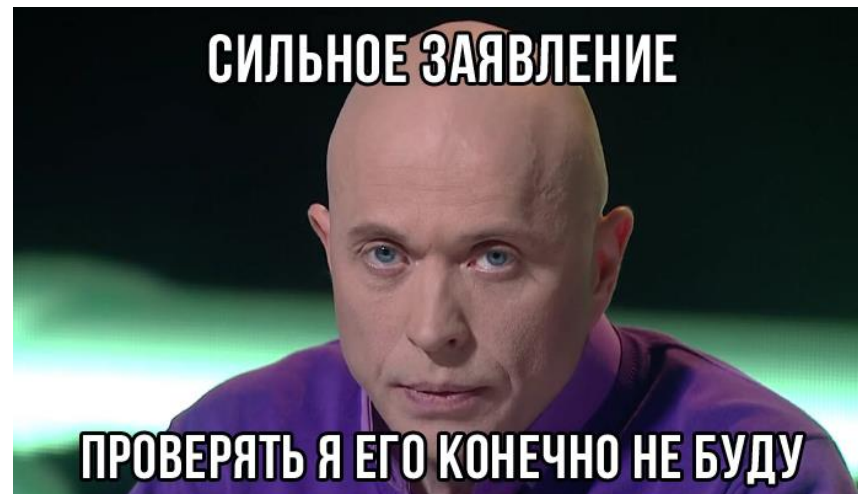
находим минимальную правую границу

добавляем эту точку-границу в ответ

избавляемся от отрезков, в которые входит добавленная точка

Время работы: $O(n^2)$. Но константы будут значительно лучше.

Также возможно добиться $O(n \log(n))$.



Наивная реализация. Ключевые моменты.

Для удобства используем псевдонимы

```
using integer = long long unsigned int;  
using Segment = std::pair<integer, integer>;
```

Наивная реализация. Ключевые моменты.

Для удобства используем псевдонимы

```
using integer = long long unsigned int;  
using Segment = std::pair<integer, integer>;
```

Чтение. Выделение буфера

```
std::vector<Segment> segments;  
segments.resize(n);  
for (auto i = 0; i < n; i++) {  
    std::cin >> segments[i].first >> segments[i].second;  
}
```

Наивная реализация. Ключевые моменты.

Для удобства используем псевдонимы

```
using integer = long long unsigned int;
using Segment = std::pair<integer, integer>;
```

Чтение. Выделение буфера

```
std::vector<Segment> segments;
segments.resize(n);
for (auto i = 0; i < n; i++) {
    std::cin >> segments[i].first >> segments[i].second;
}
```

Удаление отрезков с точкой внутри

```
std::vector<integer> points;
while (!segments.empty()) {
    auto newPoint = segments.front().second;
    points.push_back(newPoint);

    auto it = std::find_if(std::cbegin(segments), std::cend(segments),
        [point=newPoint](auto segment) {
            if (segment.first > point)
                return true;
            return false;
        });
    if (it != std::cend(segments)) {
        segments.erase(std::cbegin(segments), it);
    } else {
        segments.clear();
    }
};
```


Наивная реализация. Ключевые моменты.

Для удобства используем псевдонимы

```
using integer = long long unsigned int;
using Segment = std::pair<integer, integer>;
```

Чтение. Выделение буфера

```
std::vector<Segment> segments;
segments.resize(n);
for (auto i = 0; i < n; i++) {
    std::cin >> segments[i].first >> segments[i].second;
}
```

Удаление отрезков с точкой внутри

```
std::vector<integer> points;
while (!segments.empty()) {
    auto newPoint = segments.front().second;
    points.push_back(newPoint);

    auto it = std::find_if(std::cbegin(segments), std::cend(segments),
        [point=newPoint](auto segment) {
            if (segment.first > point)
                return true;
            return false;
        });
    if (it != std::cend(segments)) {
        segments.erase(std::cbegin(segments), it);
    } else {
        segments.clear();
    }
}
```

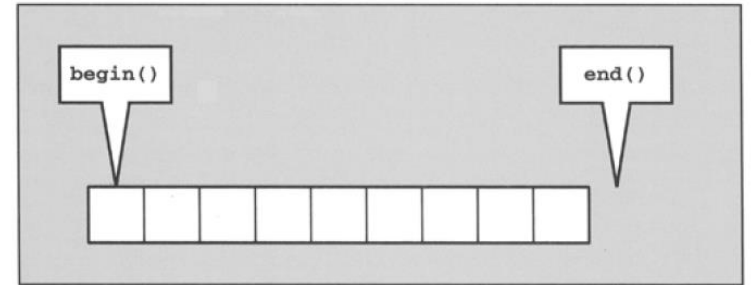
Сортировка отрезков по правому краю

```
std::sort(std::begin(segments), std::end(segments), [](auto lhs, auto rhs) {
    if (lhs.second < rhs.second)
        return true;
    return false;
});
```

Минутка C++

Рассмотрим типичного представителя алгоритма STL на примере `std::find_if`.

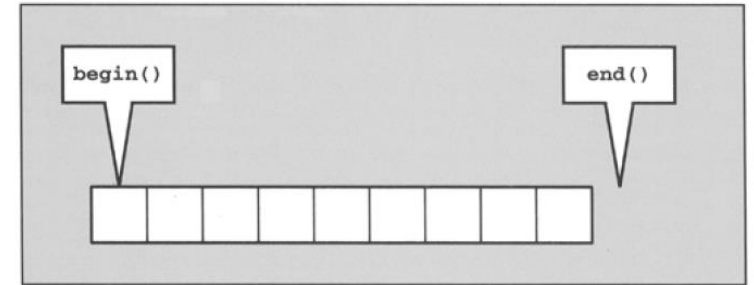
```
void remove_if(std::vector<Segment>::iterator it, Segment &segments, integer point) {  
    auto it = std::find_if(std::begin(segments), std::end(segments), [point](auto segment) {  
        if (segment.first > point)  
            return true;  
        return false;  
    });  
}
```



Минутка C++

Рассмотрим типичного представителя алгоритма STL на примере `std::find_if`.

```
void remove_if(std::vector<Segment>::iterator it, Segment &segments, integer point) {  
    auto it = std::find_if(std::begin(segments), std::end(segments), [point](auto segment) {  
        if (segment.first > point)  
            return true;  
        return false;  
    });  
}
```



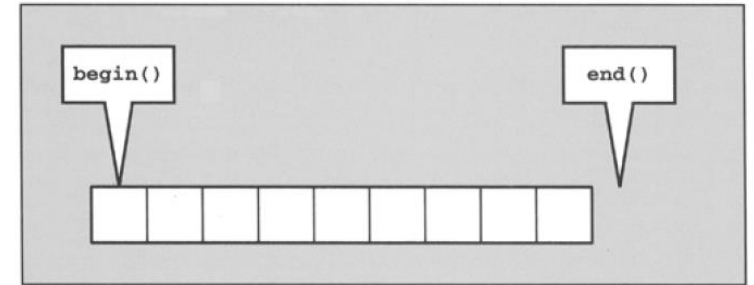
- Первый аргумент – итератор на начало.
- Второй аргумент – итератор на элемент, за последним обрабатываемым.
- Третий аргумент – предикат – некий callable объект. В нашем случае используем шаблонную лямбда-функцию (*generic lambda C++14*).
- Возвращает итератор на первый найденный элемент, для которого предикат вернул true.

```
bool cmp(const Type1 &a, const Type2 &b);
```

Минутка C++

Рассмотрим типичного представителя алгоритма STL на примере `std::find_if`.

```
void remove_if(std::vector<Segment>::iterator it, Segment &segments, integer point) {  
    auto it = std::find_if(std::begin(segments), std::end(segments), [point](auto segment) {  
        if (segment.first > point)  
            return true;  
        return false;  
    });  
    segments.erase(it, std::end(segments));  
}
```



- Первый аргумент – итератор на начало.
- Вторым аргументом – итератор на элемент, за последним обрабатываемым.
- Третий аргумент – предикат – некий callable объект. В нашем случае используем шаблонную лямбда-функцию (*generic lambda C++14*).
- Возвращает итератор на первый найденный элемент, для которого предикат вернул true.

```
if (it != std::cend(segments)) {  
    segments.erase(std::cbegin(segments), it);  
} else {  
    segments.clear();  
}
```

Если ничего найдено не было –
возвращается `end()`.

