

Алгоритмы

Заметки для Профессионалов

Chapter 15: Applications of Dynamic Programming

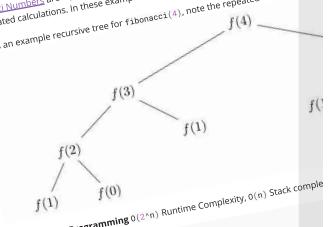
The basic idea behind dynamic programming is breaking a complex problem down to several problems that are repeated. If you can identify a simple subproblem that is repeatedly calculated in a dynamic programming approach to the problem.

As this topic is titled Applications of Dynamic Programming, it will focus more on applications of creating dynamic programming algorithms.

Section 15.1: Fibonacci Numbers

Fibonacci Numbers are a prime subject for dynamic programming as the traditional recursive solution has a lot of repeated calculations. In these examples I will be using the base case of $f(0) = f(1) = 1$.

Here is an example recursive tree for $f_{\text{fibonacci}}(4)$, note the repeated computations:



Non-Dynamic Programming $O(2^n)$ Runtime Complexity, $O(n)$ Stack complexity

```
def fibonaccin():
    if n < 2:
        return 1
    return fibonaccin(n-1) + fibonaccin(n-2)
```

This is the most intuitive way to write the problem. At most the stack space will be $O(2^n)$.

The $O(2^n)$ runtime complexity proof that can be seen here: [Complexity](#). The main point to note is that the runtime is exponential, which means the subsequent term, $f_{\text{fibonacci}}(15)$ will take twice as long as $f_{\text{fibonacci}}(14)$.

Memoized $O(n)$ Runtime Complexity, $O(n)$ Space complexity, $O(n)$ Stack complexity

```
memo = []
memo.append(0) # f(0) = 0
memo.append(1) # f(1) = 1
memo.append(1) # f(2) = 1

def fibonaccin():
    if len(memo) > n:
        return memo[n]

    if fibonaccin() == None:
        memo.append(fibonaccin(n-1) + fibonaccin(n-2))

    return memo[n]
```

GoalKicker.com - Algorithms Notes for Professionals

Chapter 6: Check if a tree is BST or not

Section 6.1: Algorithm to check if a given binary tree is BST

A binary tree is BST if it satisfies any one of the following condition:

1. It is empty.
2. It has no subtrees.
3. For every node x in the tree all the keys (if any) in the left sub tree must be less than $\text{key}(x)$ and all the keys (if any) in the right sub tree must be greater than $\text{key}(x)$.

So a straightforward recursive algorithm would be:

```
is_BST(root):
    if root == NULL:
        return true

    // Check values in left subtree
    if root->left != NULL:
        max_key_in_left = find_max_key(root->left)
        if max_key_in_left > root->key:
            return false

    // Check values in right subtree
    if root->right != NULL:
        min_key_in_right = find_min_key(root->right)
        if min_key_in_right < root->key:
            return false

    return is_BST(root->left) && is_BST(root->right)
```

The above recursive algorithm is correct but inefficient, because it traverses each node multiple times.

Another approach to minimize the multiple visits of each node is to remember the min and max possible values of the keys in the subtree we are visiting. Let the minimum possible value of any key be K_{MIN} and maximum value be K_{MAX} . When we start from the root of the tree, the range of values in the tree is $[K_{\text{MIN}}, K_{\text{MAX}}]$. Let the key of root node be x . Then the range of values in left subtree is $[K_{\text{MIN}}, x]$ and the range of values in right subtree is $(x, K_{\text{MAX}}]$. We will use this idea to develop a more efficient algorithm.

```
is_BST(root, min, max):
    if root == NULL:
        return true

    // Is the current node key out of range?
    if root->key < min || root->key > max:
        return false

    // check if left and right subtree is BST
    return is_BST(root->left, min, root->key) && is_BST(root->right, root->key+1, max)
```

It will be initially called as:

```
is_BST(my_tree.root, KEY_MIN, KEY_MAX)
```

Another approach will be to do inorder traversal of the Binary tree. If the inorder traversal produces a sorted sequence of keys then the given tree is a BST. To check if the inorder sequence is sorted remember the value of previous node.

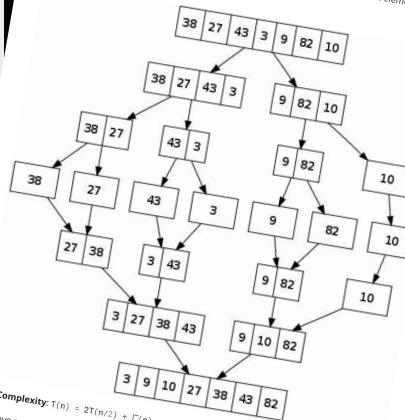
GoalKicker.com - Algorithms Notes for Professionals

Chapter 30: Merge Sort

Section 30.1: Merge Sort Basics

Merge Sort is a divide-and-conquer algorithm. It divides the input list of length n in half successively until there are n lists of size 1. Then, pairs of lists are merged together with the smaller first element among the pair of lists being added in each step. Through successive merging and through comparison of first elements, the sorted list is built.

An example:



Time Complexity: $T(n) = 2T(n/2) + C(n)$

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $C(n \log n)$. Time complexity of Merge Sort is $C(n \log n)$ in all 3 cases (worst, average and best as merge sort always divides the array in two halves and take linear time to merge two halves).

Auxiliary Space: $O(n)$

Algorithmic Paradigm: Divide and Conquer

GoalKicker.com - Algorithms Notes for Professionals

200+ страниц,

профессиональных советов и уловок

Оглавление

О книге	7
Глава 1: Приступим к алгоритмам.....	8
Раздел 1.1: Образец алгоритмической задачи	8
Раздел 1.2: Приступим к простой реализации Fizz Buzz алгоритма на Swift	8
Глава 2: Сложность Алгоритмов.....	11
Раздел 2.1: Большая-Тета нотация	11
Раздел 2.2: Сравнение асимптотических обозначений	11
Раздел 2.3: Большая-Омега нотация	12
Глава 3: Обозначение O-большое	14
Раздел 3.1: Простой цикл.....	15
Раздел 3.2: Вложенный цикл	16
Раздел 3.3: Типы алгоритмов $O(\log n)$	17
Раздел 3.4: Пример $O(\log n)$	19
Глава 4: Деревья	21
Раздел 4.1: Типичное представление анарного дерева	21
Раздел 4.2: Введение.....	22
Раздел 4.3: Проверка, одинаковы два двоичных дерева или нет	23
Глава 5: Поиск в двоичных деревьях.....	25
Раздел 5.1: Дерево поиска двоичных файлов - Вставка (Python)	25
Раздел 5.2: Дерево двоичного поиска - удаление (C++)	27
Раздел 5.3: Самый низкий предок в BST	29
Раздел 5.4: Поиск в двоичном дереве - Python	30
Глава 6: Проверка, является дерево BST или нет	32
Раздел 6.1: Алгоритм проверки, является ли данное двоичное дерево BST	32
Раздел 6.2: Имеет ли данное на входе дерево свойства бинарного дерева поиска или нет	33
Глава 7: Обходы бинарного дерева	34
Раздел 7.1: Реализация обхода в порядке уровней	34
Раздел 7.2: Прямой, симметричный и обратный обход бинарного дерева	35
Глава 8: Самый нижний общий предок бинарного дерева	37
Раздел 8.1: Нахождение самого нижнего общего предка	37
Глава 9: Граф	38
Раздел 9.1: Хранение графов (матрицы смежности)	38
Раздел 9.2: Введение в теорию графов.....	42
Раздел 9.3: Хранение графов (Список смежности)	45
Раздел 9.4: Топологическая сортировка	47
Раздел 9.5: Обнаружение цикла в ориентированном графе	48

Раздел 9.6: Алгоритм Торупа	50
Глава 10: Обходы графов.....	52
Раздел 10.1: Функция обхода поиска в глубину.....	52
Глава 11: Алгоритм Дейкстры	53
Раздел 11.1: Алгоритм Дейкстры по поиску кратчайшего пути	53
Глава 12: Поиск пути A*	59
Раздел 12.1: Введение в A*	59
Раздел 12.2: Поиск пути A* в лабиринте без препятствий	59
Раздел 12.3: Решаем “Пятнашки” (3x3) используя алгоритм A*	66
Глава 13: Алгоритм поиска пути A*	69
Раздел 13.1: Простой пример алгоритма поиска пути A*: Лабиринт без препятствий	69
Глава 14: Динамическое программирование	76
Раздел 14.1: Расстояние Левенштейна	76
Раздел 14.2: Алгоритм взвешенного планирования работ	77
Раздел 14.3: Самая длинная общая подпоследовательность.....	81
Раздел 14.4. Число Фибоначчи	83
Раздел 14.5: Самая длинная общая подстрока	84
Глава 15: Применение динамического программирования	85
Раздел 15.1: Числа Фибоначчи	85
Глава 16. Алгоритм Крускала	88
Раздел 16.1: Оптимальное применение на основе непересекающихся множеств.	88
Раздел 16.2: Простое, более детальное использование	89
Раздел 16.3: Простое использование, основанное на непересекающихся множествах	90
Раздел 16.4: Простое высокоуровневое использование.....	90
Глава 17: Жадные алгоритмы	91
Раздел 17.1: Кодировка Хаффмана	91
Раздел 17.2: Проблема выбора вида деятельности	95
Раздел 17.3: Проблема выдачи сдачи	97
Глава 18: Применение жадной стратегии	99
Раздел 18.1: Оффлайн кэширование	99
Раздел 18.2: Автомат билетов	109
Раздел 18.3: Интервальное планирование	112
Раздел 18.4: Минимизация задержки	117
Глава 19: Алгоритм Прима	121
Раздел 19.1: Введение в алгоритм Прима	121
Глава 20: Алгоритм Беллмана-Форда	130
Раздел 20.1: Алгоритм поиска кратчайшего пути от одной вершины	130
Раздел 20.2. Обнаружение отрицательного цикла в графе.....	134
Раздел 20.3: Почему нам нужно ослабить все ребра в (V- 1) раз	136
Глава 21: Линейный алгоритм	139
Раздел 21.1: Алгоритм рисования линий Брезенхэма	139
Глава 22: Алгоритм Флойда- Уоршелла	143

Раздел 22.1. Алгоритм кратчайшего пути для всех пар	143
<hr/>	
Глава 23: Числа Каталана	147
Раздел 23.1: Числа Каталана. Основная информация	147
<hr/>	
Глава 24: Многопоточные алгоритмы	149
Раздел 24.1: Многопоточное умножение квадратной матрицы	149
Раздел 24.2: Многопоточное умножение вектора матрицы	149
Раздел 24.3: Многопоточная сортировка слиянием	149
<hr/>	
Глава 25: Алгоритм Кнута-Морриса-Пратта (KMP)	152
Раздел 25.1: Пример KMP	152
<hr/>	
Глава 26: Алгоритм изменения динамического расстояния	155
Раздел 26.1: Преобразования строки 1 в строку 2	155
<hr/>	
Глава 27: Онлайн алгоритмы	158
Раздел 27.1: Пейджирование (онлайн кэширование)	160
<hr/>	
Глава 28: Сортировка	166
Раздел 28.1: Устойчивость сортировок	166
<hr/>	
Глава 29: Сортировка пузырьком (Bubble Sort)	168
Раздел 29.1: Сортировка пузырьком (Bubble Sort)	168
Раздел 29.2: Реализация на C и C++	168
Раздел 29.3: Реализация на C#	170
Раздел 29.4: Реализация на Python	171
Раздел 29.5: Реализация на Java	171
Раздел 29.6: Реализация на Javascript	172
<hr/>	
Глава 30: Сортировка слиянием	173
Раздел 30.1: Основы сортировки слиянием	173
Раздел 30.2: Реализация сортировки слиянием на Go	174
Раздел 30.3: Реализация сортировки слиянием на C & C#	175
Раздел 30.4: Реализация сортировки слиянием на Java	177
Раздел 30.5: Реализация сортировки слиянием на Python	178
Раздел 30.6: Реализация на Java метода восходящего слияния	178
<hr/>	
Глава 31: Сортировка вставками	180
Раздел 31.1: Реализация на Haskell	180
<hr/>	
Глава 32: Блочная сортировка	181
Раздел 32.1: Реализация на C#	181
<hr/>	
Глава 33: Быстрая сортировка	182
Раздел 33.1: Основы быстрой сортировки	182
33.2: Реализация быстрой сортировки на Python	184
Раздел 33.3: Реализация разбиения Ломуту на Java	184
<hr/>	
Глава 34. Сортировка подсчетом	186
Раздел 34.1: Основная информация о сортировке подсчетом	186
Раздел 34.2: Реализация на псевдокоде	186
<hr/>	
Глава 35: Пирамидальная сортировка	188
Раздел 35.1: Реализация на C#	188

<u>Раздел 35.2: Основная информация о пирамidalной сортировке</u>	189
Глава 36: Сортировка циклом	190
<u>Раздел 36.1: Реализация на псевдокоде</u>	190
Глава 37: Четно-нечетная сортировка	191
<u>Раздел 37.1: Основная информация о четно-нечетной сортировке</u>	191
Глава 38: Сортировка выбором	194
<u>Раздел 38.1: Реализация на Elixir</u>	194
<u>Раздел 38.2: Основная информация о сортировке выбором</u>	194
<u>Раздел 38.3: Реализация Сортировки выбором на C#</u>	196
Глава 39: Поиск	198
<u>Раздел 39.1: Бинарный поиск</u>	198
<u>Раздел 39.2: Рабин Карп</u>	200
<u>Раздел 39.3: Анализ линейного поиска (худший, средний и лучший случаи)</u>	202
<u>Раздел 39.4: Бинарный поиск: на отсортированных числах</u>	204
<u>Раздел 39.5: Линейный поиск</u>	205
Глава 40: Поиск подстроки	206
<u>Раздел 40.1: Введение в алгоритм Кнута-Морриса-Пратта (КМП)</u>	206
<u>Раздел 40.2: Введение в алгоритм Рабина-Карпа</u>	210
<u>Раздел 40.3: Реализация алгоритма КМП на Python</u>	213
<u>Раздел 40.4: Алгоритм КМП на Си</u>	214
Глава 41: Поиск в ширину	218
<u>Раздел 41.1. Нахождение кратчайшего пути от источника к другим узлам</u>	218
<u>Раздел 41.2. Поиск кратчайшего пути от источника в двумерном графе</u>	222
<u>Раздел 41.3: Связные компоненты неориентированного графа с использованием BFS</u>	224
Глава 42: Поиск в глубину	229
<u>Раздел 42.1: Введение в “поиск в глубину”</u>	229
Глава 43: Хеш-функции	234
<u>Раздел 43.1: Хеш-коды для общих типов в C#</u>	234
<u>Раздел 43.2: Введение в хеш-функции</u>	235
Глава 44: Задача коммивояжера	238
<u>Раздел 44.1: Алгоритм полного перебора</u>	238
<u>Раздел 44.2: Алгоритм динамического программирования</u>	238
Глава 45: Задача о рюкзаке	241
<u>Раздел 45.1: Основы задачи о рюкзаке</u>	241
<u>Раздел 45.2: Реализация решения на C#</u>	242
Глава 46: Решение уравнений	244
<u>Раздел 46.1: Линейные уравнения</u>	244
<u>Раздел 46.2: Нелинейное уравнение</u>	246
Глава 47: Самая длинная общая подпоследовательность	251
<u>Раздел 47.1: Объяснение самой длинной общей подпоследовательности</u>	251
Глава 48: Самая длинная возрастающая подпоследовательность	257
<u>Раздел 48.1: Самая длинная возрастающая подпоследовательность</u>	257

Глава 49: Проверка двух строк на анаграмму	261
Раздел 49.1: Ввод и вывод образцов	261
Раздел 49.2: Общий код для анаграмм	262
Глава 50 : Треугольник Паскаля	264
Раздел 50.1: Треугольник Паскаля на С	264
Глава 51: Алгоритм:- Вывод матрицы m^*n по спирали	265
Раздел 51.1: Пример	265
Раздел 51.2: Напишем общий код	265
Глава 52: Возведение матрицы в степень	267
Раздел 52.1: Возведение матрицы в степень для решения типовых задач	267
Глава 53: Алгоритм минимального вершинного покрытия	272
Псевдокод алгоритма	272
Глава 54: Динамическая трансформация временной шкалы (DTW)	273
Раздел 54.1: Введение в динамическую трансформацию временной шкалы (DTW)	273
Глава 55: Быстрое преобразование Фурье	278
Раздел 55.1: Алгоритм Radix-2 для БПФ	278
Раздел 55.2: Обратное быстрое преобразование Фурье с двумя корнями	283
Приложение А: Псевдокод	285
Раздел А.1: Переменные аффектации	285
Раздел А.2: Функции	285
Титры	286

О книге

Пожалуйста, не стесняйтесь делиться этим PDF-файлом бесплатно, последнюю версию книги вы можете скачать по ссылке ниже:
<https://goalkicker.com/AlgorithmsBook>

Книга *Алгоритмы. Заметки для профессионалов* собрана из [документации Stack Overflow](#), содержание написано прекрасными людьми на Stack Overflow. Содержание книги находится под лицензией Creative-Commons BY-SA, смотрите авторов, предоставивших разные главы, в конце книги. Изображения могут быть защищены авторским правом их владельцами, если не указано иное.

Эта неофициальная бесплатная книга создана в учебных целях и она не связана с официальными компаниями или группами об алгоритмах, а также Stack Overflow. Все товарные знаки, в том числе зарегистрированные, являются собственностью соответствующих владельцев.