Приставка *c* означает *constant*, *r* – *reverse*.

Минутка философии

Если возникает интуитивно-стандартная задача поиска по критерию, удалению, изменению порядка, сортировки, модификации по правилу и тому подобного, то стараемся использовать **стандартное решение**.

- Уменьшается количество костылей (в чём мы более уверены: в алгоритме стандартной библиотеки или в самопальном велосипеде?)
- Вслед за этим уменьшается сложность и стоимость сопровождения (вообще само проектирование ПО и всякие методики программирования служат именно этой цели — уменьшение сложности)
- Заодно расширяем кругозор возможностей своих библиотек, не стесняемся гуглить (лучше по-английски)
- Есть варианты, когда библиотечной скорости не хватает, и нужно что-то побыстрее. Понимаем корень проблемы, локализуем её, обвешиваем тестами, делаем бенчмарки, одним словом — профилировка
- «Неважно, насколько код быстр, если он неправилен»

Форматированная реализация

Теперь функция main выглядит так:

```
19  int main() {
20      auto segments = readSegments(std::cin);
21      sortSegmentsByEnds(segments);
22
23      std::vector<integer> points;
24      while (!segments.empty()) {
25          auto newPoint = findGreedyNewPoint(segments);
26          points.push_back(newPoint);
27          removeSegmentWithPointInside(segments, newPoint);
28      };
29
30      printOptimalPointCovering(points);
31  }
```

Форматированная реализация

Теперь функция main выглядит так:

```
19  int main() {
20      auto segments = readSegments(std::cin);
21      sortSegmentsByEnds(segments);
22
23      std::vector<integer> points;
24      while (!segments.empty()) {
25          auto newPoint = findGreedyNewPoint(segments);
26          points.push_back(newPoint);
27          removeSegmentWithPointInside(segments, newPoint);
28      };
29
30      printOptimalPointCovering(points);
31  }
```

- При необходимости внесения изменений в реализацию нужно тратить меньше умственных усилий. Сложность спрятана за ширмой.
- Легко тестировать отдельные функции
- Код читается как проза. Алгоритм читается как псевдокод.

Пару слов о чтении данных



Поддерживайте модульность кода путем использования абстракций потока. Это позволит отвязать фрагменты исходного кода друг от друга и облегчит тестирование исходного кода, поскольку можно внедрить любой другой соответствующий тип потока.

`std::cin` и `std::ifstream` взаимозаменяемы. `cin` имеет тип `std::istream`, а `std::ifstream` наследует от `std::istream`.

```
30  std::vector<Segment> readSegments(std::istream &in) {
31      int n;
32      in >> n;
33      std::vector<Segment> segments;
34      segments.resize(n);
35
36      // псеводокод! Необходимо создать отдельный тип вместо п
37      std::istream_iterator<Segment> it{std::cin};
38      std::istream_iterator<Segment> end;
39      std::copy(it, end, std::back_inserter(segments));
40      return segments;
41  }
```

Также не забываем про возможность перенаправления потока в консоли:

```
C:\Users\Максим\Desktop\pia
λ .\a.exe < test.txt |
```

Есть ещё способ работать с потоками круче — но о нем попозже.

Другие типичные жадные задачи

Вообще задача о покрытии отрезков точками имеет более популярную вариацию – **задачу о покрытии точек отрезками**. Естественно, также минимальным их количеством.



Другая классика имеет название «**Задача о составлении расписания**».

Алгоритм Борувки – один из алгоритмов построения MST (минимальное остовное дерево).

Задача о рюкзаке (непрерывном)

Нахождение кратчайшего пути (вариации и размышления)

Выводы

- Жадные алгоритмы конструируют решения итеративно, посредством последовательности близоруких решений в надежде, что в конце полученные решение будет оптимальным
- Жадные алгоритмы не всегда являются правильными
- Важно проверять выполнение обоих условий, но это не всегда бывает тривиально

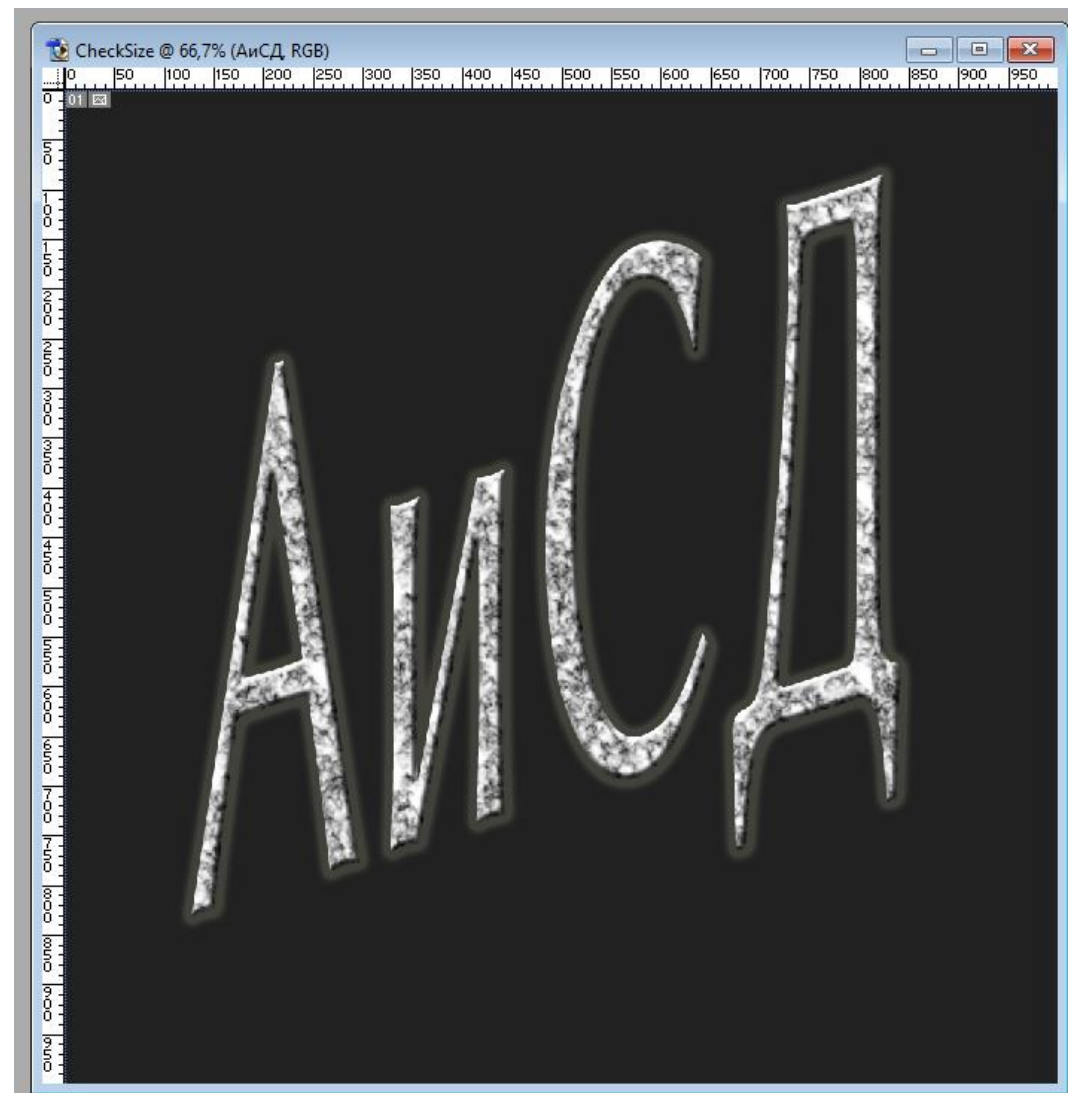
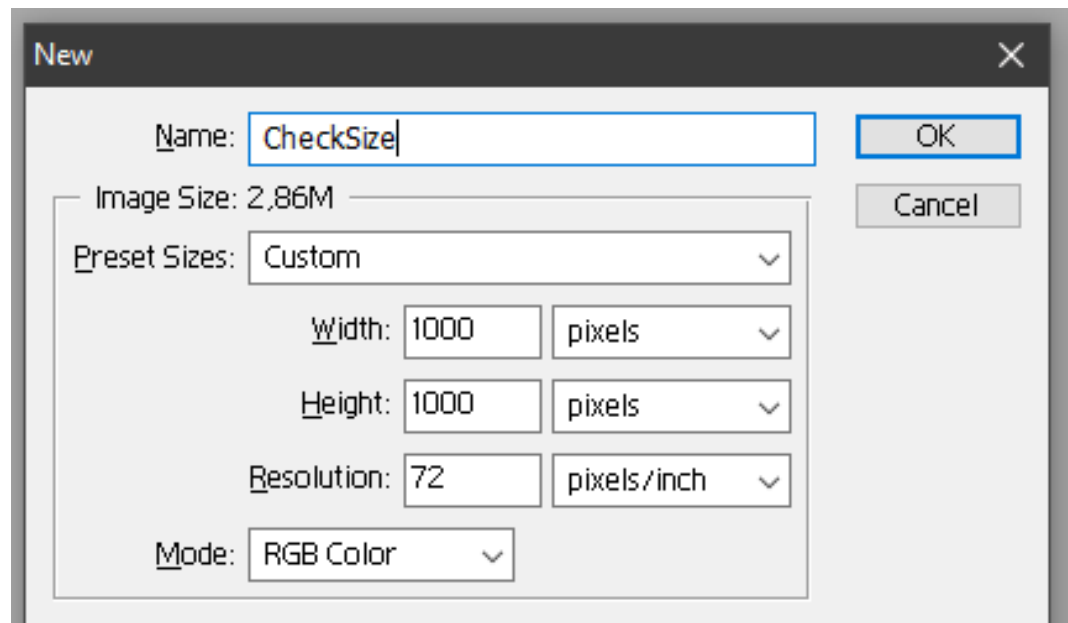
Сжимающее кодирование