Информация, представленная в книге, не гарантирует ни правильности, ни точности. Вы используете материал на свой страх и риск.

Пожалуйста, если вы найдете ошибки и неточности, пишите нам на web@petercv.com

Глава 1: Приступим к алгоритмам

Раздел 1.1: Образец алгоритмической задачи

Алгоритмическая задача дается описанием полного набора объектов, с которыми алгоритм должен работать, а также выходных данных после работы над каким-нибудь одним объектом входных данных. Эта разница между задачей и объектом задачи фундаментальна. Алгоритмическая задача, известная как сортировка, определяется следующим образом:

[Skiena:2008:ADM:1410219]

- Задача: Сортировка
- Входные данные: последовательность из n элементов: a_1, a_2, \dots, a_n
- Выходные данные: изменение порядка входной последовательности таким образом, что $a_1 \leq a_2 \leq \dots \leq a_{n-1} \leq a_n$

Объектом сортировки может быть массив из строк, например, { Haskell, Emacs } или последовательность чисел, например, { 154, 245, 1337 }.

Раздел 1.2: Приступим к простой реализации Fizz Buzz алгоритма на Swift

Для тех из вас, кто еще не знаком с программированием в среде Swift и тех, кто начинал обучение с других базовых языков, таких как Python или Java, эта статья должна помочь. Здесь мы обсудим простое решение для создания алгоритмов swift.

Fizz Buzz

Вы, может быть, встречали такие варианты написания как Fizz Buzz, FizzBuzz, или Fizz-Buzz; все они ссылаются на одно и то же. Это “вещь” является основной темой для сегодняшнего обсуждения. Во-первых, что такое FizzBuzz?

Это частый вопрос, который возникает на собеседованиях.

Представьте себе последовательность чисел от 1 до 10.

1 2 3 4 5 6 7 8 9 10

Fizz и Buzz могут означать любое число, делящееся на 3 и на 5 соответственно. Другими словами, если число делится на 3, его заменяют на “fizz”; если оно делится на 5, то его заменяют на “buzz”. Если же число делится И на 3 И на 5 одновременно, то его заменяют на “fizz buzz”. По сути, мы симулируем известную детскую игру “fizz buzz”.

Для работы над этой задачей, откройте Xcode для создания новой песочницы и инициализируйте массив как приведено ниже:

```
// например
let number = [1,2,3,4,5]
// здесь "3" - это fizz, а "5" - buzz
```

Для нахождения всех “fizz” и “buzz” мы должны пройтись по всему массиву и проверить, какие числа есть “fizz”, а какие “buzz”. Для этого создайте цикл for для прохождения по массиву, который мы инициализировали:

```
for num in number {  
    // здесь находится тело цикла и расчеты  
}
```

После этого, мы можем просто использовать конструкцию “if else” и оператор деления с остатком “%” в swift для определения fizz и buzz.

```
for num in number { // для всех чисел из number  
    if num % 3 == 0 { // если число делится на 3 без остатка  
        print("\\"(num) fizz") // вывести "\"(num) fizz"  
    } else {  
        print(num) // иначе просто вывести это число  
    }  
}
```

Замечательно! Вы можете зайти в консоль отладки в песочнице Xcode и посмотреть результат работы программы. Вы увидите, что “fizz” были отсортированы в вашем массиве.

Для проставления “buzz”, мы воспользуемся той же техникой. Попробуйте это сделать сами перед тем, как прочитать далее - вы можете сверить свои результаты с результатами, которые приведены в данной статье, как только вы закончите.

```
for num in number {  
    if num % 3 == 0 {  
        print("\\"(num) fizz")  
    } else if num % 5 == 0 { // иначе, если число делится на 5 без остатка  
        print("\\"(num) buzz")  
    } else {  
        print(num)  
    }  
}
```

Проверьте выходные данные!

Это довольно просто — вы разделили число на 3, “fizz” и разделили число на 5, “buzz”. Теперь увеличьте количество чисел в массиве

```
let number = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Мы увеличили диапазон чисел с 1-10 до 1-15 для того, чтобы продемонстрировать концепт “fizz buzz”. Так как число 15 кратно 3 и 5, оно должно быть заменено на “fizz buzz”. Попробуйте сами и проверьте результат!

Вот решение:

```

for num in number {
    if num % 3 == 0 && num % 5 == 0 /*если число делится без остатка и на 5,
    и на 3 */
        print("\\"(num) fizz buzz")
    } else if num % 3 == 0 {
        print("\\"(num) fizz")
    } else if num % 5 == 0 {
        print("\\"(num) buzz")
    } else {
        print(num)
    }
}

```

Подождите ... мы еще не закончили! Весь смысл алгоритма заключается в том, чтобы правильно организовать время выполнения. Представьте, если диапазон увеличивается с 1-15 до 1-100. Компилятор проверит каждое число, делится ли оно на 3 или на 5. После чего он пройдется по числам снова, чтобы определить, делится ли число на 3 и на 5. Код по сути, должен будет пройти через каждое число в массиве дважды - ему следовало бы пройтись по числам, делящимся сначала на 3 и потом на 5. Чтобы ускорить процесс, мы можем просто "сказать" нашему коду проверять делимость сразу на 15.

Конечный вариант нашего кода:

```

for num in number {
    if num % 15 == 0 {
        print("\\"(num) fizz buzz")
    } else if num % 3 == 0 {
        print("\\"(num) fizz")
    } else if num % 5 == 0 {
        print("\\"(num) buzz")
    } else {
        print(num)
    }
}

```

Так же легко, как здесь, вы можете использовать любой язык по вашему выбору и начать с него.

Приятного программирования

Глава 2: Сложность Алгоритмов

Раздел 2.1: Большая-Тета нотация

В отличии от Большой-О нотации, которая описывает только верхнюю границу времени выполнения для некоторого алгоритма, Большая-Тета ограничена узкими рамками; верхней и нижней границами. Узкие рамки более точны, однако их сложнее вычислить.

Нотация Большой-Теты симметрична: $f(x) = \Theta(g(x)) \Leftrightarrow g(x) = \Theta(f(x))$

Для интуитивного понимания, $f(x) = \Theta(g(x))$ означает, что графики от $f(x)$ и $g(x)$ растут с одинаковой скоростью, или что графики “ведут себя” схоже при достаточно больших x .

Полное математическое выражение для Большой-Теты: $\Theta(f(x)) = \{g: N_0 \rightarrow R \text{ и } c_1, c_2, n_0 > 0,$
где $c_1 < \text{abs}(g(n)/f(n)),$ для каждого $n > n_0$ и abs - модуль числа\}

Пример

Если алгоритм для ввода n данных требует $42n^2 + 25n + 4$ операций для завершения, мы говорим, что это $O(n^2)$, но также $O(n^3)$ и $O(n^{100})$. Однако, это $\Theta(n^2)$, а не $\Theta(n^3), \Theta(n^4)$ и т.д. Алгоритм $\Theta(f(n))$ является также $O(f(n))$, но не наоборот! **Формальное математическое определение**

т.е. предел существует и он положительный, тогда

$$f(x) = \Theta(g(x))$$

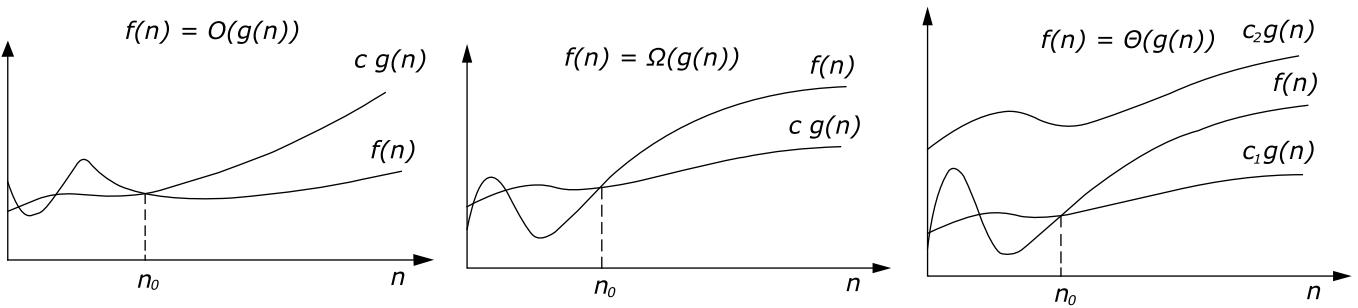
Классы общей сложности

Название	Нотация	n=10	n=100
Постоянная	$\Theta(1)$	1	1
Логарифмическая	$\Theta(\log(n))$	3	7
Линейная	$\Theta(n)$	10	100
Линеарифметическая	$\Theta(n * \log(n))$	30	700
Квадратическая	$\Theta(n^2)$	100	10 000
Экспоненциальная	$\Theta(2^n)$	1 024	1.267650e+30
Факториальная	$\Theta(n!)$	3 628 800	9.332622e+157

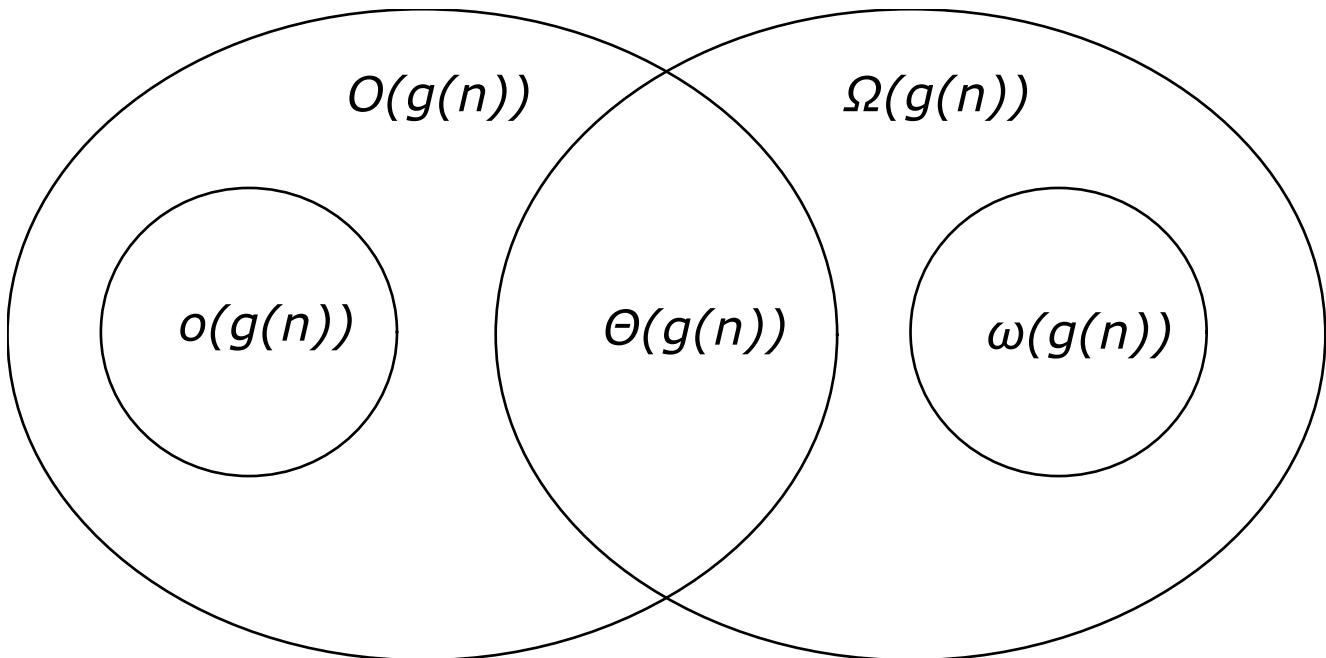
Раздел 2.2: Сравнение асимптотических обозначений

Пусть $f(n)$ и $g(n)$ - две функции, определенные на множестве положительных действительных чисел, c, c_1, c_2, n_0 - положительные действительные константы.

Нотация	$f(n) = O(g(n))$	$f(n) = \Omega(g(n))$	$f(n) = \Theta(g(n))$	$f(n) = o(g(n))$	$f(n) = \omega(g(n))$
Формальное определение	$\exists c > 0, \exists n_0 > 0 : \forall n \geq n_0, 0 \leq f(n) \leq c g(n)$	$\exists c > 0, \exists n_0 > 0 : \forall n \geq n_0, 0 \leq c g(n) \leq f(n)$	$\exists c_1, c_2 > 0, \exists n_0 > 0 : \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$	$\forall c > 0, \exists n_0 > 0 : \forall n \geq n_0, c g(n) < f(n)$	$\forall c > 0, \exists n_0 > 0 : \forall n \geq n_0, 0 < c g(n) < f(n)$
Аналогия между асимптотическим сравнением f, g и действительными числами a, b	$a \leq b$	$a \geq b$	$a = b$	$a < b$	$a > b$
Пример	$7n+10 = O(n^2+n-9)$	$n^3 - 34 = \Omega(10n^2 - 1/2n^2 - 7n = \Theta(n^2))$	$5n^2 = o(n^3)$	$7n^2 = \Omega(n)$	



Асимптотическая нотация может быть представлена на диаграмме Венна следующим образом:



Ссылки

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms

Раздел 2.3: Большая-Омега нотация

Ω -нотация используется для асимптотической нижней границы.

Формальное определение

Пусть $f(n)$ и $g(n)$ - две функции, определённые на множестве положительных действительных чисел. Мы пишем $f(n) = \Omega(g(n))$, если есть положительные константы c и n_0 такие, что:

$$0 \leq c g(n) \leq f(n) \text{ для всех } n \geq n_0.$$

Примечание

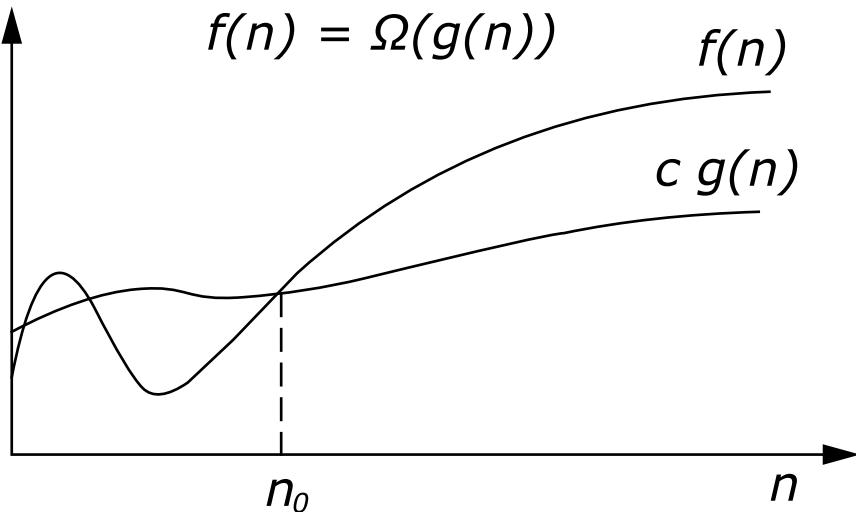
$f(n) = \Omega(g(n))$ означает, что $f(n)$ растет асимптотически не медленнее, чем $g(n)$. Также мы

можем сказать об $\Omega(g(n))$ когда алгоритма анализа недостаточно для утверждения об $\Theta(g(n))$ и/или $O(g(n))$.

Из определения нотации следует теорема:

Для двух любых функций $f(n)$ и $g(n)$ мы имеем $f(n) = \Theta(g(n))$ тогда и только тогда, когда $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$.

Графически Ω -нотация может быть представлена следующим образом:



Например пусть $f(n) = 3n^2 + 5n - 4$. Тогда $f(n) = \Omega(n^2)$. Также правильно будет $f(n) = \Omega(n)$, или даже $f(n) = \Omega(1)$.

Другой пример для решения алгоритма идеального паросочетания: если число вершин нечётное, вывести “Нет идеальных пар”, иначе попробуйте все возможные комбинации.

Нам хотелось бы сказать, что алгоритм требует экспоненциального времени, но вы не можете доказать, что $\Omega(n^2)$ - нижняя граница, используя обычное определение Ω , так как алгоритм работает за линейное время, если n - нечётное. Вместо этого следует определить $f(n) = \Omega(g(n))$ так, что для некоторой константы $c > 0$, $f(n) \geq cg(n)$ для бесконечно большого n . Это даёт хорошее соответствие между верхней и нижней границами: $f(n) = \Omega(g(n))$ тогда и только тогда, когда $f(n) \neq o(g(n))$.

Ссылки

Формальное определение и теорема взяты из книги "Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms".

Глава 3: Обозначение O-большое

Определение

В основе обозначения O-большое лежит математическая форма записи, используемая для сравнения скорости сходимости функций. Пусть $n \rightarrow f(n)$ и $n \rightarrow g(n)$ будут функциями, определёнными над натуральными числами. Скажем, что $f = O(g)$ тогда и только тогда, когда $f(n)/g(n)$ ограничена и n стремится к бесконечности. Другими словами, $f = O(g)$, тогда и только тогда, когда существует константа A , такая, что для всех n , $f(n)/g(n) \leq A$.

На самом деле область применения обозначения O-большое немного шире в математике, но для лучшего понимания я упростил её до необходимых в анализе сложности алгоритма понятий: функции, определённые на натуральных величинах, которые имеют ненулевые значения, и случаи, когда n растёт до бесконечности.

Что это означает?

Рассмотрим случай $f(n) = 100n^2 + 10n + 1$ и $g(n) = n^2$. Совершенно очевидно, что обе эти функции стремятся к бесконечности при n стремящемся к бесконечности. Но иногда недостаточно знать предел, мы также хотим знать и *скорость*, с которой функции стремятся к своему пределу. Такие понятия, как O-большое, помогают сравнивать и классифицировать функции по характеру их изменения.

Давайте выясним, что если $f = O(g)$ по определению, тогда выходит, что $f(n)/g(n) = 100 + 10/n + 1/n^2$. Так как $10/n$ равно 10, когда n равно 1 и уменьшается, также $1/n^2$ равно 1, когда n равно 1 и уменьшается при росте n , тогда получаем $f(n)/g(n) \leq 100 + 10 + 1 = 111$. Это удовлетворяет определению, так как мы нашли границу $f(n)/g(n)$ (111) и поэтому $f = O(g)$ (скажем, что f — это O-большое от n^2).

Это значит, что f стремится к бесконечности примерно с той же скоростью, что и g . Это может показаться странным, потому что мы обнаружили, что f в 111 раз больше, чем g , или другими словами, когда g вырастает на 1, f вырастает максимум на 111. Может показаться, что рост в 111 раз быстрее это не "примерно та же скорость". Действительно, O-большое — не очень точный способ классификации скорости сходимости функций, вот почему в математике мы используем [отношение эквивалентности](#), когда нам нужна точная оценка. Однако для разделения алгоритмов в больших классах скоростей достаточно O-большого. Нам не нужно разделять функции, которые растут в фиксированное количество раз быстрее друг друга, а только те, которые растут бесконечно быстрее друг друга. Например, если взять $h(n) = n^2 * \log(n)$, то мы увидим, что $h(n)/g(n) = \log(n)$, который стремится к бесконечности с увеличением n , поэтому h не является $O(n^2)$, так как h растет бесконечно быстрее, чем n^2 .

Теперь мне нужно сделать замечание: вы могли заметить, что если $f = O(g)$ и $g = O(h)$, то $f = O(h)$. Например, в нашем случае имеем $f = O(n^3)$ и $f = O(n^4)$... В анализе сложности алгоритмов мы часто говорим $f = O(g)$, что означает $f = O(g) \wedge g = O(f)$, это можно понимать как " g — самое маленькое O-большое для f ". В математике мы говорим, что такие функции — это Тета-большие друг друга.

Как это используется?

При сравнении производительности алгоритмов нас интересует количество операций, выполняемых алгоритмом. Это называется *временной сложностью*. В данной модели мы считаем, что каждая основная операция (сложение, умножение, сравнение, присваивание и т.д.) занимает фиксированное количество времени, и подсчитываем количество таких операций. Обычно мы можем выразить это число как функцию размера входного значения, которую мы называем n . И, к сожалению, это число обычно вырастает до бесконечности с n (если это не так, то мы говорим, что алгоритм — $O(1)$). Мы разделяем наши алгоритмы на классы больших скоростей, определяемые O -большое: когда мы говорим об "алгоритме $O(n^2)$ " то имеем в виду, что количество выполняемых им операций, выраженное в функции от n , равно $O(n^2)$. Это говорит о том, что наш алгоритм примерно так же быстр, как и алгоритм, который бы выполнял ряд операций, равных квадрату размера его входного значения, или даже *быстрее* его. Я говорю "даже быстрее его", потому что я использовал O -большое вместо Тета-большое, но обычно люди говорят O -большое, подразумевая Тета-большое.

При подсчете операций обычно учитывается наихудший случай: например, если у нас есть цикл, который может выполняться в большинстве случаев и содержит 5 операций, то количество операций, которое мы считаем, составляет $5n$.

Краткое примечание: быстрый алгоритм — это алгоритм, который выполняет несколько операций, поэтому если количество операций *быстрее* возрастает до бесконечности, то алгоритм работает *медленнее*: $O(n)$ лучше, чем $O(n^2)$.

Нас также иногда интересует *пространственная сложность* нашего алгоритма. Для этого мы рассматриваем количество байт в памяти, занимаемое алгоритмом, как функцию от размера входа, и используем Big-O таким же образом.

Раздел 3.1: Простой цикл

Следующая функция находит максимальный элемент в массиве:

```
//int перед названием ф-ии означает, что ф-ия возвращает значение типа int

int find_max(const int *array, size_t len) {
    //создание функции find_max, которая находит максимальный элемент в массиве
    //array длины len
    int max = INT_MIN; //переменной max задаётся минимально возможное значение,
    //сама переменная будет содержать макс эл-т в конце выполнения ф-ии
    for (size_t i = 0; i < len; i++) {
        //проходим по массиву сравнивая эл-ты с максимальным
        if (max < array[i]) {
            max = array[i];
        }
    }
    return max;
}
```

Входной размер — это размер массива, который я назвал len в коде.

Давайте посчитаем операции.

```
int max = INT_MIN;  
size_t i = 0; // из цикла for
```

Эти два задания выполняются только один раз, следовательно это 2 операции. Зацикленные операции:

```
if (max < array[i])// 1  
    i++; // 2  
max = array[i] // 3
```

Так как в цикле 3 операции, а цикл выполняется n раз, то мы добавляем $3n$ к уже существующим 2 операциям, чтобы получить $3n + 2$. Таким образом, наша функция принимает $3n + 2$ операции для нахождения максимума (его сложность $3n + 2$). Это многочлен, где самым быстрорастущим членом является фактор n , так что это сложность $O(n)$.

Вы, наверное, заметили, что "операция" не очень хорошо определена. Например, я сказал, что **если** ($max < array[i]$) — это одна операция, но в зависимости от архитектуры этот оператор может скомпилироваться, например, в три инструкции: одна операция чтения памяти, одна операция сравнения и одна ветка. Я также считал все операции одинаковыми, даже несмотря на то, что, например, операции с памятью будут медленнее других, и их производительность будет варьироваться в больших пределах, например, для эффектов кэширования. Я также полностью проигнорировал операцию возврата, тот факт, что для функции будет создан фрейм и так далее. В конце концов, для анализа сложности это не имеет значения, потому что, каким бы способом я ни выбрал подсчет операций, он изменит только коэффициент n и константу, так что результат все равно будет $O(n)$. Сложность показывает, как алгоритм меняется с размером входного значения, но это не единственный аспект производительности!

Очевидно, что во втором случае выполняется меньше операций, и поэтому он более эффективен. Как это интерпретируется на Big-O нотацию? Ну, а теперь внутренний цикл тела выполняется $1 + 2 + \dots + n - 1 = n(n-1)/2$ раз. Это *все еще* многочлен от второй степени, и поэтому все еще лишь $O(n^2)$. Мы явно снизили сложность, так как грубо разделили на 2 количество операций, которые выполняем, но мы все еще находимся в том же классе сложности — Big-O. Для того, чтобы понизить сложность до более низкого класса, нам нужно было разделить количество операций на то, что стремится к бесконечности с n .

Раздел 3.2: Вложенный цикл

Следующая функция проверяет, есть ли в массиве дубликаты, выбирая каждый элемент, а затем проводя итерации по всему массиву, чтобы увидеть, есть ли в нем этот элемент

```
_Bool contains_duplicates(const int *array, size_t len) {  
    // _Bool - возвращаемый тип функции  
    for (int i = 0; i < len - 1; i++) {  
        for (int j = 0; j < len; j++) {  
            // цикл по очереди сравнивает каждый элемент кроме последнего
```

```

    // со всеми эл-ами
    if (i != j && array[i] == array[j]) {
        return 1;
        // возвращает 1, если вдруг находят равные элементы из
        // разных ячеек массива
    }
}
return 0;
}

```

Внутренний цикл выполняет на каждой итерации несколько постоянных операций с п. Внешний цикл также выполняет несколько постоянных операций и выполняет внутренний цикл n раз. Сам внешний цикл выполняется n раз. Таким образом, операции внутри внутреннего цикла выполняются n^2 раз, во внешнем — n раз, а присваивание i выполняется один раз. Таким образом, сложность будет примерно равна $an^2 + bn + c$, а так как наивысший член — n^2 , то обозначение O - O(n^2).

Как вы, возможно, заметили, мы можем улучшить алгоритм, не делая одни и те же сравнения несколько раз. Мы можем начать с i + 1 во внутреннем цикле, потому что все элементы до него уже были проверены на наличие элементов массива, в том числе и с индексом i + 1. Это позволяет отказаться от проверки i == j.

```

_Bool faster_contains_duplicates(const int *array, size_t len) {
    // len - длина данного массива
    for (int i = 0; i < len - 1; i++) {
        for (int j = i + 1; j < len; j++) {
            // тот же цикл что и предыдущий, но сравнивает каждый элемент со всеми
            // последующими эл-ами массива
            if (array[i] == array[j]) {
                return 1;
            }
        }
    }
    return 0;
}

```

Раздел 3.3: Типы алгоритмов O(log n)

Допустим, у нас есть сложность размера n. Теперь для каждого шага нашего алгоритма (который нам нужно написать), наша исходная сложность становится вдвое меньше предыдущей ($n/2$).

Таким образом, на каждом шаге наша задача становится вдвое меньше.

Step Problem

- | | |
|---|-----|
| 1 | n/2 |
| 2 | n/4 |

- 3 n/8
- 4 n/16

Когда требуемое пространство сокращено (т.е. задача решена полностью), уменьшение становится невозможным (n становится равным 1) после выхода из условия проверки.

1. Поговорим о k-ом шаге или количестве операций:

$$\text{сложность-задачи} = 1$$

2. Но мы знаем, что на k-ом шаге сложность-задачи должна быть:

$$\text{сложность-задачи} = n/2^k$$

3. От 1 и 2:

$$n/2^k = 1 \text{ или}$$

$$n = 2^k$$

4. Возьмем логарифм от обеих частей

$$\log_2 n = k \log_2 2$$

или

$$k = \log_2 n / \log_2 2$$

5. Воспользуемся формулой $\log_2 m / \log_2 n = \log_2 m$

$$k = \log_2 m$$

$$\text{или проще } k = \log n$$

Теперь мы знаем, что наш алгоритм может работать максимум до $\log n$, следовательно, сложность по времени будет $O(\log n)$

Вот очень простой пример кода, дополняющий приведенный выше текст:

```
for(int i=1; i<=n; i=i*2)
{
    // здесь может быть выполнена какая-либо операция
}
```

Так что теперь, если кто-то спросит, сколько шагов выполнит цикл с $n=256$ (или любой другой алгоритм, уменьшающий размер задачи вдвое), то вы можете очень легко вычислить.

$k = \log_2 256$

$k = \log_2 2^8 (\Rightarrow \log_2 8 = 3)$

$k = 8$

Другой очень хороший пример для подобного случая - **алгоритм двоичного поиска**.

```
int bSearch(int arr[], int size, int item){  
    // Бинарный поиск. item - эл-т который ищем,  
    // size - размер массива(кол-во эл-ов), arr[] - массив с которым работаем  
    int low=0; // левая граница эл-ов в которых ведём поиск  
    int mid; // номер проверяемого эл-та  
    int high=size-1; // правая граница  
    while(low<=high){ // пока не кончается эл-ты  
        mid=low+(high-low)/2; // берется средний элемент между low и high  
        if(arr[mid]==item) // если эл-т массива - искомый  
            return mid; // возвращаем номер(итерационный номер в массиве)  
        else if(arr[mid]<item) // если эл-т массива меньше чем нужно нам  
            low=mid+1; // передвигаем левую границу вправо  
        else high=mid-1; // иначе двигаем правую границу влево  
    }  
    return -1; // Неудачный результат  
}
```

Раздел 3.4: Пример $O(\log n)$

Предисловие

Рассмотрим следующую проблему:

L — отсортированный список, содержащий n подписанных целых чисел (n достаточно большое), например $[-5, -2, -1, 0, 1, 2, 4]$ (здесь n имеет значение 7). Если известно, что L содержит целое число 0, то как найти индекс 0 ?

Примитивный подход

Первое, что приходит на ум, это просто читать каждый индекс до тех пор, пока не будет найден 0. В худшем случае, количество операций равно n , поэтому сложность — $O(n)$.

Это хорошо работает для малых значений n . Но есть ли более эффективный способ ?

Дихотомия

Рассмотрим следующий алгоритм (Python3):

```
a = 0  
b = n-1  
while True:
```

```

h = (a+b)/2 # делим нацело поэтому h будет целым
if L[h] == 0:
    return h
elif L[h] > 0: # если э-т под номером h больше нуля
    b = h #двигаем правую границу влево
elif L[h] < 0: # если э-т под номером h меньше нуля
    a = h #двигаем левую границу вправо

```

а и b — это индексы, между которыми находится 0. Каждый раз, когда мы входим в цикл, мы выбираем индекс между а и b и используем его, чтобы сократить область поиска.

В худшем случае, цикл работает до тех пор, пока а и b не будут равны. Но сколько для этого нужно операций? Не n, потому что каждый раз, когда мы входим в цикл, то делим расстояние между а и b примерно вдвое. Таким образом, сложность представляет собой $O(\log n)$.

Пояснение

Примечание: Когда мы пишем "log" то имеем в виду двоичный логарифм, или базу логарифма 2 (в которой мы напишем " \log_2 "). В качестве $O(\log_2 n) = O(\log n)$ мы будем использовать "log" вместо " \log_2 ".

Назовем x количеством операций: мы знаем, что $1 = n / (2^x)$.

Тогда $2^x = n$, значит $x = \log n$.

Вывод

Когда сталкиваешься с последовательным делением (на два или на любое число), помни, что сложность логарифмическая.

Глава 4: Деревья

Раздел 4.1: Типичное представление анарного дерева

Обычно мы представляем анарное дерево (один узел с потенциально неограниченным числом дочерних элементов) в виде двоичного дерева (один узел ровно с парой детей). «Следующий» ребенок считается родным братом. Обратите внимание, что если дерево является двоичным, то такое представление создает дополнительные узлы.

Затем мы повторяем итерацию над братьями и сестрами и возвращаемся к детям. Поскольку большинство деревьев относительно мелкие, имеем много детей только на нескольких уровнях иерархии, что приводит к оптимизации кода. Обратите внимание, что человеческие родословные являются исключением (много уровней предков и только несколько детей на уровень).

При необходимости можно сохранить обратные указатели, чтобы по дереву можно было подняться наверх. Такие деревья сложнее хранить.

Обратите внимание, что нормально иметь одну функцию для вызова у корня и рекурсивную функцию с дополнительными параметрами, в данном случае глубиной дерева.

```
struct node // структура узла дерева
{
    struct node *next; // указатель на структуру типа node,
                      // являющейся братом для данного узла
    struct node *child; // указатель на структуру типа node,
                      // являющейся ребенком для данного узла
    std::string data; // строка данных
}

void printtree_r(struct node *node, int depth) // функция вывода дерева,
// которая принимает параметры указатель на узел и уровень отступа
{
    int i;
    while(node) // цикл действующий пока узел не пустой
    {
        if(node->child) // если ребенок не пустой
        {
            for(i=0;i<depth*3;i++) // отступ длиной в уровень узла,
            // умноженный на 3
            printf(" ");
            printf("{\n");
            printtree_r(node->child,depth+1); // рекурсивный вызов
            // функции вывода дерева с ребенком в качестве аргумента
            for(i=0;i<depth;depth+1); // отступ длиной на один
            // уровень больше
            printf(" ");
            printf("{\n");
        }
    }
}
```

```

    }
    for(i=0;i<depth*3;i++) // отступ длиной в уровень узла
        printf(" ");
    printf("%s\n", node->data.c_str()); // вывод значения узла