Зачем нужно сжатие данных

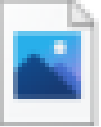
Пример из обычной жизни:

Зачем нужно сжатие данных

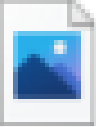
Пример из обычной жизни:





Зачем нужно сжатие данных

	CheckSize.bmp
Тип файла:	Файл "BMP" (.bmp)
Приложение:	Фотографии
Расположение:	C:\Users\Максим\Desktop
Размер:	2,86 МБ (3 000 056 байт)
На диске:	2,86 МБ (3 002 368 байт)

Зачем нужно сжатие данных

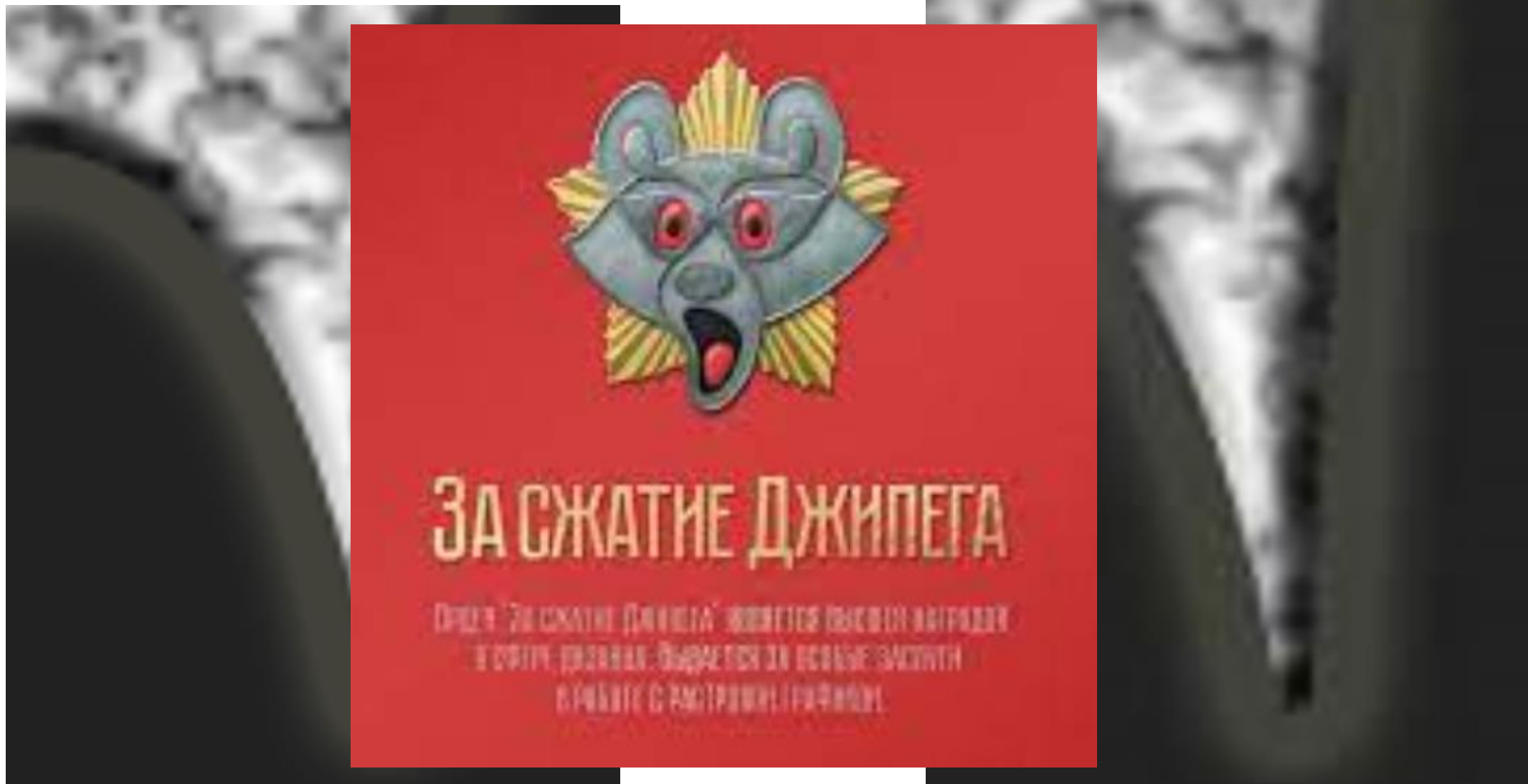
	CheckSize.bmp
Тип файла:	Файл "BMP" (.bmp)
Приложение:	Фотографии
Расположение:	C:\Users\Максим\Desktop
Размер:	2,86 МБ (3 000 056 байт)
На диске:	2,86 МБ (3 002 368 байт)

	CheckSize.jpg
Тип файла:	IrfanView JPG File (.jpg)
Приложение:	 IrfanView 64-bit
Расположение:	C:\Users\Максим\Desktop
Размер:	50,1 КБ (51 332 байт)
На диске:	52,0 КБ (53 248 байт)

Но здесь есть одно но...

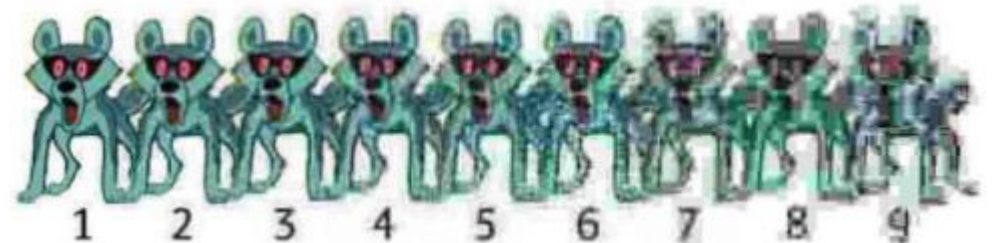


Но здесь есть одно но...



Особенности сжатия

На примере сжатия картинок мы увидели так называемое **сжимание с потерей** (качества). Теряем мы изначальную информацию, отбрасывая её. За счёт такого действия мы выигрываем в требуемом объёме данных для хранения, но при этом проигрываем в их полноте.



Особенности сжатия

На примере сжатия картинок мы увидели так называемое **сжимание с потерей** (качества). Теряем мы изначальную информацию, отбрасывая её. За счёт такого действия мы выигрываем в требуемом объёме данных для хранения, но при этом проигрываем в их полноте.

Можно ли такой фокус провернуть с музыкой?

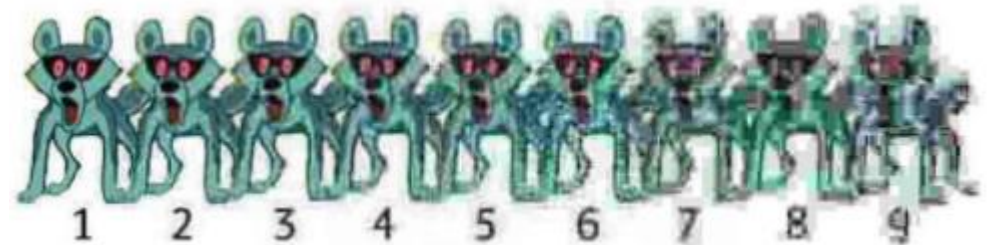


Особенности сжатия

На примере сжатия картинок мы увидели так называемое **сжимание с потерей** (качества). Теряем мы изначальную информацию, отбрасывая её. За счёт такого действия мы выигрываем в требуемом объёме данных для хранения, но при этом проигрываем в их полноте.

Можно ли такой фокус провернуть с музыкой?

А с текстом?



Сжатие текста

Со сжатием текста так не получится, т.к. ценность текстовой информации заключается именно в последовательности букв и символов, а сжимающее кодирование будет их нарушать, превращая текст в кашу.

Сжатие текста

Со сжатием текста так не получится, т.к. ценность текстовой информации заключается именно в последовательности букв и символов, а сжимающее кодирование будет их нарушать, превращая текст в кашу.

Поэтому для текста требуется **сжатие без потерь**.

Такое сжатие пытается уменьшить «избыточность» информации.

Основной способ:

Сжатие текста

Со сжатием текста так не получится, т.к. ценность текстовой информации заключается именно в последовательности букв и символов, а сжимающее кодирование будет их нарушать, превращая текст в кашу.

Поэтому для текста требуется **сжатие без потерь**.
Такое сжатие пытается уменьшить «избыточность» информации.

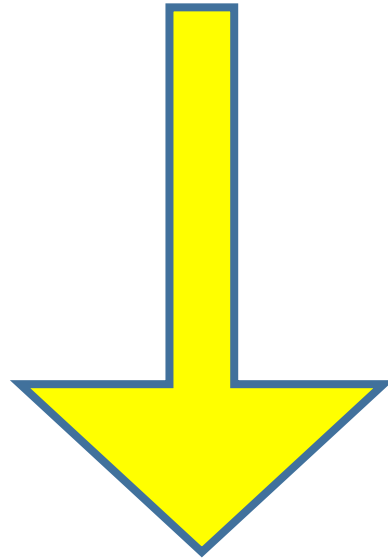
Основной способ:

замена длинных последовательностей на более короткие

Как ищем такие последовательности?

Анализируем
сочетания букв

Анализируем
текст побуквенно

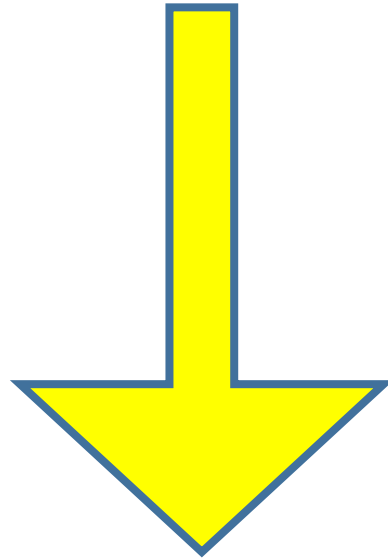


Статистика

Как ищем такие последовательности?

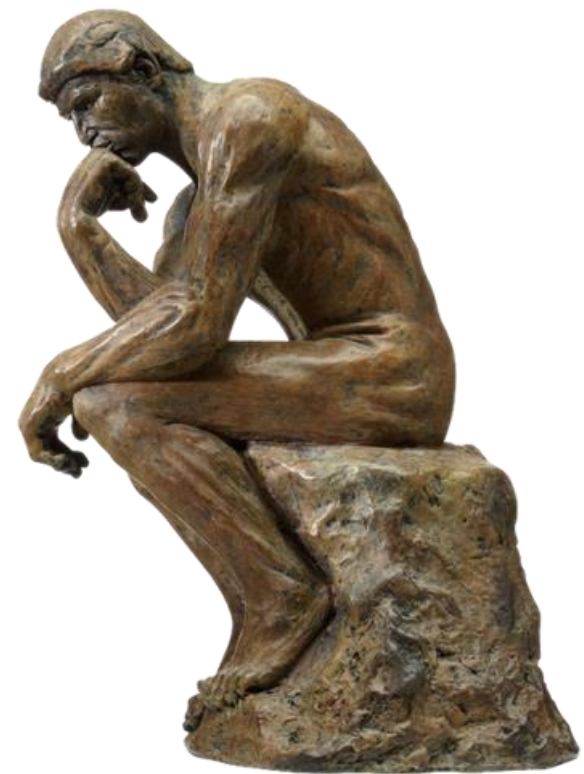
Анализируем
сочетания букв

Анализируем
текст побуквенно



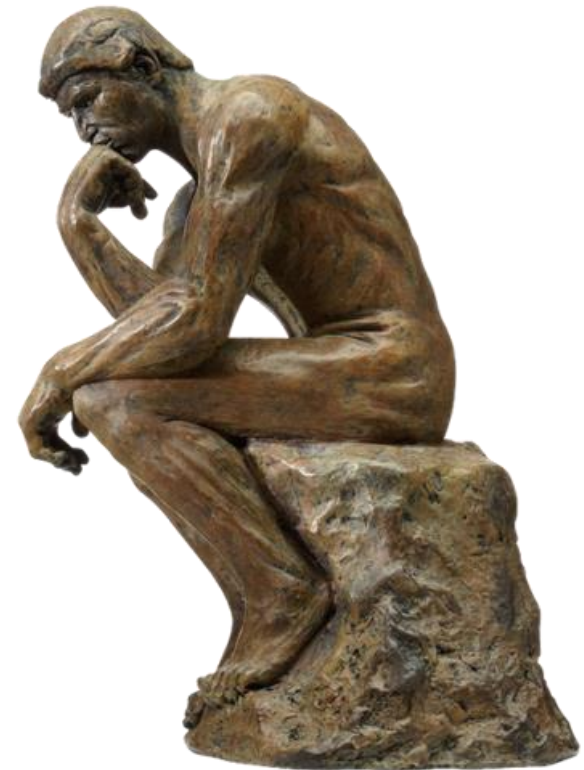
Статистика

Что такое текст?



Что такое текст?

С первого семестра помним, что буквы в тексте кодируются определённым набором бит. За такие наборы отвечают таблицы кодировки.



Что такое текст?

С первого семестра помним, что буквы в тексте кодируются определённым набором бит. За такие наборы отвечают таблицы кодировки.

Самый простой пример: ASCII-таблица

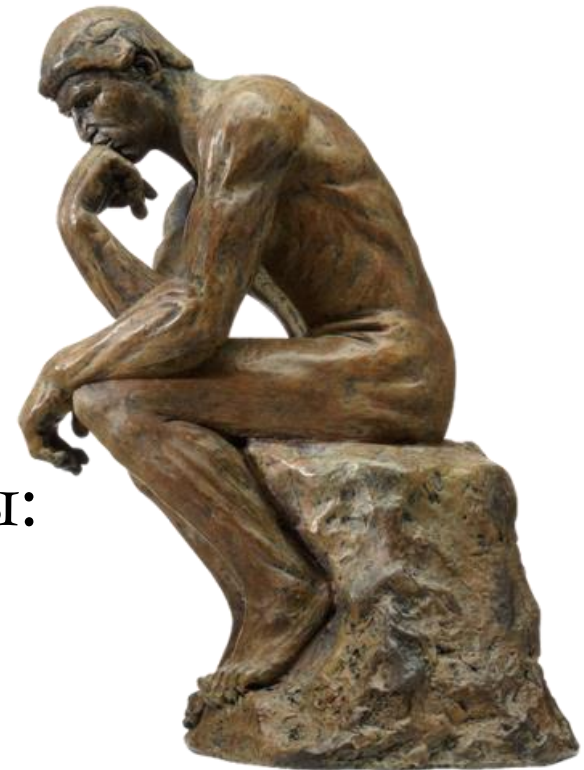
A – 0x41

B – 0x42

Z – 0x5A

Такой код везде один — код **фиксированной** длины:

На каждый символ — 1 байт.