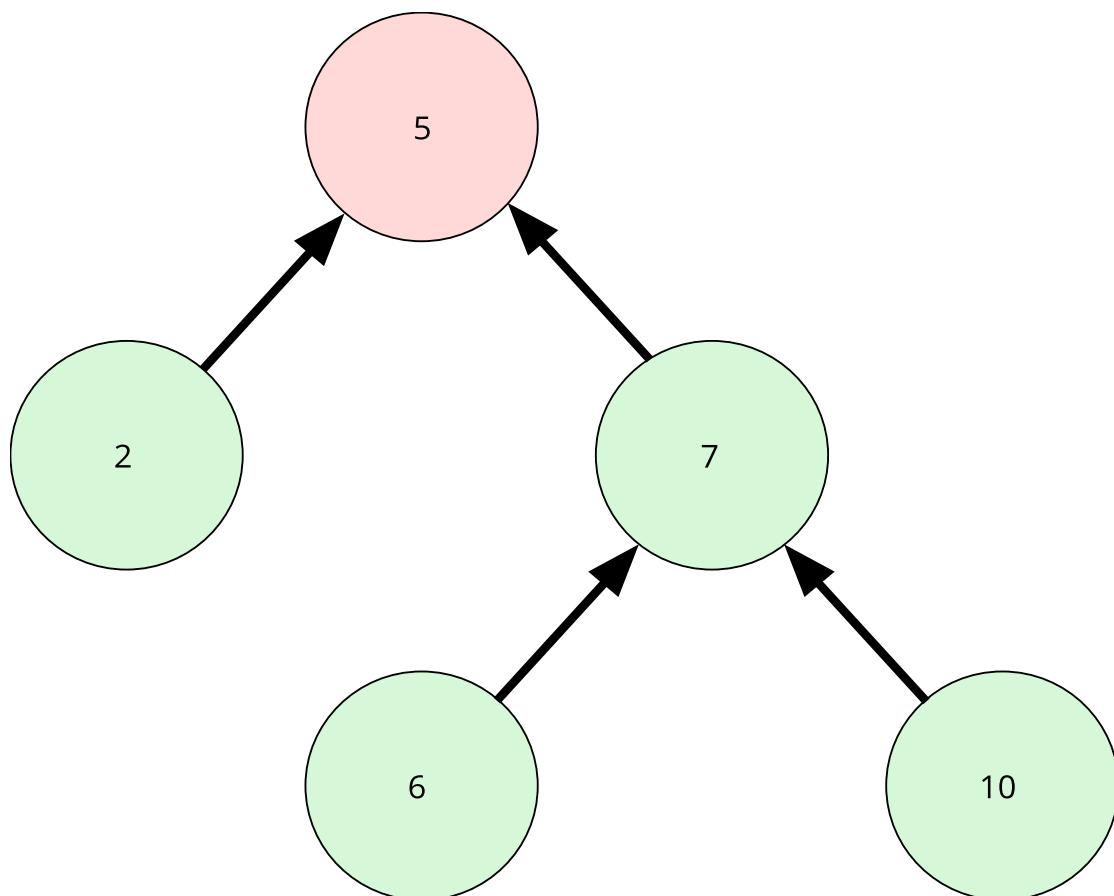
    node = node->next; // переход к брату
}

void printtree(node *root) // Выход дерева начиная с корня
{
    printtree_r(root, 0); // Вызов функции вывода с произвольной
    // позиции с корнем, как узлом и нулевым начальным отступом
}

```

Раздел 4.2: Введение

Деревья являются подтипов более общей структуры данных узловых графов.



Чтобы быть деревом, граф должен удовлетворять двум требованиям:

- **Он является ациклическим.** Он не содержит циклов (или "зацикливания").
- **Он подключен.** Для любой заданной вершины на графе доступна каждая вершина. Все узлы доступны через один путь в графе.

Древовидная структура данных довольно распространена в компьютерной науке. Деревья используются для моделирования многих различных алгоритмических структур данных, таких как обычные бинарные деревья, красно-черные деревья, В-деревья, АВ-деревья, куча, 23-деревья и многие другие.

Принято называть дерево "корневым":

Выбирается **один** узел, который будет называться "корнем".

Нарисовав "корень" на вершине,

создаем нижние слои для каждой ячейки на графике в зависимости от их расстояния до корня
- чем больше расстояние, тем ниже листья (пример выше)

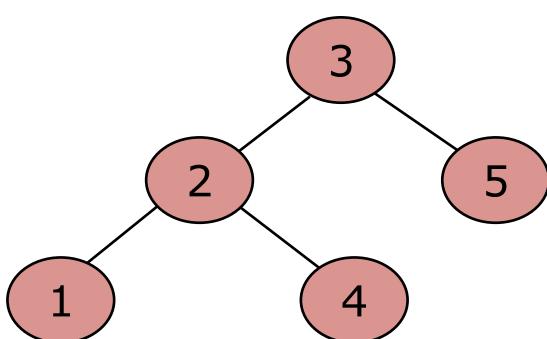
Общий символ для деревьев: Т

Раздел 4.3: Проверка, одинаковы два двоичных дерева или нет

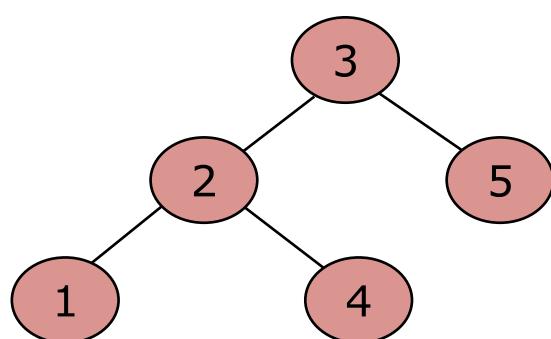
Например, если входные данные такие:

Пример: 1

a)



b)

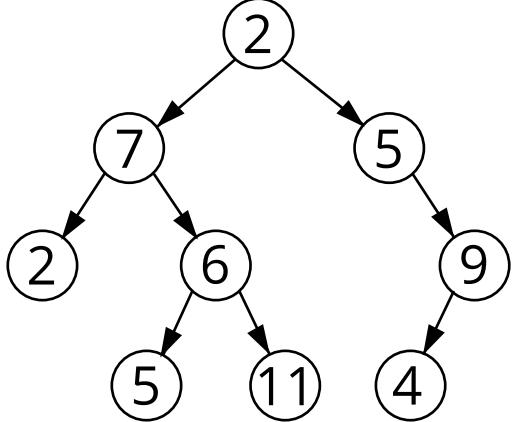


Результатом должна быть истина.

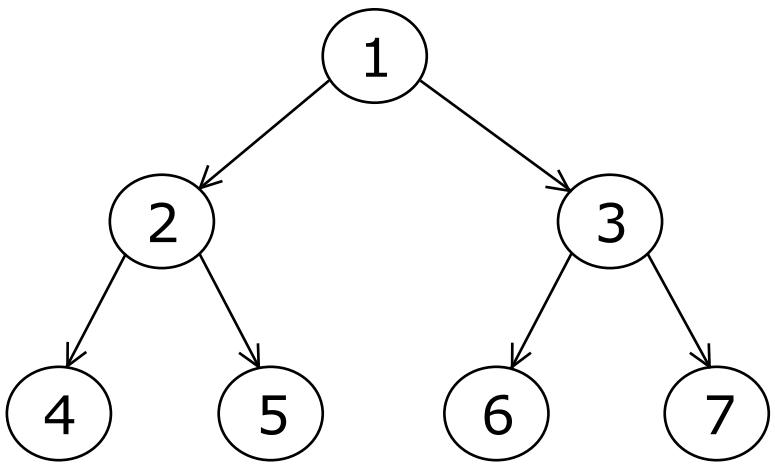
Пример: 2

Если входные данные такие:

a)



b)



Результатом должна быть ложь.

Псевдокод для этого случая:

```
bookean sameTree(node root1, node root2){ // функция сравнивающая два дерева,  
    // принимает в себя указатели на узлы деревьев  
    if(root1 == NULL && root2 == NULL) // проверяет пустые ли два узла  
        return true; // если оба узла пустые, то эти 2 поддерева равны  
        // и мы возвращаем истину  
    if(root1 == NULL || root2 == NULL) // если только один из узлов пустой  
        return false; //то поддеревья не равны и мы возвращаем ложь  
    // если оба поддерева не пустые то сравниваем их значение, и равны ли левое и  
    // правое поддерево  
    if(root1->data == root2->data  
        && sameTree(root1->left,root2->left) //рекурсивно сравниваем поддеревья  
        && sameTree(root1->right, root2->right))  
        return true;  
    }
```

Глава 5: Поиск в двоичных деревьях

Двоичное дерево - это дерево, в котором каждый узел имеет максимум два дочерних элемента. Двоичное дерево поиска (BST) - это двоичное дерево, элементы которого расположены в специальном порядке. В каждом BST все значения (т.е. ключи) в левом поддереве меньше по значению, чем в правом

Раздел 5.1: Дерево поиска двоичных файлов - Вставка (Python)

Это простая реализация вставки Binary Search Tree с использованием Python.

Пример: [анимация](#)

Следуя за фрагментом кода, каждое изображение показывает визуализацию выполнения кода, что облегчает представление того, как это происходит.

```
class Node:      # класс узла поддерева
    def __init__(self, val): # Поля класса
        self.l_child = None # указатель на узел левого поддерева
        self.r_child = None # указатель на узел правого поддерева
        self.data = val # значение
```

Структуры Объекты

Глобальная структура

Node

Класс Node
скрытые атрибуты

функция
__init__
__init__(self, val)

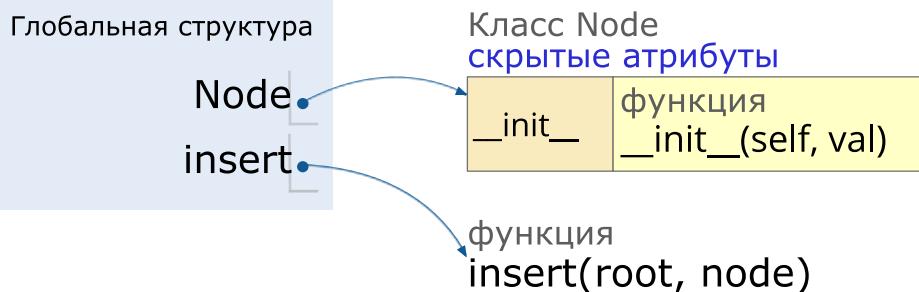
```
def insert(root, node): # добавление элемента в дерево
    # принимает в себя корень и новый узел
    if root is None: # пустой ли корень
        root = node # если корень пуст, то присваиваем ему введенный узел
    else:
        if root.data > node.data: # больше ли элемент данного узла
            if root.l_child is None: # пустой ли левый ребенок
                root.l_child = node # присваиваем ему узел
            else: #если левый узел не пустой
                insert(root.l_child, node) # повторяем функцию
                # для левого поддерева
        else:
```

```

if root.r_child is None: # пустой ли правый ребенок
    root.r_childe = node # присваиваем ему узел
else: #если правый узел не пустой
    insert(root.r_child, node) # повторяем функцию
        # для правого поддерева

```

Структуры Объекты



```

def in_order_print(root): # инфиксный вывод дерева
    if not root: # пустой ли корень
        return
    in_order_print(root.l_child) # рекурсивно вызываем функцию
        # с левым ребенком
    print root.data # выводим данные
    in_order_print(root.r_child) # рекурсивно вызываем функцию с правым ребенком

```

Структуры Объекты



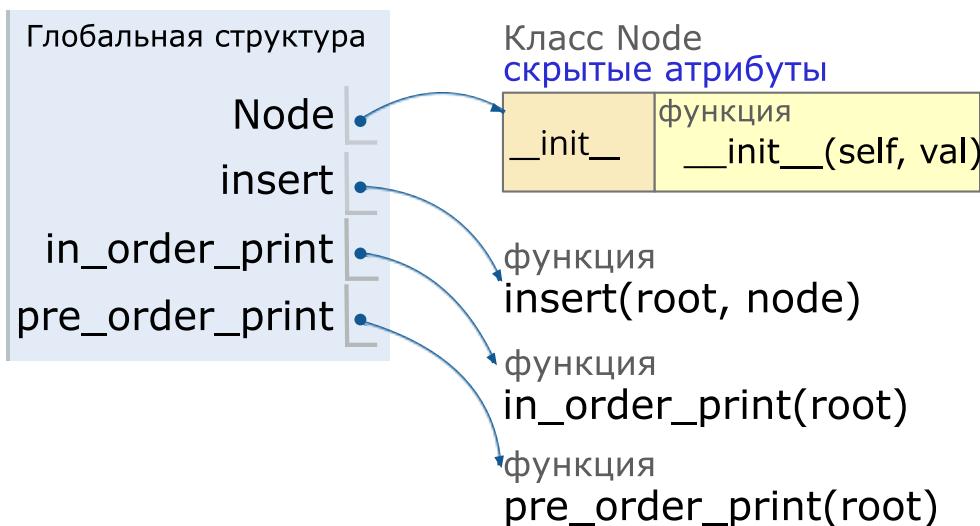
```

def pre_order_print(root): # префиксный вывод дерева
    if not root: # пустой ли корень
        return
    print root.data # выводим данные
    pre_order_print(root.l_child) # рекурсивно вызываем функцию
        # с левым ребенком
    pre_order_print(root.r_child) # рекурсивно вызываем функцию
        # с правым ребенком

```

Структуры

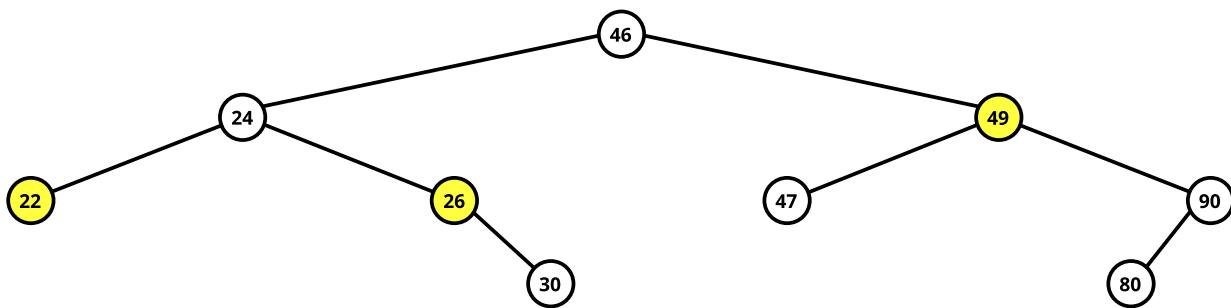
Объекты



Раздел 5.2: Дерево двоичного поиска - удаление (C++)

Перед началом удаления, я хочу рассказать, что является деревом двоичного поиска (BST). Каждый узел в BST может иметь максимум две вершины (левая и правая дочерние). Левое поддерево вершины имеет ключ, меньший или равный ключу родительского узла. Правое поддерево узла имеет ключ больше, чем ключ родительского узла.

Удаление узла в дереве при сохранении **свойства** самого **дерева двоичного поиска**.



При удалении узла необходимо учитывать три случая:

- Случай 1: Удаляемый узел - это узел листа (узел со значением 22).
- Случай 2: У удаляемого узла есть один дочерний элемент (узел со значением 26).
- Случай 3: У удаляемого узла есть оба дочерних узла (узел со значением 49).

Пояснения по случаям:

1. Когда удаляемый узел является узлом листа, просто удалите узел и передайте nullptr его родительскому узлу.

2. Когда удаляемый узел имеет только один дочерний, скопируйте дочернее значение в значение узла и удалите ребёнка (**преобразовано в случай 1**)
3. Если удаляемый узел имеет двух детей, то минимальное количество детей из его правого поддерева может быть скопировано в узел, а затем минимальное значение может быть удалено из правого поддерева узла (**преобразовано в случай 2**)

Примечание: минимум в правом поддереве может иметь максимум одного ребенка, и этот правый ребенок, если у него есть левый ребенок, либо не минимальное значение, либо не следует свойству BST.

Структура узла в дереве и код для удаления:

```

struct node // структура узла бинарного дерева
{
    int data; // данные
    node *left, *right; // указатели на левое и правое поддерево
};

node* delete_node(node *root, int data) // удаление данных из бинарного дерева
{
    if(root == nullptr) return root; // если дерево пустое, то ничего не
    // делаем
    // если данные меньше значения узла, повторяем удаление для левого поддерева
    else if(data < root->data) root->left = delete_node(root->left, data);
    // если данные больше значения узла, повторяем удаление для правого поддерева
    else if(data > root->data) root->right = delete_node(root->right, data);
    // если они равны
    else
    {
        if(root->left == nullptr && root->right == nullptr) //1 случай
        {
            free(root);
            root = nullptr;
        }
        else if(root->left == nullptr) //2 случай
        {
            node* temp = root;
            root = root->right;
            free(temp);
        }
        else if(root->right == nullptr) //2 случай
        {
            node* temp = root;
            root = root->left;
            free(temp);
        }
        else //3 случай
        {
            node* temp = root->right;
            while(temp->left != nullptr) temp = temp->left;
            root->data = temp->data;
        }
    }
}

```

```

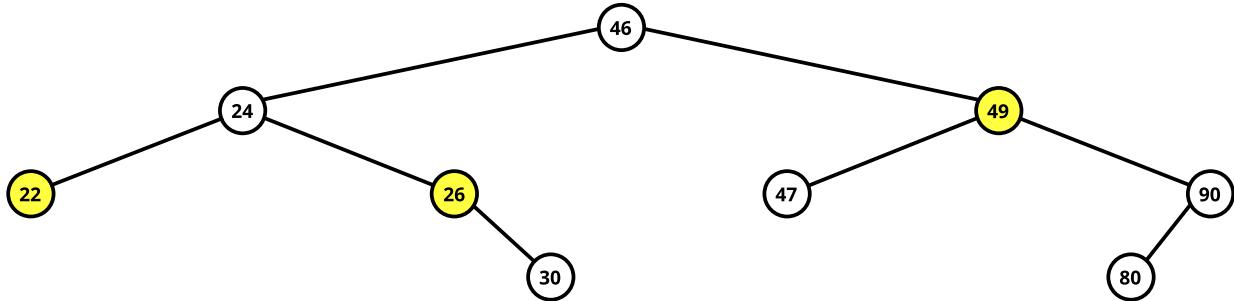
        root->right = delete_node(root->right, temp->data);
    }
}
return root;
}

```

Временная сложность приведенного кода - $O(h)$, где h - это высота дерева.

Раздел 5.3: Самый низкий предок в BST

Рассмотрим BST:



Самый низкий родоначальник 22 и 26 - 24

Самый низкий родоначальник 26 и 49 - 46.

Самый низкий родоначальник 22 и 24 - 24

Свойство поиска двоичного дерева может быть использовано для нахождения узлов самого низкого предка

Псевдокод:

```

lowestCommonAncestor(root,node1,node2) //нахождение общего предка для 2 узлов
{
    if(root == NULL) return NULL;
    else if(node1->data == root->data || node2->data == root->data) //один из узлов
        //корень
        return root; //возвращаем корень
    else if((node1->data <= root->data && node2->data > root->data) //узлы по разные
        || (node2->data <= root->data && node1->data > root->data)){ //стороны
        //от корня
        return root; //возвращаем корень
    }
    else if(root->data > max(node1->data,node2->data)){ //узлы меньше корня
        //ищем общего предка для этих узлов в левом поддереве
        return lowestCommonAncestor(root->left, node1, node2);
    }
    else{ //узлы больше корня
    }
}

```