Что такое текст?

С первого семестра помним, что буквы в тексте кодируются определённым набором бит. За такие наборы отвечают таблицы кодировки.

Самый простой пример: ASCII-таблица

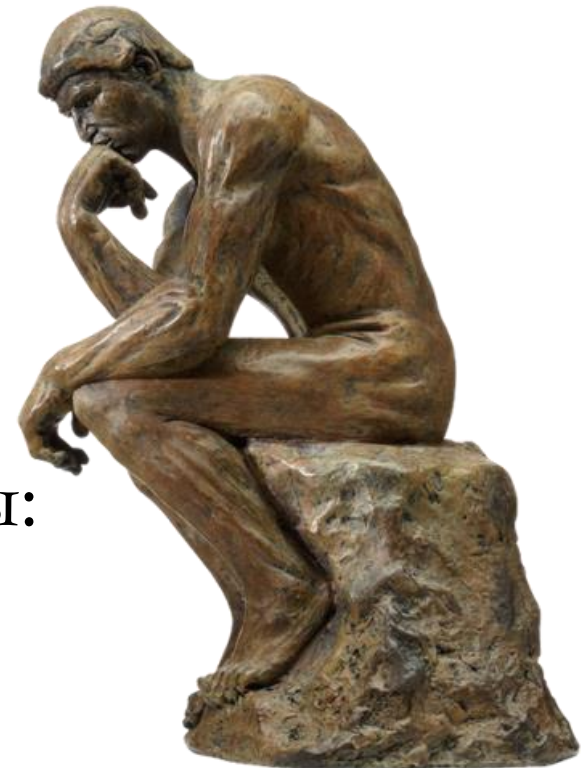
A – 0x41

B – 0x42

Z – 0x5A

Такой код везде один — код **фиксированной** длины:
На каждый символ — 1 байт.

А можно ли лучше? Код переменной длины?



Алгоритм Хаффмана. Основная идея

Можем задавать код символа в зависимости от частоты его встречи в тексте

Чем чаще встречаем символ, тем меньше бит будем тратить на кодирование этого символа.

Пример:

Текст: АААСССС. А будем нулем (0), а С – единицей (1).

ASCII-кодировка: $7 * 1 \text{ байт/символ} = 7 \text{ байтов}$.

Haffman-encode: $1 \text{ бит/символ} * 7 = 1 \text{ байт}$: **0001111**

Закодируем же быстрее!

Рассмотрим алфавит $\Sigma = \{A, B, C, D\}$

Код фиксированной длины

Символ	Кодирование
<i>A</i>	00
<i>B</i>	01
<i>C</i>	10
<i>D</i>	11

Код переменной длины

Символ	Кодирование
<i>A</i>	0
<i>B</i>	01
<i>C</i>	10
<i>D</i>	1

Сколько тратим на хранение сообщения «BCCDAAD» ???

Закодируем же быстрее!

Рассмотрим алфавит $\Sigma = \{A, B, C, D\}$

Код фиксированной длины

Символ	Кодирование
<i>A</i>	00
<i>B</i>	01
<i>C</i>	10
<i>D</i>	11

Код переменной длины

Символ	Кодирование
<i>A</i>	0
<i>B</i>	01
<i>C</i>	10
<i>D</i>	1

Сколько затратим на хранение сообщения «BCCDAAD» ???

На 25% меньше!

Первая проблема

Закодировать-то мы закодировали. Но как теперь декодировать?

Сообщение с прошлого слайда:

011010101001

Символ	Кодирование
<i>A</i>	0
<i>B</i>	01
<i>C</i>	10
<i>D</i>	1

Первая проблема

Закодировать-то мы закодировали. Но как теперь декодировать?

Сообщение с прошлого слайда:

011010101001

Символ	Кодирование
<i>A</i>	0
<i>B</i>	01
<i>C</i>	10
<i>D</i>	1

Никак! Потому что возникла неоднозначность при декодировании.

Первая проблема

Закодировать-то мы закодировали. Но как теперь декодировать?

Сообщение с прошлого слайда:

011010101001

Символ	Кодирование
<i>A</i>	0
<i>B</i>	01
<i>C</i>	10
<i>D</i>	1

Никак! Потому что возникла неоднозначность при декодировании.

Всё, приплыли?

Решение – беспрефиксный код

Такой код, в котором для любых
символов $a, b \in \Sigma$ их коды не являются
префиксами друг друга.

Решение – беспрефиксный код

Такой код, в котором для любых символов $a, b \in \Sigma$ их коды не являются префиксами друг друга.

Внимательнее посмотрим на таблицу кодировки.
Это префиксный код?

Символ	Кодирование
<i>A</i>	0
<i>B</i>	01
<i>C</i>	10
<i>D</i>	1

Решение – беспрефиксный код

Такой код, в котором для любых символов $a, b \in \Sigma$ их коды не являются префиксами друг друга.

Символ	Кодирование
<i>A</i>	0
<i>B</i>	01
<i>C</i>	10
<i>D</i>	1



Символ	Кодирование
<i>A</i>	0
<i>B</i>	10
<i>C</i>	110
<i>D</i>	111

Как генерировать такие коды?

Требуется алгоритмически генерировать беспрефиксные коды минимальной длины. На помощь приходят бинарные деревья. Почему?

Как генерировать такие коды?

Требуется алгоритмически генерировать беспрефиксные коды минимальной длины. На помощь приходят бинарные деревья. Почему?

Символ	Кодирование
<i>A</i>	00
<i>B</i>	01
<i>C</i>	10
<i>D</i>	11

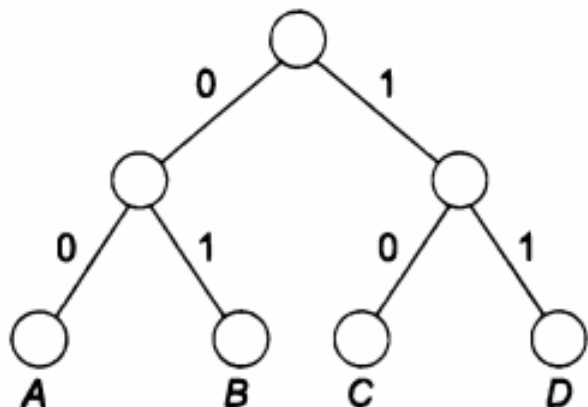
Символ	Кодирование
<i>A</i>	0
<i>B</i>	01
<i>C</i>	10
<i>D</i>	1

Символ	Кодирование
<i>A</i>	0
<i>B</i>	10
<i>C</i>	110
<i>D</i>	111

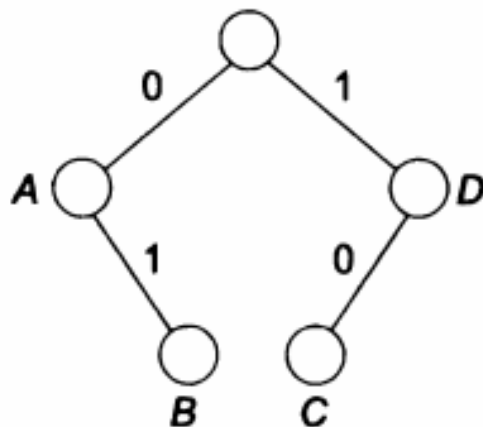
Как генерировать такие коды?

Требуется алгоритмически генерировать беспрефиксные коды минимальной длины. На помощь приходят бинарные деревья. Почему?

Символ	Кодирование
<i>A</i>	00
<i>B</i>	01
<i>C</i>	10
<i>D</i>	11



Символ	Кодирование
<i>A</i>	0
<i>B</i>	01
<i>C</i>	10
<i>D</i>	1

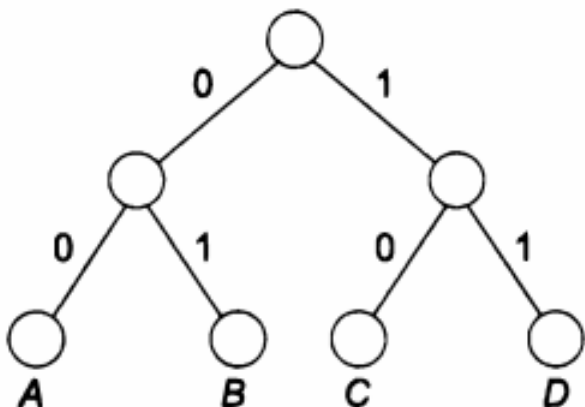


Символ	Кодирование
<i>A</i>	0
<i>B</i>	10
<i>C</i>	110
<i>D</i>	111

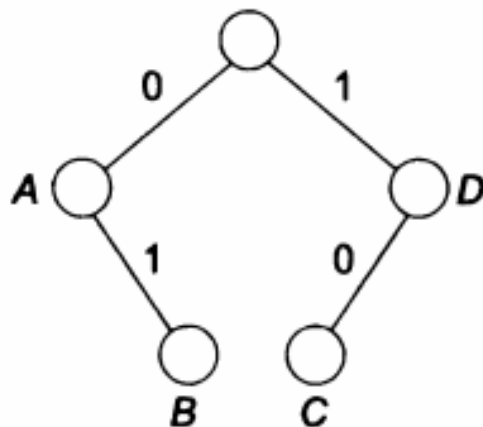
Как генерировать такие коды?

Требуется алгоритмически генерировать беспрефиксные коды минимальной длины. На помощь приходят бинарные деревья. Почему?

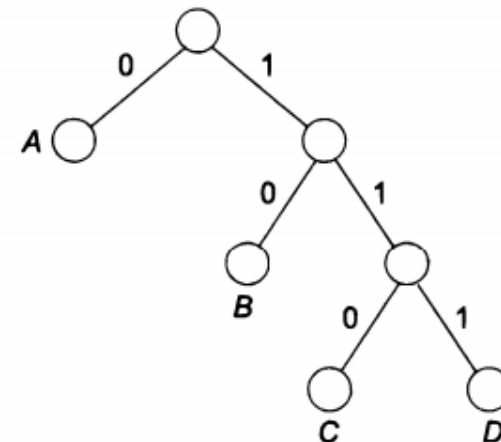
Символ	Кодирование
<i>A</i>	00
<i>B</i>	01
<i>C</i>	10
<i>D</i>	11



Символ	Кодирование
<i>A</i>	0
<i>B</i>	01
<i>C</i>	10
<i>D</i>	1



Символ	Кодирование
<i>A</i>	0
<i>B</i>	10
<i>C</i>	110
<i>D</i>	111



Формулируем определение

Каждый двоичный код может быть представлено в виде двоичного дерева, в котором левое и правое дочерние рёбра помечены соответственно 0 и 1, и каждый символ алфавита используется в качестве метки только для одного узла. И наоборот.

Формулируем определение

Каждый двоичный код может быть представлено в виде двоичного дерева, в котором левое и правое дочерние рёбра помечены соответственно 0 и 1, и каждый символ алфавита используется в качестве метки только для одного узла. И наоборот.

Вводим ограничение: Помеченными могут быть только листья.

Формулируем определение

Каждый двоичный код может быть представлено в виде двоичного дерева, в котором левое и правое дочерние рёбра помечены соответственно 0 и 1, и каждый символ алфавита используется в качестве метки только для одного узла. И наоборот.

Вводим ограничение: Помеченными могут быть только листья.

**ЗАДАЧА: ОПТИМАЛЬНЫЕ БЕСПРЕФИКСНЫЕ КОДЫ
(В НОВОЙ ФОРМУЛИРОВКЕ)**

Вход: неотрицательная частота p_a для каждого символа a алфавита Σ размера $n \geq 2$.

Выход: Σ -дерево с минимально возможной средней глубиной листа (14.1).

Строим дерево – алгоритм Хаффмана

Суть алгоритма Хаффмана заключается в использовании восходящего подхода, начиная с листьев дерева.