```

//ищем общего предка для этих узлов в левом поддереве
    return lowestCommonAncestor(root->right, node1, node2);
}
}

```

Раздел 5.4: Поиск в двоичном дереве - Python

```

class Node(object): # узел двоичного дерева
    def __init__(self, val):
        self.l_child = None # Указатель на левое поддерево
        self.r_child = None # Указатель на правое поддерево
        self.val = val # Значение

class BinarySearchTree(object): # Функция поиска в бинарном дереве
    def insert(self, root, node):
        if root is None: # если корень пустой, то корень равен узлу
            return node
        if root.val < node.val # если значение узла больше узла корня
            root.r_child = self.insert(root.r_child, node) # ищем в правом поддереве
        else: # если меньше
            root.l_child = self.insert(root.l_child, node) # ищем в левом поддереве
        return root

    def in_order_place(self, root): # вывод в инфиксной форме
        if not root: # пустой ли корень
            return None
        else:
            self.in_order_place(root.l_child) # рекурсивно вызываем функцию
                                              # с левым ребенком
            print root.val # выводим данные
            self.in_order_place(root.r_child) # рекурсивно вызываем функцию
                                              # с правым ребенком

    def pre_order_place(self, root): # вывод в префиксной форме
        if not root: # пустой ли корень
            return None
        else:
            print root.val # выводим данные
            self.pre_order_place(root.l_child) # рекурсивно вызываем функцию
                                              # с левым ребенком
            self.pre_order_place(root.r_child) # рекурсивно вызываем функцию
                                              # с правым ребенком

    def post_order_place(self, root): # вывод в постфиксной форме
        if not root: # пустой ли корень
            return None
        else:
            print root.val # выводим данные

```

```
self.pre_order_place(root.l_child) # рекурсивно вызываем функцию
                                    # с левым ребенком
self.pre_order_place(root.r_child) # рекурсивно вызываем функцию
                                    # с правым ребенком
```

"Создать другой узел и вставить в него данные"

```
r = Node(3)                      # создания корня со значением 3
node = BinarySearchTree()          # создание бинарного дерева
nodeList = [1, 8, 5, 12, 14, 6, 15, 7, 16, 8] # массив вводимых значений
for nd in nodeList:               # заполнение дерева
    node.insert(r, Node(nd))
print "-----In order -----"      # вывод дерева
print (node.in_order_place(r))
print "-----Pre order -----"
print (node.pre_order_place(r))
print "-----Post order -----"
print (node.post_order_place(r))
```

Глава 6: Проверка, является дерево BST или нет

Раздел 6.1: Алгоритм проверки, является ли данное двоичное дерево BST

Двоичное дерево является BST, если оно удовлетворяет одному из следующих условий:

1. Оно пусто.
2. У него нет поддерева
3. В дереве для каждого узла x все ключи (если таковые имеются) в левом поддереве должны быть меньше, чем ключ (x), а все ключи (любые) в правом поддереве должны быть больше, чем ключ(x).

Так что прямолинейный рекурсивный алгоритм был бы такой:

```
is_BST(root): # является ли дерево бинарным поиском
    if root == NULL: # если дерево пустое то оно BST
        return true
    # Проверка значения левого поддерева
    if root->left != NULL;
        max_key_in_left = find_max_key(root->left) # максимальное значение
                                                        # в левом поддереве
        if max_key_in_left > root->key: # если оно больше корня, то оно не BST
            return false

    # Проверка правого поддерева
    if root->right != NULL:
        min_key_in_right = find_min_key(root->right) # максимальное значение
                                                        # в правом поддереве
        if min_key_in_right < root->key: # если оно меньше корня, то оно не BST
            return false
    return is_BST(root->left) && is_BST(root->right) # возвращаем значение
                                                        # для левого и правого поддерева
```

Вышеуказанный рекурсивный алгоритм корректен, но неэффективен, т.к. он обходит каждый узел несколько раз.

Другой подход к минимизации многократных посещений каждого узла заключается в запоминании минимально и максимально возможных значений ключей в поддереве, которые мы посещаем. Пусть минимально возможное значение любого ключа будет K_MIN, а максимальное значение будет K_MAX. Когда мы начинаем с корня дерева, диапазон значений в дереве будет [K_MIN, K_MAX]. Пусть ключ от корня узла будет x , тогда диапазон значений в левом поддереве равен [K_MIN, x], а диапазон значений в правом поддереве равен (x , K_MAX]. Мы будем использовать эту идею для разработки более эффективного алгоритма.

```

is_BST(root, min, max): # является ли дерево бинарным в заданном диапазоне
if root == NULL # если пустое, то является
    return true

# узел выходит за диапазон?
if root->key < min || root->key >max # если нет, то возвращаем ложь
    return false
# проверка, являются ли левое и правое поддерево BST на диапазоне
return if_BST(root->left,min,root->key-1) && is_BST(root->right,root->key+1,max)

```

Первоначально он будет называться так:

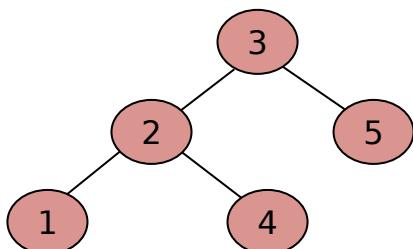
```
is_BST(my_tree_root, KEY_MIN, KEY_MAX)
```

Другой подход будет заключаться в обходе двоичного дерева. Если обход по порядку производит отсортированную последовательность ключей, то данное дерево является BST. Чтобы проверить, отсортирована ли последовательность ключей, запомните значение ранее посещенного узла и сравните его с текущим узлом.

Раздел 6.2: Имеет ли данное на входе дерево свойства бинарного дерева поиска или нет

Например

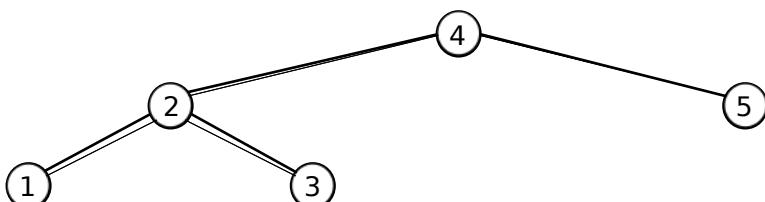
Если входные данные таковы:



Результат должен быть ложным:

Так как 4 в левом поддереве больше, чем значение корня(3)

Если входные данные таковы:



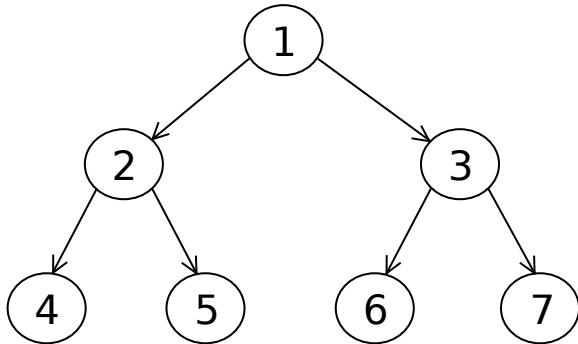
Результат должен быть истинным

Глава 7: Обходы бинарного дерева

Посещение узла бинарного дерева в определенном порядке называется обходом.

Раздел 7.1: Реализация обхода в порядке уровней

Например, если данное дерево таково:



Обход в порядке уровней будет

1 2 3 4 5 6 7

Печать данных узла производится по уровням

Код:

```
#include <iostream>
#include <queue>
#include <malloc.h>
using namespace std;
struct node{
    int data;
    node *left;
    node *right;
};
void levelOrder(struct node *root){

    if(root == NULL) return;

    queue<node *> Q;
    Q.push(root);

    while(!Q.empty()){
        struct node* curr = Q.front();
        cout << curr->data << " ";
        if(curr->left != NULL) Q.push(curr-> left);
        if(curr->right != NULL) Q.push(curr-> right);
        Q.pop();
    }
}
```

```

    }
}

struct node* newNode(int data)
{
    struct node* node = (struct node*)
                           malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int main(){

    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    printf("Level Order traversal of binary tree is \n");
    levelOrder(root);

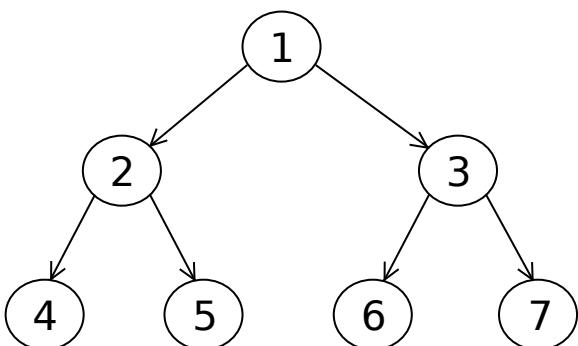
    return 0;
}

```

Такая структура данных, как очередь, используется для решения выше поставленной задачи.

Раздел 7.2:Прямой, симметричный и обратный обход бинарного дерева

Рассмотрим бинарное дерево:



Прямым обходом(корня) называется обход, при котором сначала происходит обход узла, затем левого и правого поддерева этого узла.

Таким образом, прямой обход показанного выше дерева будет:

1 2 4 5 3 6 7

Симметричным обходом(корня) называется обход, при котором сначала происходит обход левого поддерева узла, затем самого узла , а после правого поддерева узла.

Таким образом, симметричный обход показанного выше дерева будет:

4 2 5 1 6 3 7

Обратным обходом(корня) называется обход, при котором сначала происходит обход левого поддерева узла, затем правого поддерева узла, а после самого узла.

Таким образом, обратный порядок обхода дерева, показанного выше, будет:

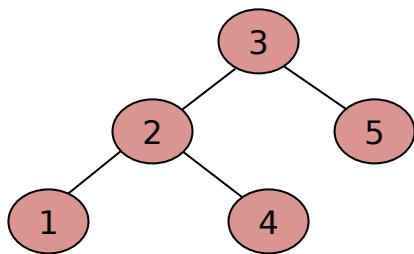
4 5 2 6 7 3 1

Глава 8: Самый нижний общий предок бинарного дерева

Самый нижний общий предок между двумя узлами $n1$ и $n2$ определяется как самый нижний узел в дереве, который имеет оба узла $n1$ и $n2$ в качестве потомков

Раздел 8.1: Нахождение самого нижнего общего предка

Рассмотрим дерево:



Самым низким общим предком узлов со значениями 1 и 4 является 2

Самым низким общим предком узлов со значениями 1 и 5 является 3

Самым низким общим предком узлов со значениями 2 и 4 является 4

Самым низким общим предком узлов со значениями 1 и 2 является 2

Глава 9: Граф

Граф - это совокупность точек и линий, соединяющих некоторое (возможно пустое) их подмножество. Точки графа называются вершинами графа, узлами или просто точками. Также линии, соединяющие вершины графа, называются ребрами графа, дугами или линиями.

Граф G может быть определен как пара (V, E) , где V - множество вершин, а E - множество ребер между вершинами $E \subseteq \{(u, v) | u, v \in V\}$.

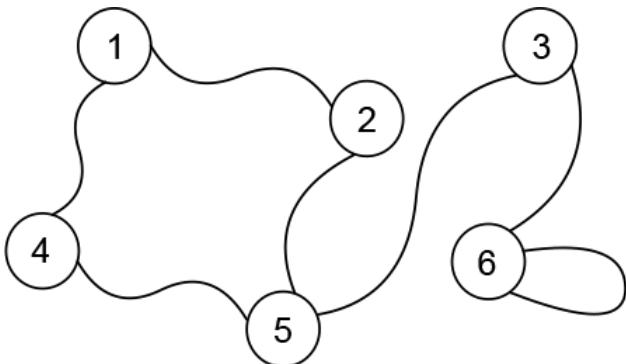
Раздел 9.1: Хранение графов (матрицы смежности)

Для хранения графа обычно используется два способа:

- Матрица смежности
- Список смежности

Матрица смежности - это квадратная матрица, используемая для конечного представления графа. Элементы матрицы показывают, какие пары вершин в графе являются смежными, а какие - нет.

Смежный - это либо граничащий с чем-то, либо находящийся рядом с чем-то. Например, ваши соседи смежны с вами. В теории графов, если мы можем попасть в **узел B** из **узла A**, мы говорим, что **узел B** смежный с **узлом A**. Теперь мы узнаем о том, как хранить такие узлы, которые примыкают к каким-то из них с помощью матрицы смежности. Это значит, что мы изобразим узлы, которые делят ребро между собой. Здесь матрица подразумевается как двумерный массив.

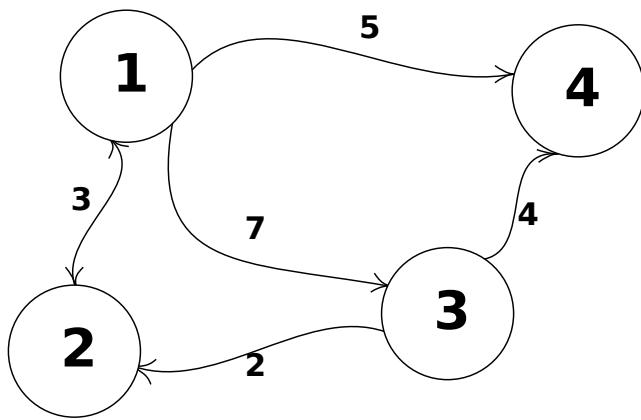


Узел	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	1	1	0
3	0	0	0	0	1	1
4	1	1	0	0	1	0
5	0	1	1	1	0	0
6	0	0	1	0	0	1

Здесь вы можете увидеть таблицу рядом с графиком. Это наша матрица смежности. Здесь $\text{Matrix}[i][j] = 1$ показывает, что существует ребро между i и j . Если ребра нет, мы просто помещаем 0 в $\text{Matrix}[i][j]$.

Эти ребра могут быть нагруженными, что можно представить как расстояние между двумя городами. Тогда мы поместим вместо 1 какое-то значение в $\text{Matrix}[i][j]$.

Граф, описанный выше, называется двунаправленным или ненаправленным. Это значит, что если мы можем попасть из **узла 2** в **узел 1**, то мы также можем попасть из **узла 1** в **узел 2**. Если бы граф был Направленным, то на одной стороне графа был знак-стрелка. И даже такой граф можно представить в виде матрицы смежности.



Узел	1	2	3	4
1	∞	3	7	5
2	3	∞	∞	∞
3	∞	3	∞	4
4	∞	∞	∞	∞

Мы представляем узлы, которые не делят ребро на бесконечность. Стоит отметить, что если граф является ненаправленным, то матрица становится симметричной.

Псевдокод для создания матрицы:

```
Procedure AdjacencyMatrix(N): // N показывает количество вершин
Matrix[N] [N]
for i from 1 to N
    for j from 1 to N
        Take input -> Matrix[i] [j]
    endfor
endfor
```

Мы также можем заполнить матрицу, пользуясь обычным методом:

```
Procedure AdjacencyMatrix(N, E): // N -> количество вершин
Matrix[N] [E] // E -> количество рёбер
for i from 1 to E
    input -> n1, n2, cost
    Matrix[n1] [n2]=cost
    Matrix[n2] [n1]=cost
endfor
```

Для направленных графов мы можем удалить строку, где $Matrix[n2][n1] = cost$.

Недостатки использования матрицы смежности:

Память - это огромная проблема. Вне зависимости от того, как много ребер, нам всегда нужна матрица размеров $N \times N$, где N - это количество узлов. Для 10000 узлов размер матрицы будет $4 \times 10000 \times 10000$, что составляет примерно 381 Мбайт. Это огромная трата памяти, если мы рассматриваем графы, которые имеют несколько ребер.

Предположим, что нам надо выяснить, в какой узел мы можем попасть из узла u . Нам необходимо проверить каждый ряд с u , что занимает много времени.

Единственное преимущество в том, что мы можем легко найти связь между узлами $u-v$ и их стоимость, используя матрицу смежности.

Java-код, который реализует использованный выше псевдокод:

```
import java.util.Scanner;

public class Represent_Graph_Adjacency_Matrix
{
    private final int vertices;
    private int[][] adjacency_matrix;

    public Represent_Graph_Adjacency_Matrix(int v)
    {
        vertices = v;
        adjacency_matrix = new int[vertices + 1][vertices + 1];
    }

    public void makeEdge(int to, int from, int edge)
    {
        try
        {
            adjacency_matrix[to][from] = edge;
        }
        catch (ArrayIndexOutOfBoundsException index)
        {
            System.out.println("The vertices does not exists");
        }
    }

    public int getEdge(int to, int from)
    {
        try
        {
            return adjacency_matrix[to][from];
        }
        catch (ArrayIndexOutOfBoundsException index)
        {
            System.out.println("The vertices does not exists");
        }
        return -1;
    }

    public static void main(String args[])
    {
        int v, e, count = 1, to = 0, from = 0;
        Scanner sc = new Scanner(System.in);
        Represent_Graph_Adjacency_Matrix graph;
        try
        {
            System.out.println("Enter the number of vertices: ");
            v = sc.nextInt();
            System.out.println("Enter the number of edges: ");
            e = sc.nextInt();

            graph = new Represent_Graph_Adjacency_Matrix(v);
```

```

System.out.println("Enter the edges: <to> <from>");
while (count <= e)
{
    to = sc.nextInt();
    from = sc.nextInt();

    graph.makeEdge(to, from, 1);
    count++;
}

System.out.println("The adjacency matrix for the given graph is: ");
System.out.print(" ");
for (int i = 1; i <= v; i++)
    System.out.print(i + " ");
System.out.println();
for (int i = 1; i <= v; i++)
{
    System.out.print(i + " ");
    for (int j = 1; j <= v; j++)
        System.out.print(graph.getEdge(i, j) + " ");
    System.out.println();
}
}
catch (Exception E)
{
    System.out.println("Somthing went wrong");
}

sc.close();
}
}

```

Запускаем код: сохраняем файл и компилируем, используя javac
Represent_Graph_Adjacency_Matrix.java

Пример:

```

$ java Represent_Graph_Adjacency_Matrix
Enter the number of vertices:
4
Enter the number of edges:
6
Enter the edges:
1 1
3 4
2 3
1 4
2 4
1 2
The adjacency matrix for the given graph is:
1 2 3 4
1 1 1 0 1

```

2	0	0	1	1
3	0	0	0	1
4	0	0	0	0

Раздел 9.2: Введение в теорию графов

Теория графов - это изучение тех графов, которые представляют собой математические структуры, используемые для моделирования попарных отношений между объектами.

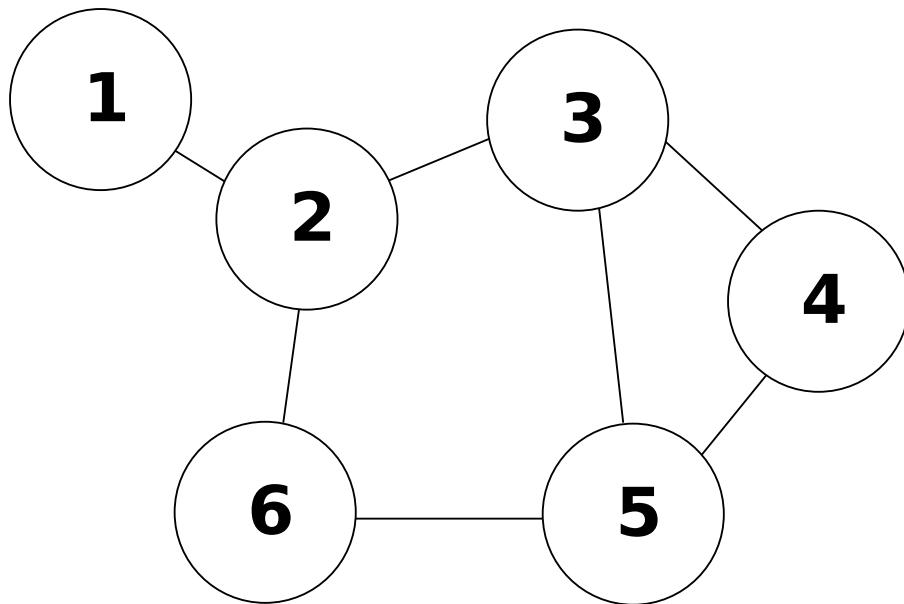
Вы знали, что почти все задачи на планете Земля можно конвертировать в задачи про дороги и города и решить? Теория графов была изобретена много лет назад до изобретения компьютера. Леонард Эйлер написал статью о семи мостах Кёнигсберга, который считают первым документом по теории графов. С тех пор люди начали понимать, что если мы можем конвертировать любую задачу в задачу про города и дороги, то можем легко решить ее с помощью теории графов.

Теория графов имеет множество применений. Одно из самых распространенных - поиск кратчайшего расстояния между двумя городами. Мы все знаем, что для того, чтобы добраться до вашего ПК, этой веб-странице пришлось пройти множество маршрутизаторов с сервера. Теория графов помогает выяснить, какие маршрутизаторы необходимо пересечь. Какую улицу необходимо было бомбардировать во время войны, чтобы отсоединить столицу от других городов, можно было выяснить с помощью теории графов.

Давайте для начала выучим несколько базовых определений в теории графов.

Граф:

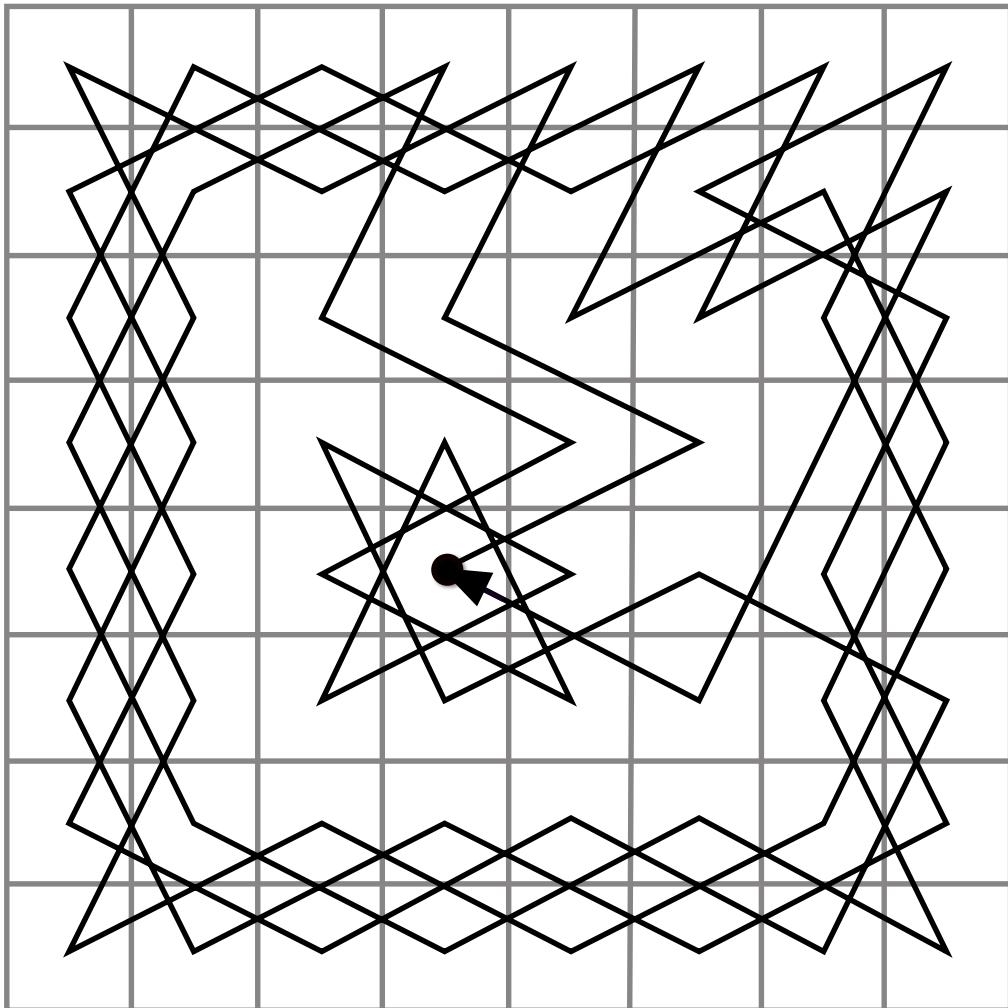
Пусть у нас есть 6 городов, которые мы отметим цифрами 1, 2, 3, 4, 5, 6. Теперь соединим города, между которыми есть дорога.



Это простой граф, где некоторые города изображены с дорогами, которые их соединяют. В теории графов мы называем каждый из этих городов **узлом** или **вершиной**, а дороги - **ребрами**. Проще говоря, граф - это совокупность узлов и ребер.

Узел может представлять собой очень многое. В некоторых графах узлы представляют собой

города, некоторые - аэропорты, а какие-то - квадрат на шахматной доске. **Ребро** представляет собой отношение между узлами. Такое отношение может быть временем, чтобы добраться от одного аэропорта до другого, движения коня от одного квадрата до других и т.д.



Путь Коня на шахматной доске

Проще говоря, **узел** представляет собой любой объект, а **ребро** представляет отношение между двумя объектами.

Смежный узел:

Если узел **A** и узел **B** делят между собой ребро, то **B** считается смежным с **A**. Другими словами, если два узла напрямую связаны, они называются смежными узлами. Один узел может иметь несколько смежных узлов.

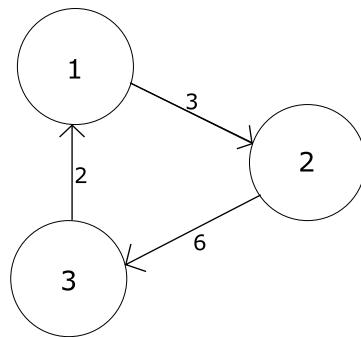
Ориентированный и неориентированный граф:

Если в ориентированных графах ребра имеют указатели направления только на одной стороне, то это означает, что ребра *однонаправлены*. Если же ребра неориентированных графов имеют указатели направления с обеих сторон, то это означает, что они *двунаправлены*. Обычно неориентированные графы представлены без каких-либо знаков по обе стороны от ребер.

Давайте предположим, что у нас идёт вечеринка в самом разгаре. Люди на вечеринке - это узлы, и когда они пожимают друг другу руки, то образуется связь(грань) между двумя людьми. Тогда этот граф является неориентированным, потому что любой человек **A** пожимает руку человеку **B** тогда и только тогда, когда **B** также пожимает руку **A**. Однако, если ребру между человеком **A** и **B** соответствует такая связь, как восхищение, то граф ориентированный, так как восхищение не обязательно является взаимным. Первый тип графа называется *неориентированным графом*, а его ребра называются *неориентированными ребрами*, а последний тип графа называется *ориентированным графом*, а его ребра - *ориентированными ребрами*.

Взвешенный и невзвешенный граф:

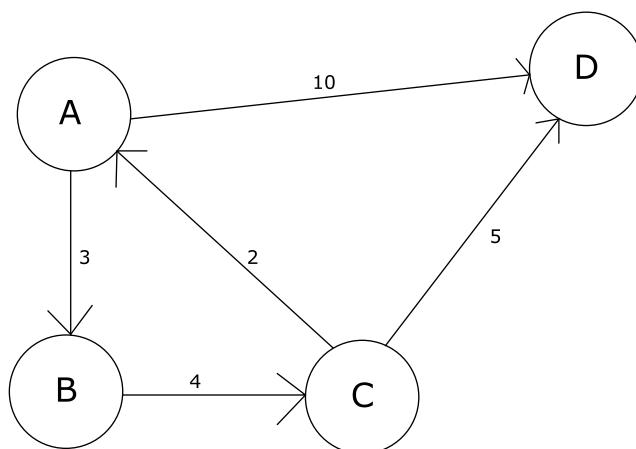
Взвешенный граф - это граф, в котором число (вес) присваивается каждому ребру. Такие веса могут обозначать, например, стоимость, длину или производительность, в зависимости от поставленной задачи.



Невзвешенный граф - обратный случай (ребра в нём не имеют веса). Предположим, что вес всех ребер одинаков (например 1).

Путь:

Путь представляет собой способ перехода от одного узла к другому. Он состоит из последовательности ребер. Между двумя узлами может быть несколько путей.



В приведенном выше примере есть два пути от **A** до **D**. **A->B, B-> C, C-> D** - это один путь. Стоимость этого пути **$3 + 4 + 2 = 9$** . Опять же, есть другой путь **A->D**. Стоимость этого пути - **10**. Путь с наименьшей стоимостью называется *кратчайшим*.

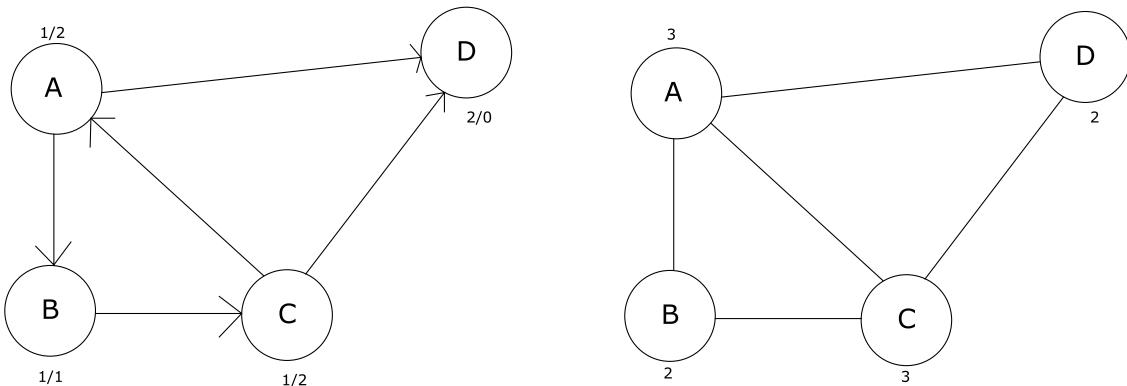
Степень:

Степень вершины - это число ребер, которые связаны с ней. Если есть ребро, которое соединяется с вершиной на обоих концах (цикл), то оно считается два раза.

В ориентированных графах узлы имеют два типа степеней:

- Входящая степень: количество ребер, которые указывают на узел.
- Исходящая степень: количество ребер, которые, исходя из данного узла, указывают на другие

Для неориентированного графа, они просто называются степенью.



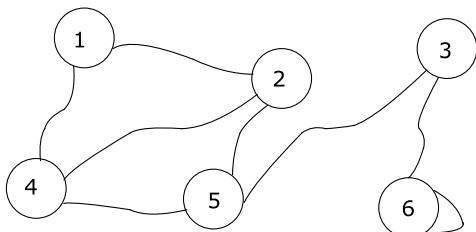
Некоторые алгоритмы, связанные с теoriей графов:

- Алгоритм Беллмана – Форда
- Алгоритм Дейкстры
- Алгоритм Форда – Фулкерсона
- Алгоритм Крускала
- Алгоритм ближайшего соседа
- Алгоритм Прима
- Поиск в глубину
- Поиск в ширину

Раздел 9.3: Хранение графов (Список смежности)

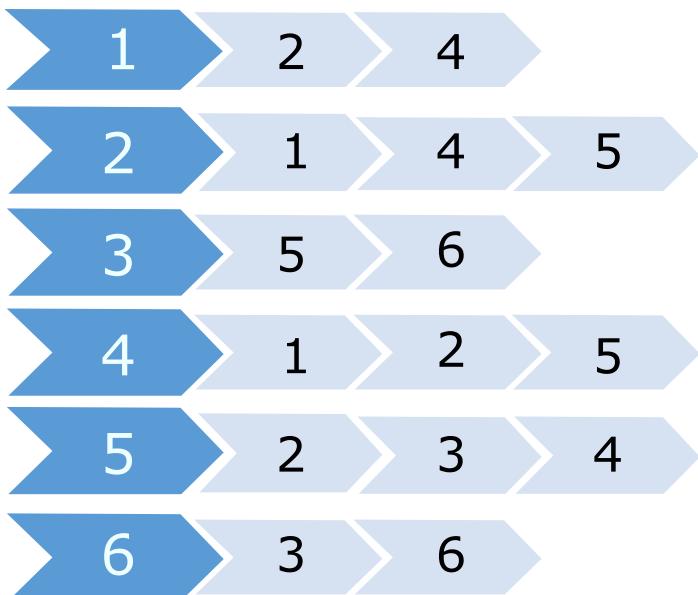
Список смежности представляет собой набор неупорядоченных списков, используемых для представления конечного графа. Каждый список описывает набор соседних вершин в графе. Для хранения графов используется меньше памяти.

Давайте посмотрим на граф и его матрицу смежности:



Узел	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	1	1	0
3	0	0	0	0	1	1
4	1	1	0	0	1	0
5	0	1	1	1	0	0
6	0	0	1	0	0	1

Теперь мы создадим список, используя эти значения.



Это называется списком смежности. Он показывает взаимосвязь между узлами графа. Мы можем хранить эту информацию, используя двумерный массив. Но это будет стоить нам столько же памяти, сколько стоит матрица смежности. Вместо этого мы будем динамически выделять память для её хранения.

Многие языки программирования поддерживают **векторы (Vector)** или **списки (List)**, которые мы можем использовать для хранения списка смежности. Для этого нам не нужно указывать размер **списка**. Нам нужно только указать максимальное количество узлов.

Соответствующий псевдокод будет таким:

```

Procedure Adjacency-List(maxN, E):
edge[maxN] = Vector()
for i from 1 to E
    input -> x, y
    edge[x].push(y)
    edge[y].push(x)
end for
Return edge

```

Так как это неориентированный граф, то если есть ребро от x до y , то соответственно есть и ребро от y до x . Если бы это был ориентированный граф, мы бы опустили второе условие. Для взвешенных графов нам также нужно хранить стоимость (вес). Мы создадим **вектор** или **список** с именем **cost** [] для её хранения. Псевдокод:

```
Procedure Adjacency-List(maxN, E) :  
edge[maxN] = Vector()  
cost[maxN] = Vector()  
for i from 1 to E  
    input -> x, y, w  
    edge[x].push(y)  
    cost[x].push(w)  
end for  
Return edge, cost
```

Из этого мы можем легко узнать общее количество связанных узлов, а также что это именно за узлы.

Это занимает меньше времени, чем Матрица Смежности. Но если бы нам нужно было выяснить, есть ли грань между u и v , то было бы проще, если бы мы сохранили матрицу смежности.

Раздел 9.4: Топологическая сортировка

Топологическое упорядочение или топологическая сортировка упорядочивает вершины в ориентированном ациклическом графе в линию (то есть в список) так, чтобы все направленные ребра шли слева направо. Такое упорядочение не может существовать, если график содержит направленный цикл, потому что нет такого способа, которым вы можете продолжать идти прямо по линии и вернуться туда, откуда вы начали.

Формально в графике $G = (V, E)$ линейное упорядочение всех его вершин таково, что если G содержит ребро $(u, v) \in E$ из вершины u в вершину v , то u предшествует v в упорядочении.

Важно отметить, что каждый направленный ациклический график (DAG) имеет *по крайней мере один* топологический вид.

Известны алгоритмы для построения топологического упорядочения любого DAG за линейное время, одним из примеров является:

1. Вызовите `depth_first_search (G)` для вычисления времени окончания $v.f$ для каждой вершины v
2. По завершении каждой вершины, вставьте её в начало связанного списка
3. Получаем отсортированный связанный список вершин.

Топологическая сортировка может быть выполнена за $O(V + E)$, поскольку алгоритм поиска в глубину занимает $O(V + E)$ времени и требуется $\Omega(1)$ (постоянное время) для вставки каждой из $|V|$ вершины в начале связанного списка.

Многие приложения используют ориентированные ациклические графы для указания приоритетов среди событий. Мы используем топологическую сортировку, чтобы получить порядок обработки каждой вершины перед любым из ее наследников.

Вершины в графике могут представлять задачи, которые должны быть выполнены, а ребра могут представлять ограничения, в соответствии с которыми одна задача должна быть выполнена

перед другой; топологическое упорядочение является допустимой последовательностью для выполнения задач, набор которых описан в V.

Пример проблемы и ее решение

Пусть вершина v описывает задачу Task(hours_to_complete: int), т.е. Task(4) описывает задачу, выполнение которой занимает 4 часа, а грань e описывает время отката Cooldown(hours: int), так что Cooldown(3) описывает время следующего отката после выполненной задачи.

Пусть наш граф называется dag (так как это ориентированный ациклический граф), и пусть он содержит 5 вершин:

```
A <-dag.add_vertex(Task(4));
B <-dag.add_vertex(Task(5));
C <-dag.add_vertex(Task(3));
D <-dag.add_vertex(Task(2));
E <-dag.add_vertex(Task(7));
```

где мы соединяем вершины с направленными ребрами так, чтобы граф был ациклическим,

```
// A ---> C -----+
// |         |         |
// v         v         v
// B ---> D ---> E

dag.add_edge(A, B, Cooldown(2));
dag.add_edge(A, C, Cooldown(2));
dag.add_edge(B, D, Cooldown(1));
dag.add_edge(C, D, Cooldown(1));
dag.add_edge(C, E, Cooldown(1));
dag.add_edge(D, E, Cooldown(3));
```

и следовательно, мы имеем 3 возможных топологических упорядочивания между A и E,

1. A → B → D → E
2. A → C → D → E
3. A → C → E

Раздел 9.5. Обнаружение цикла в ориентированном графе

Цикл в ориентированном графе существует, если во время DFS обнаружено заднее ребро. Заднее ребро - это грань, идущая от узла к себе или к одному из предков в дереве DFS. Для неориентированного графа мы получаем лес DFS, поэтому нужно перебрать все вершины графа, чтобы найти несвязанные деревья DFS.

Реализация на C ++:

```
#include <iostream>
#include <list>

using namespace std;

#define NUM_V 4
```

```

bool helper(list<int> *graph, int u, bool* visited, bool* recStack)
{
    visited[u]=true;
    recStack[u]=true;
    list<int>::iterator i;
    for(i = graph[u].begin();i!=graph[u].end();++i)
    {
        if(recStack[*i])
            return true;
        else if(*i==u)
            return true;
        else if(!visited[*i])
        {
            if(helper(graph, *i, visited, recStack))
                return true;
        }
    }
    recStack[u]=false;
    return false;
}

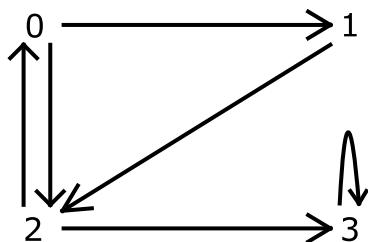
bool isCyclic(list<int> *graph, int V)
{
    bool visited[V];
    bool recStack[V];

    for(int i = 0;i<V;i++)
        visited[i]=false, recStack[i]=false;
    for(int u = 0; u < V; u++)
    {
        if(visited[u]==false)
        {
            if(helper(graph, u, visited, recStack))
                return true;
        }
    }
    return false;
}

int main()
{
    list<int>* graph = new list<int>[NUM_V];
    graph[0].push_back(1);
    graph[0].push_back(2);
    graph[1].push_back(2);
    graph[2].push_back(0);
    graph[2].push_back(3);
    graph[3].push_back(3);
    bool res = isCyclic(graph, NUM_V);
    cout<<res<<endl;
}

```

Результат: как показано ниже, на графе есть три задних ребра. Один между вершиной 0 и 2; между вершинами 0, 1 и 2; и вершина 3. Временная сложность поиска составляет $O(V + E)$, где V - количество вершин, а E - количество ребер.



Раздел 9.6: Алгоритм Торупа

Алгоритм Торупа для кратчайшего пути с одним источником для неориентированного графа имеет временную сложность $O(m)$, меньшую, чем у Дейкстры.

Основные идеи следующие. (Извините, я еще не пытался это реализовать, поэтому могу пропустить некоторые мелкие детали. А оригинал документа платный, так что я попытался восстановить его из других источников, ссылающихся на него. Пожалуйста, удалите этот комментарий, если вы можете проверить.)

- Есть способы найти связующее дерево за время $O(m)$ (здесь это не описано). Вам нужно «вырастить» оствное дерево от самого короткого ребра до самого длинного, и это будет лес с несколькими связными компонентами до полного выращивания.
- Выберите целое число b ($b \geq 2$) и рассмотрите только оставные леса с ограничением длины b^k . Объедините компоненты, которые абсолютно одинаковы, но с разными k , и назовите минимальным k уровнем компоненту. Затем логически превратите компоненты в дерево. и является родителем v , если и является наименьшей компонентой, отличной от v , которая полностью содержит v . Корень - это весь граф, а листья - это отдельные вершины в исходном графе (с уровнем отрицательной бесконечности). Дерево все еще имеет только $O(n)$ узлов.
- Сохраняйте расстояние от каждой компоненты до источника (как в алгоритме Дейкстры). Расстояние компоненты с несколькими вершинами - это минимальное расстояние ее нерасширенных дочерних элементов. Установите расстояние от исходной вершины на 0 и обновите предков соответственно.
- Рассмотрим расстояния в базе b . При первом посещении узла уровня k поместите его дочерние элементы в сегменты, совместно используемые всеми узлами уровня k (как при сортировке сегментов, заменяя путаницу в алгоритме Дейкстры) цифрой k и выше ее расстояния. Каждый раз, посещая узел, учитывайте только его первые b сегментов, посещайте и удалите каждый из них, обновляйте расстояние текущего узла и связывайте текущий узел со своим родителем, используя новое расстояние, и ждите следующего посещения для следующего ведра (корзины).
- Когда лист посещается, текущее расстояние является конечным расстоянием вершины. Увеличьте все ребра из него в исходном графе и обновите расстояния соответственно.
- Посетите корневой узел (всего графа) несколько раз, пока не будет достигнут пункт назначения.

Это основано на том факте, что между двумя соединенными компонентами оставного леса с ограничением длины l нет ребра с длиной меньше l , поэтому, начиная с расстояния x , вы можете сосредоточиться только на одной подключенной компоненте, пока не достигнете расстояния

$x + 1$. Вы посетите некоторые вершины до того, как все вершины с более коротким расстоянием будут посещены, но это не имеет значения, потому что известно, что из этих вершин не будет более короткого пути сюда. Другие части работают так же, как корзинная сортировка / поразрядная сортировка MSD, и, конечно, для этого требуется связующее дерево $O(m)$.

Глава 10: Обходы графов

Раздел 10.1: Функция обхода поиска в глубину

Функция принимает аргумент текущего индекса узла, список смежности (хранищийся в векторе векторов в этом примере), и вектор логического типа, чтобы отслеживать, какой узел был посещен.

```
// Depth-first search - поиск в глубину
void dfs(int node, vector<vector<int>>* graph, vector<bool>* visited) {
    // проверить, был ли узел посещен ранее
    if((*visited)[node])
        return;

    /* установить как посещенный, чтобы избежать посещения одного и того же
    узла дважды*/
    (*visited)[node] = true;

    // выполнить какое-то действие здесь
    cout << node;

    // перейти к соседним узлам в глубину рекурсивно
    for(int i = 0; i < (*graph)[node].size(); ++i)
        // запускаем рекурсивно для каждого узла
        dfs((*graph)[node][i], graph, visited);
}
```

Глава 11: Алгоритм Дейкстры

Раздел 11.1: Алгоритм Дейкстры по поиску кратчайшего пути

Перед тем как продолжить, рекомендуется иметь краткое представление о матрице смежности и поиске в ширину.

Алгоритм Дейкстры известен как алгоритм поиска кратчайшего пути из одного узла-источника во все остальные. Он используется для нахождения кратчайшего пути между узлами в графе, который может представлять собой, например, сети дорог. Он был сформулирован Эдсгером В. Дейкстрой в 1956 и опубликован три года спустя.

Мы можем найти кратчайший путь, используя алгоритм поиска в ширину. Этот алгоритм работает хорошо, но проблема заключается в том, что он предполагает, что стоимость прохождения каждого пути одинакова, а это означает, что вес каждого ребра должен быть одинаковым. Алгоритм Дейкстры помогает нам найти кратчайший путь там, где стоимость прохождения каждого пути неодинакова.

Для начала мы посмотрим, как модифицировать поиск в ширину, чтобы написать алгоритм Дейкстры, затем добавим очередь с приоритетом, чтобы окончательно сделать из него алгоритм Дейкстры.

Пусть расстояние до каждого узла от вершины-источника храниться в массиве $d[]$. Так, например, $d[3]$ означает, что необходимо $d[3]$ времени, чтобы достигнуть **узла 3** из вершины-источника. Если мы не знаем расстояние, мы будем хранить бесконечность в $d[3]$. Также, пусть $\text{cost}[u][v]$ обозначает стоимость пути $u-v$. Это означает, что требуется $\text{cost}[u][v]$, чтобы пройти из узла u в узел v .



Нам необходимо понять, что такое Релаксация Ребра. Пусть, от вашего дома, который является **вершиной-источником**, до места **A** на дорогу требуется 10 минут. И до места **B** требуется 25 минут. Мы имеем,

$$\begin{aligned}d[A] &= 10 \\d[B] &= 25\end{aligned}$$

Теперь скажем, что необходимо 7 минут, чтобы добраться от места **A** до места **B**, это означает:

$$\text{cost}[A][B] = 7$$

Затем мы можем попасть в место **B** из **вершины-источника**, перейдя в место **A** из **вершины-источника**, а затем из места **A**, в место **B**, что займет $10 + 7 = 17$ минут, вместо 25 минут. Итак,

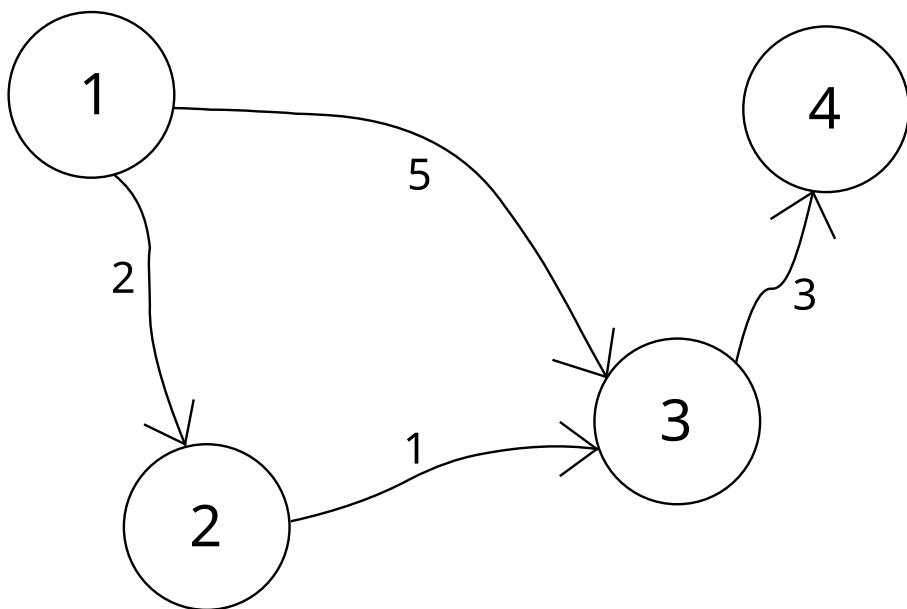
$$d[A] + \text{cost}[A][B] < d[B]$$

Тогда мы присваиваем,

$$d[B] = d[A] + \text{cost}[A][B]$$

Это называется **релаксацией**. Если мы идем из узла **u** в узел **v** и $d[u] + \text{cost}[u][v] < d[v]$, то мы присваиваем $d[v] = d[u] + \text{cost}[u][v]$.

При поиске в ширину нам не нужно было посещать какой-либо узел дважды. Мы только проверяли, посещен узел или нет. Если он не был посещен, мы помещали узел в очередь, отмечали его как посещенный и инкрементировали расстояние на 1. В алгоритме Дейкстры, мы можем поместить узел в очередь и вместо того, чтобы отметить его как посещенный, мы релаксируем или обновляем новое ребро. Посмотрим на один пример:



Давайте предположим, что **Узел 1** является **Источником**. Тогда,

$$\begin{aligned} d[1] &= 0 \\ d[2] &= d[3] = d[4] = \text{infinity (or a large value)} \end{aligned}$$

Мы присваиваем $d[2]$, $d[3]$ и $d[4]$ значение бесконечности, потому что мы еще не знаем расстояние. И расстояние от **вершины-источника** до нее же самой, конечно же, равно 0. Теперь мы идем в другие узлы из **вершины-источника** и, если мы можем присвоить им новое значение, то мы помещаем их в очередь. Например, мы проходим **ребро 1-2**. Так как $d[1] + 2 < d[2]$, мы имеем $d[2] = 2$. Схожим образом мы пройдем через **ребро 1-3**, которое делает $d[3] = 5$.

Мы четко видим, что 5 не является кратчайшим расстоянием, которое мы можем пересечь,

чтобы попасть в **узел 3**. Поэтому обход узла только один раз, как в поиске в ширину, здесь не работает. Если мы перейдем от **узла 2** к **узлу 3**, используя **ребро 2-3**, мы можем присвоить значение $d[3] = d[2] + 1 = 3$. Таким образом, мы можем заметить, что один узел может быть обновлен много раз. Сколько раз вы спрашиваете? Максимальное число обновлений узла равно полуостепени захода узла.

Давайте посмотрим псевдокод для посещения любого узла несколько раз. Мы просто модифицируем поиск в ширину:

```
// модифицированный поиск в ширину
procedure BFSmodified(G, source):
    // объект очереди
    Q = queue()
    // массив дистанций между source и другими узлами
    distance[] = infinity
    // добавление текущей вершины
    Q.enqueue(source)
    // инициализация дистанции для выбранного узла
    distance[source] = 0
    // пока очередь не пустая
    while Q is not empty
        // выбираем первый элемент из очереди
        u <- Q.pop()
        for all edges from u to v in G.adjacentEdges(v) do
            /* если текущая стоимость прохода до проверяемого узла
            меньше зафиксированной, то обновляем зафиксированное значение */
            if distance[u] + cost[u][v] < distance[v]
                distance[v] = distance[u] + cost[u][v]
            end if
        end for
    end while
    return distance
```

Этот код может быть использован для нахождения кратчайшего пути до всех узлов из вершины-источника. Сложность этого кода не так хороша. Вот почему,

При поиске в ширину, когда мы идем из **узла 1** во все другие узлы, мы следуем методу первым пришел, первым обслужен. Например, мы пошли в **узел 3** из вершины-источника до обработки **узла 2**. Если мы идем в **узел 3** из **вершины-источника**, мы присваиваем значение **узлу 4**, соответствующее $5 + 3 = 8$. Когда мы снова обновляем значение **узла 3** из **узла 2**, нам нужно снова присвоить значение **узлу 4**, соответствующее $3 + 3 = 6$! Итак, **узел 4** обновлен дважды.

Дейкстры предложил вместо метода *первым пришел, первым обслужен*, сначала обновлять ближайшие узлы, тогда потребуется меньше обновлений. Если мы обработали **узел 2** до этого, тогда **узел 3** должен был быть обновлен до и после обновления **узла 4** соответственно, тогда мы бы с легкостью получили кратчайшее расстояние! Идея состоит в том, чтобы выбирать из очереди узел, который является ближайшим к **вершине-источнику**. Итак мы будем тут использовать (очередь с приоритетом), чтобы при удалении элемента из очереди, она выдавала нам ближайший узел и к **вершине-источнику**. Как она сделает это? Она будет проверять значение $d[u]$ внутри.

Давайте посмотрим на псевдокод:

```
procedure dijkstra(G, source):
    // объект очереди с приоритетами
    Q = priority_queue()
    // массив дистанций между source и другими узлами
    distance[] = infinity
    // добавление текущей вершины