Символ	Частота
<i>A</i>	0,60
<i>B</i>	0,25
<i>C</i>	0,10
<i>D</i>	0,05

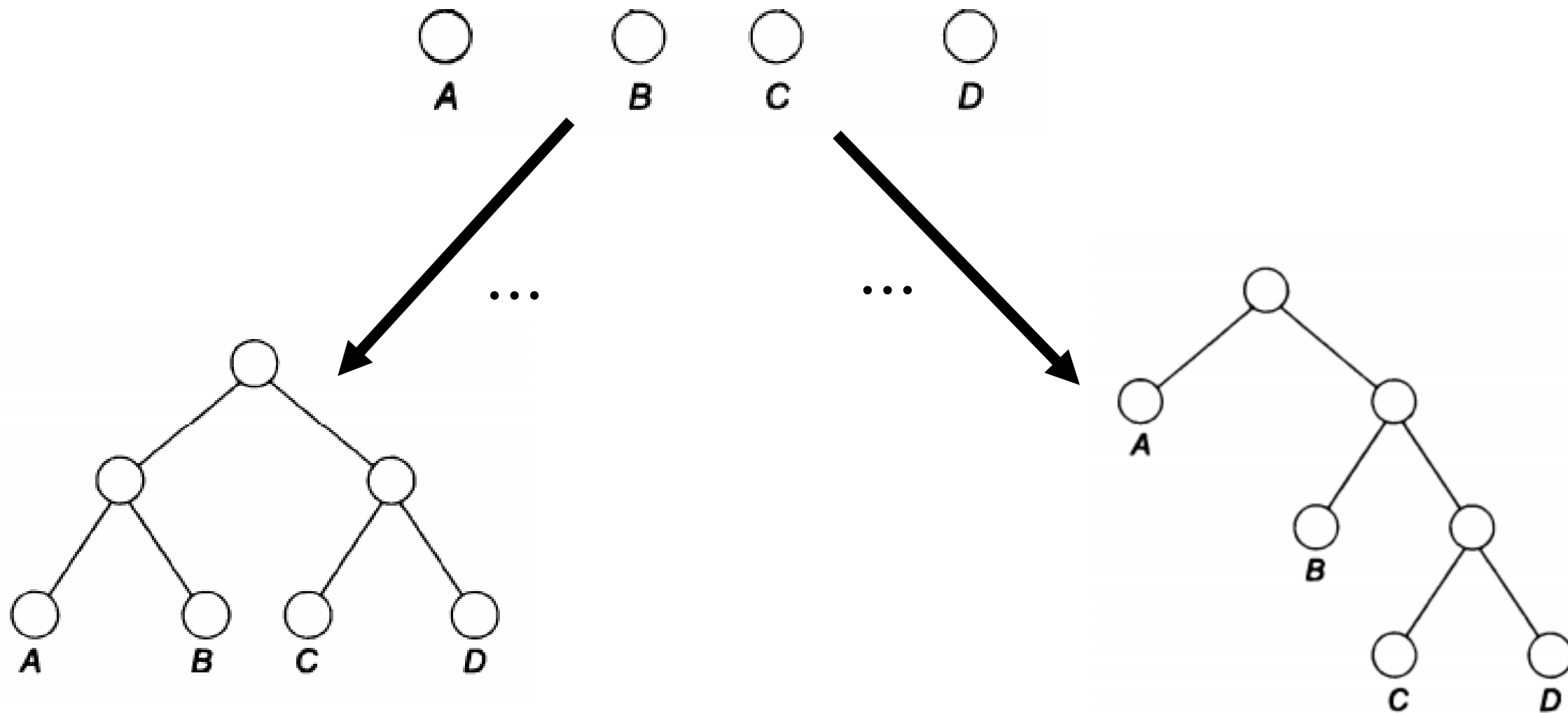
○
A

○
B

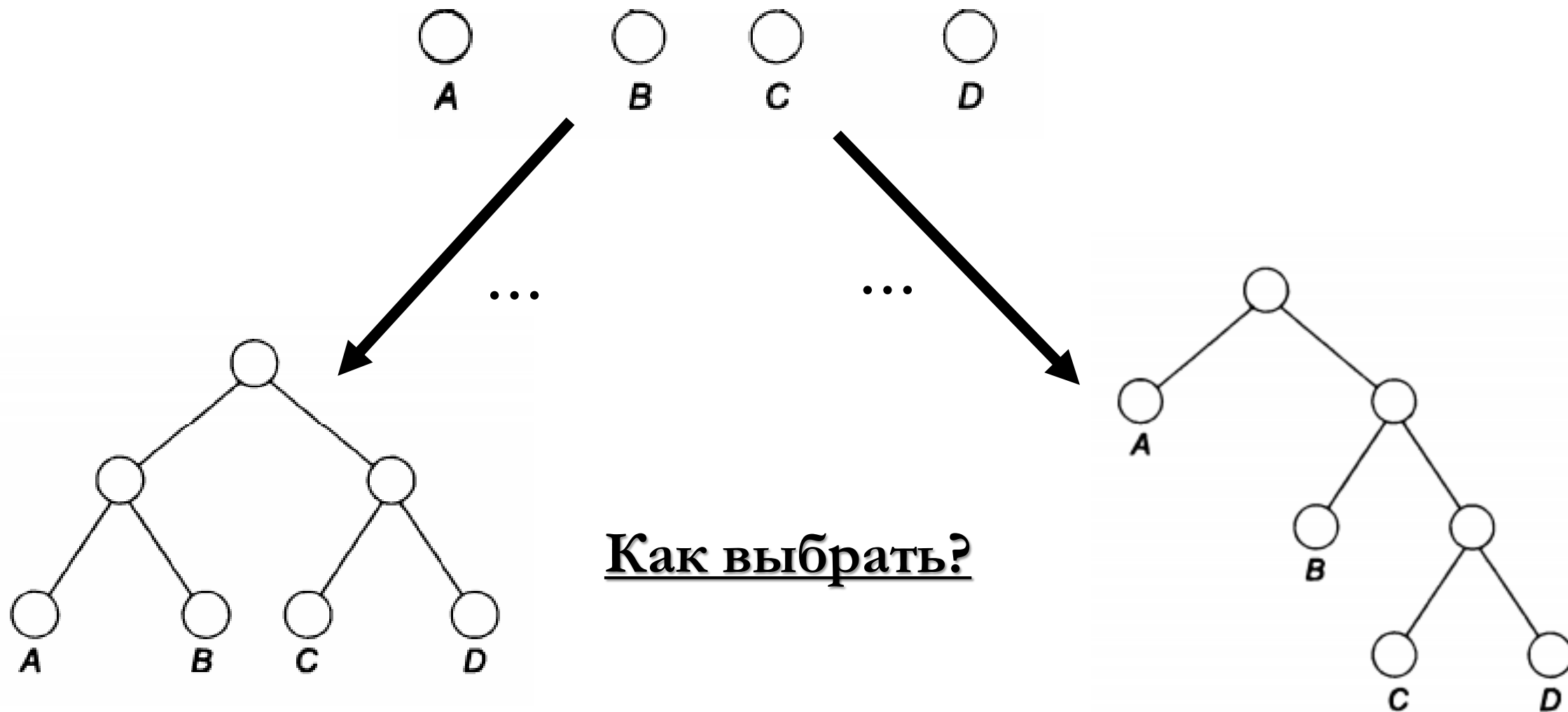
○
C

○
D

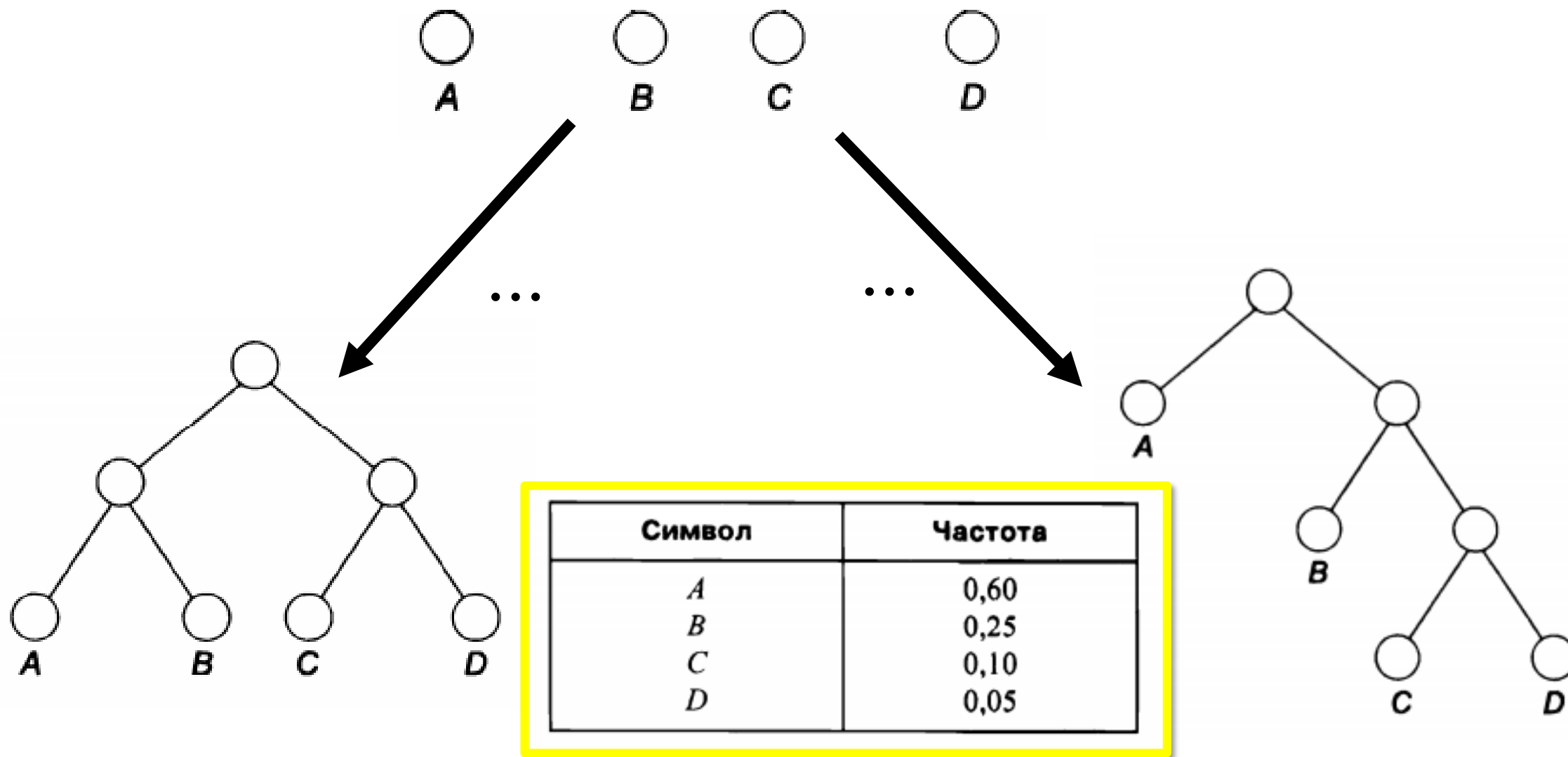
Строим дерево – алгоритм Хаффмана



Строим дерево – алгоритм Хаффмана



Строим дерево – алгоритм Хаффмана



Жадный критерий хаффмана

Шаг алгоритма: слияние деревьев.

Какие деревья выбрать?

Жадный критерий хаффмана

Шаг алгоритма: слияние деревьев.

Какие деревья выбрать?

Критерий: слияние должно приводить к минимального возможному увеличению средней глубины листа.

Жадный критерий хатфмана

Шаг алгоритма: слияние деревьев.

Какие деревья выбрать?

Критерий: слияние должно приводить к минимального возможному увеличению средней глубины листа.

Для каждого символа a в одном из двух участвующих деревьев глубина соответствующего листа увеличивается на 1, а вклад соответствующего члена в сумму (14.1) увеличивается на p_a . Таким образом, слияние двух деревьев T_1 и T_2 увеличивает среднюю глубину листа на сумму частот участвующих символов:

$$\sum_{a \in T_1} p_a + \sum_{a \in T_2} p_a, \quad (14.2)$$

Жадный критерий хатфмана

Шаг алгоритма: слияние деревьев.

Какие деревья выбрать?

Критерий: слияние должно приводить к минимального возможному увеличению средней глубины листа.

Символ	Частота
<i>A</i>	0,60
<i>B</i>	0,25
<i>C</i>	0,10
<i>D</i>	0,05

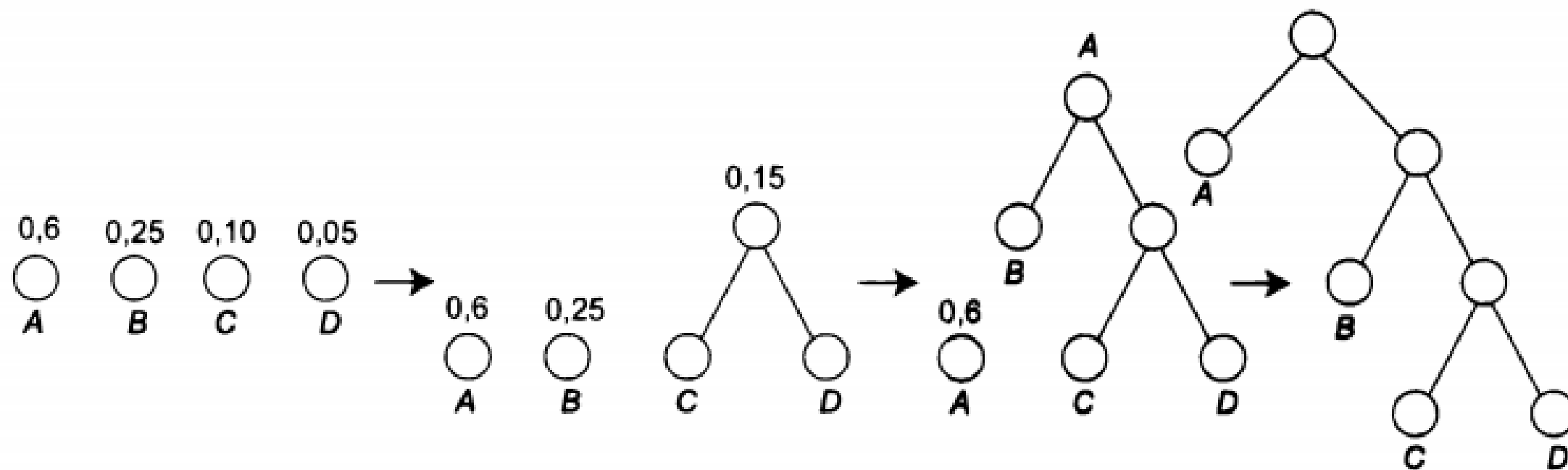
Для каждого символа a в одном из двух участвующих деревьев глубина соответствующего листа увеличивается на 1, а вклад соответствующего члена в сумму (14.1) увеличивается на p_a . Таким образом, слияние двух деревьев T_1 и T_2 увеличивает среднюю глубину листа на сумму частот участвующих символов:

$$\sum_{a \in T_1} p_a + \sum_{a \in T_2} p_a,$$

(14.2)



Шаги построения дерева



Анализ сложности

Анализ сложности

Построение таблицы распределения символов

Пока больше одного узла:

ищем два минимальных

объединяем

кладем обратно

Анализ сложности

Построение таблицы распределения символов

Построение кучи из массива деревьев

Пока больше одного узла:

ищем два минимальных

объединяем

кладем обратно

ВЫВОДЫ

- ★ Беспрефиксные двоичные коды переменной длины могут иметь меньшие средние длины кодирования, чем коды фиксированной длины, когда разные символы алфавита имеют разные частоты.
- ★ Жадный алгоритм Хаффмана поддерживает лес, где листья находятся в соответствии с символами алфавита, и на каждой итерации жадно выполняет слияние пары деревьев, вызывая минимально возможное увеличение средней глубины листа.
- ★ Алгоритм Хаффмана гарантированно вычисляет беспрефиксный код с минимально возможной средней длиной кодирования.
- ★ Алгоритм Хаффмана может быть реализован с работой за время $O(n \log n)$, где n — это число символов.

```
func buildHuffmanTree(charFreq freqsTable) *huffmanBTNode {  
    nodes := make(heapOfNodes, 0, len(charFreq))  
    for char, freq := range charFreq {  
        nodes = append(nodes, &huffmanBTNode{chars: []rune{char}, weight: freq})  
    }  
    heap.Init(&nodes)  
  
    for len(nodes) > 1 {  
        leftNode := heap.Pop(&nodes).(*huffmanBTNode)  
        rightNode := heap.Pop(&nodes).(*huffmanBTNode)  
        newNode := mergeHuffmanBTNodes(leftNode, rightNode)  
        heap.Push(&nodes, newNode)  
    }  
    return heap.Pop(&nodes).(*huffmanBTNode)  
}
```

```
199 func generateCodesByTreeTraverse(root *huffmanBTNode, codes encodeTable) {
200     if root.IsLeaf() {
201         codes[root.chars[0]] = "0"
202         return
203     }
204
205     var traverse func(rootNode *huffmanBTNode, prevCode string)
206     traverse = func(rootNode *huffmanBTNode, prevCode string) {
207         if rootNode.IsLeaf() {
208             if len(rootNode.chars) != 1 {
209                 panic("Leaf has != 1 lenght of chars")
210             }
211             codes[rootNode.chars[0]] = prevCode
212             return
213         }
214         traverse(rootNode.left, prevCode+"0")
215         traverse(rootNode.right, prevCode+"1")
216     }
217     traverse(root, "")
218 }
```

Лекция 10:

Очередь с приоритетом

Берленко Татьяна Андреевна,
МОЭВМ

2021

tatyana.berlenko@moevm.info

Очереди с приоритетом

Абстрактный тип данных, поддерживающий следующие операции:

- добавить новый элемент в очередь с заданным приоритетом;
- извлечь элемент с максимальным приоритетом;
- удалить заданный элемент;
- нахождение максимальный приоритет;
- изменить приоритет заданного элемента.

Очередь с приоритетом

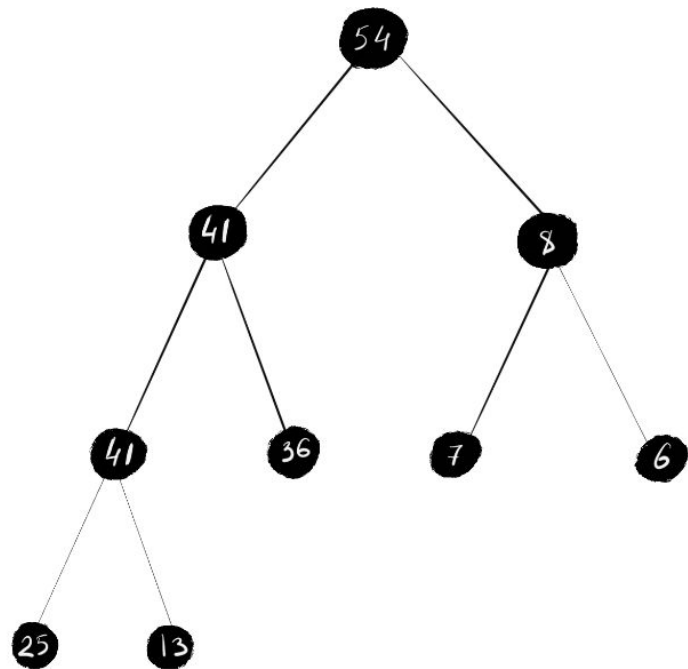
- Простейшие реализации:
 - Связный список / массив
 - Упорядоченный массив / список
- Эффективная реализация:
 - Двоичная куча

Двоичная макс-куча (Binary heap)

➤ Определение

Двоичное дерево, в котором значение в каждой вершине не меньше значений в ее детях.

Дерево должно быть полностью заполнено на всех уровнях, кроме последнего (он заполняется слева направо)



Куча. Операции

- Найти максимальное значение: это корень
- Вставка: подвесить к листу новую вершину, затем просеять вверх
- Извлечь максимум: обмен корня и листа, просеивание вниз
- Удалить элемент: изменить приоритет на бесконечность, просеять вверх, извлечь максимум

Вопросы

- Дан массив:

3, 10, 3, 8, 11, 3, 3, 9, 9

Сколько нарушений свойства мин-кучи в этом массиве?

- Сколько минимально и максимально может быть вершин в двоичной куче высоты 4? Для высоты 10? Считаем, что высота - это максимальное количество **вершин** от корня к листу.

- Дана макс-куча

5, 4, 3

Были произведены операции: insert(6), insert(3), insert(4), extractMax().

Как сейчас выглядит макс-куча?

Практика:
построение двоичной макс-
кучи

Сортировка кучей

- in-place
- $O(n \log n)$ в лучшем, среднем, худшем случае
- Используется для внешней сортировки данных большого размера

Практика:
решение задачи на
построение кучи

Полезные ссылки

- [Двоичная куча](#)
- [Лекции Куликова](#)