    Q.enqueue(source)
    // инициализация дистанции для выбранного узла
    distance[source] = 0
    // пока очередь не пустая
    while Q is not empty
        // заносим ближайшую вершину к вершине-источнику в и
        u <- nodes in Q with minimum distance[]
        // удаляем ближайшую вершину из очереди
        remove u from the Q
        for all edges from u to v in G.adjacentEdges(v) do
            /* если текущая стоимость прохода до проверяемого узла меньше
            зафиксированной, то обновляем зафиксированное значение */
            if distance[u] + cost[u][v] < distance[v]
                distance[v] = distance[u] + cost[u][v]
                // заносим примыкающий узел
                Q.enqueue(v)
        end if
    end for
end while
Return distance
```

Псевдокод возвращает расстояние до всех других узлов от **вершины-источника**. Если мы хотим узнать расстояние до одного узла **v**, мы можем просто вернуть значение, когда **v** удаляется из очереди.

А теперь узнаем, работает ли алгоритм Дейкстры при наличии отрицательного ребра? Если в графе присутствует отрицательный цикл, то получим бесконечный цикл, так как отрицательный цикл будет продолжать уменьшать значение каждый раз. Даже если в графе присутствует лишь отрицательное ребро, алгоритм Дейкстры не будет работать, за исключением, если мы вернемся после того, как цель установлена. Но тогда это не будет алгоритмом Дейкстры. Нам понадобится алгоритм Беллмана-Форда для обработки отрицательных ребер/циклов.

Сложность:

Сложность поиска в ширину равна $O(\log(V+E))$, где **V** - это число узлов и **E** — это число ребер. Для алгоритма Дейкстры сложность схожая, но сортировка очереди с приоритетом требует $O(\log V)$. Поэтому общая сложность: $O(V \log(V) + E)$.

Ниже расположен Java пример, использующий матрицу смежности, для реализации алгоритма Дейкстры по поиску кратчайшего пути:

```

import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
    // количество вершин <-> размерность матрицы
    static final int V=9;

    int minDistance(int dist[], Boolean sptSet[])
    {
        /* ищем минимальное расстояние до непомеченных вершин */
        // ставим планку для сравниваемого значения и индекса
        int min = Integer.MAX_VALUE, min_index=-1;

        // проход по строкам матрицы
        for (int v = 0; v < V; v++)
            // если ребро не учтено и минимально на текущий момент
            if (sptSet[v] == false && dist[v] <= min)
            {
                // устанавливаем новую планку
                min = dist[v];
                min_index = v;
            }
        // возвращаем индекс самого малого веса ребра
        return min_index;
    }

    void printSolution(int dist[], int n)
    {
        /* вывод результата в виде списка */
        System.out.println("Vertex Distance from Source");

        // первый столбец - индекс вершины; второй - минимальной дистанции
        for (int i = 0; i < V; i++)
            System.out.println(i+" \t\t "+dist[i]);
    }

    void dijkstra(int graph[][][], int src)
    {
        // инициализация булева массива по количеству вершин
        Boolean sptSet[] = new Boolean[V];

        for (int i = 0; i < V; i++)
        {
            // присваиваем каждому ребру максимальный вес и ставим флаг
            dist[i] = Integer.MAX_VALUE;
            // "непосещение" текущего ребра
            sptSet[i] = false;
        }
    }
}

```

```

}

// стартовое состояние одной из вершин
dist[src] = 0;

for (int count = 0; count < V-1; count++)
{
    int u = minDistance(dist, sptSet);

    // помечаем вершину с минимальным расстоянием
    sptSet[u] = true;

    /* ищем меньшие варианты обхода,
    обновляем ранее установленное расстояние */
    for (int v = 0; v < V; v++)
        if (!sptSet[v] && graph[u][v] !=0 &&
            dist[u] != Integer.MAX_VALUE &&
            dist[u]+graph[u][v] < dist[v])
            dist[v] = dist[u] + graph[u][v];
    }

    // выводим результат
    printSolution(dist, V);
}

public static void main (String[] args)
{
    // инициализируем двумерный массив симметричной матрицы смежности
    int graph[][] = new int[][]{{0, 4, 0, 0, 0, 0, 0, 8, 0},
                                {4, 0, 8, 0, 0, 0, 0, 11, 0},
                                {0, 8, 0, 7, 0, 4, 0, 0, 2},
                                {0, 0, 7, 0, 9, 14, 0, 0, 0},
                                {0, 0, 0, 9, 0, 10, 0, 0, 0},
                                {0, 0, 4, 14, 10, 0, 2, 0, 0},
                                {0, 0, 0, 0, 0, 2, 0, 1, 6},
                                {8, 11, 0, 0, 0, 0, 1, 0, 7},
                                {0, 0, 2, 0, 0, 0, 6, 7, 0}
    };

    ShortestPath t = new ShortestPath();

    t.dijkstra(graph, 0);
}
}

```

Ожидаемый вывод программы:

Vertex	Distance from Source
0	0
1	4

2	12
3	19
4	21
5	11
6	9
7	8
8	14

Глава 12: Поиск пути A*

Раздел 12.1: Введение в A*

A*(A star) - это поисковый алгоритм, который используется для поиска пути от одного узла к другому. Его можно сравнить с такими алгоритмами, как поиск в ширину, алгоритм Дейкстры, поиск в глубину или поиск по первому наилучшему совпадению. Алгоритм A* широко используется в поиске графов из-за более высокой эффективности и точности там, где предварительная обработка данных графа неуместна.

A* - разновидность поиска по первому наилучшему совпадению (Best-first search), в котором функция оценки f определяется особым образом.

$f(n) = g(n) + h(n)$ имеет минимальную стоимость с момента, когда исходный узел, предназначенный для необходимых целей, вынужден проходить через узел n .

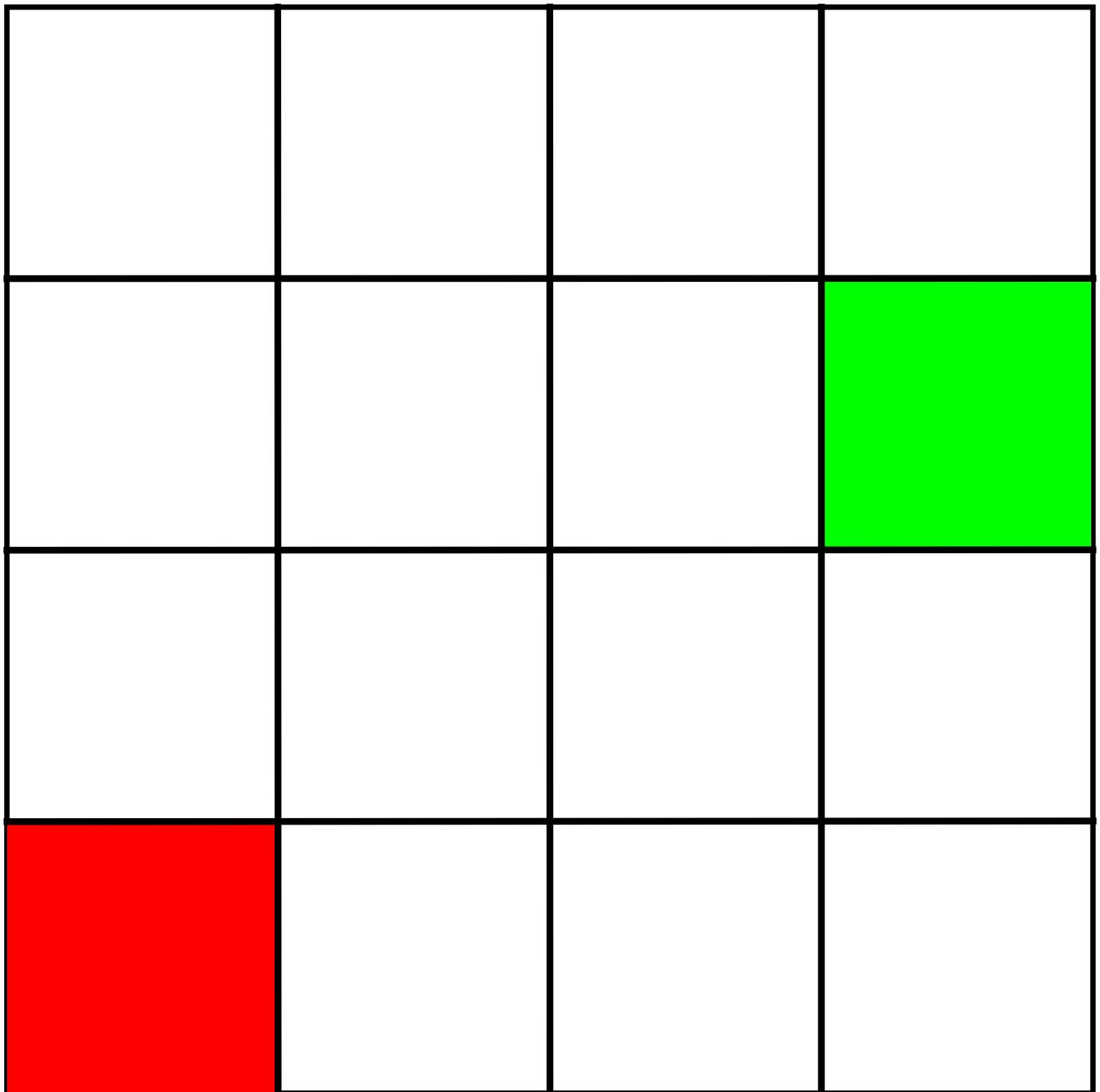
$g(n)$ имеет минимальную стоимость от корневого узла до n .

$h(n)$ имеет минимальную стоимость от n до ближайшей к n цели.

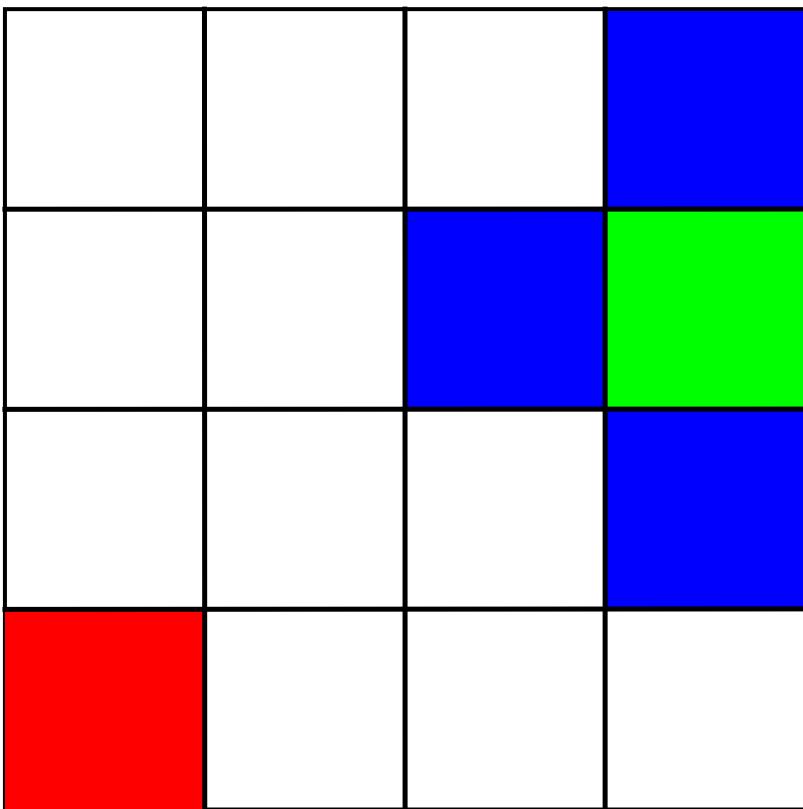
A* является информированным методом поиска, и он всегда гарантированно найдет кратчайший путь (путь с минимальной стоимостью) за наименьшее возможное время (если использует приемлемую эвристику). Поэтому он и оптимален, и совершенен. Следующая [анимация](#) наглядно демонстрирует работу A*.

Раздел 12.2: Поиск пути A* в лабиринте без препятствий

Скажем, что у нас есть сетка размера 4 на 4:



Представим, что перед нами лабиринт. Однако в нем отсутствуют препятствия/стенки. У нас имеется только стартовая точка (зеленая клетка) и конечная точка (красная клетка). Также примем тот факт, что при прохождении нашего пути мы не можем двигаться по диагонали. Так, начиная движение с зеленоей клетки, посмотрим, в какие клетки мы можем перейти из неё и пометим эти клетки синим цветом:



Чтобы выбрать, на какую же клетку нам стоит перейти, мы должны принять во внимание 2 эвристики:

1. Значение “g” . Показатель того, насколько далеко текущий узел находится от зеленой клетки.
2. Значение “h” . Показатель того, насколько далеко текущий узел находится от красной клетки.
3. Значение “f” -.Сумма значений “g” и “h”. Это результирующее число, которое и говорит нам, в какой узел (клетку) стоит идти.

Для вычисления этих эвристик мы будем использовать следующую формулу:

$$\text{distance} = \text{abs}(\text{from.x} - \text{to.x}) + \text{abs}(\text{from.y} - \text{to.y})$$

Эта формула также известна как “[Расстояние городских кварталов](#)”.

Посчитаем значение g клетки, находящейся слева от зеленой клетки:

$$\text{abs}(3 - 2) + \text{abs}(2 - 2) = 1$$

Отлично! Мы получили значение: “1”. А теперь, попробуем посчитать значение “h”:

$$\text{abs}(2 - 0) + \text{abs}(2 - 0) = 4$$

Превосходно. А теперь получим значение f:

$$1 + 4 = 5$$

В итоге мы получили финальное значение: “5”.

Проделаем аналогичные действия с другими синими клетками. Большое число посередине клетки - значение f , тем временем как числа в верхнем левом и верхнем правом углах - значения g и h соответственно:

			1 6 7
		1 4 5	
			1 4 5
Red			

Мы посчитали значения g , h и f для всех синих клеток(узлов). Что же мы должны выбрать сейчас?

Мы должны выбрать узел с наименьшим значением f .

Однако в таком случае мы имеем 2 узла с одинаковыми значениями f , равными 5. Какой из них нам выбрать?

Один из них мы просто берем случайно, или выставляем список приоритетов. Я обычно предполагаю расставлять приоритет следующим образом: “Вправо > Вверх > Вниз > Влево”

Один из узлов со значением 5 поведет нас в направлении “Вниз”, а другой поведет в направлении “Влево”. Так как приоритет направления “Вниз” выше приоритета направления “Влево”, мы выберем тот узел, что повел нас вниз.

Теперь я помечаю узлы, для которых мы вычислили эвристику, но не перешли в них - оранжевым. Узел, который мы выбрали, я отмечу бирюзовым:

			1 6 7
		1 4 5	
			1 4 5
Red			

Хорошо, а теперь вычислим те же эвристики для узлов, расположенных вокруг бирюзовой клетки:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
Red			2 3 5

И снова, мы выбираем узел, идущий вниз от бирюзовой клетки, так как мы вновь имеем две клетки с одинаковыми f значениями:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
			2 3 5

Посчитаем эвристики для единственного соседа бирюзовой клетки:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
		3 2 5	2 3 5

Хорошо, пока мы продолжаем в том же духе, мы будем видеть следующее:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
		3 2 5	2 3 5

И еще раз, посчитаем эвристики для соседнего узла:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
	4 1 5	3 2 5	2 3 5

Двинемся сюда:

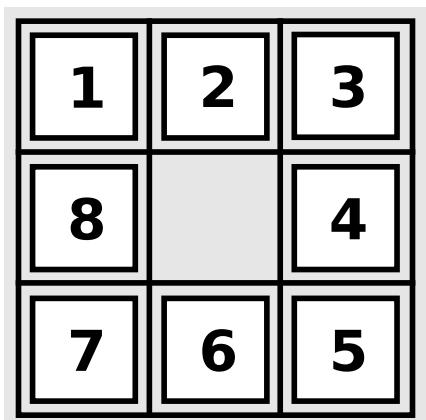
			1 6 7
		1 4 5	
		2 3 5	1 4 5
4 1 5	3 2 5	2 3 5	

И, наконец, мы можем увидеть клетку-победителя, делаем последнее передвижение, и дело сделано.

Раздел 12.3: Решаем “Пятнашки” (3x3) используя алгоритм A*

Описание проблемы:

“Пятнашки” - это простая игра, представляющая собой сетку размера 3 на 3 (содержащую 9 клеток). Одна из клеток пустая. Задача заключается в передвижении клеток между собой для получения определенного расположения чисел на этих клетках.



Нам же требуется получить из “начального” состояния “итоговое” состояние, используя минимальное количество действий.

Начальное состояние:

1	3
4	2
7	8

Итоговое состояние

1	2	3
4	5	6
7	8	_

Предполагаемая эвристика

Предположим, что расстояние городских кварталов(Manhattan distance) между текущим и финальным состоянием является эвристикой, требуемой для данной задачи.

$$h(n) = |x - p| + |y - q|$$

где x и у координаты в текущем состоянии
р и q координаты в финальном состоянии

Функция конечной стоимости

Функция конечной стоимости , полученная,

$$f(n) = g(n) + h(n), \text{ где } g(n) \text{ стоимость необходимая для достижения текущего состояния из заданного начального состояния}$$

Решение задачи

Сначала найдем значение эвристики, требуемое для получения итогового состояния из начального. Функция стоимости $g(n) = 0$, пока мы находимся в начальном состоянии.

$$h(n) = 8$$

Значение выше показывает, что “1” в текущем состоянии смещена на одну клетку по горизонтали от своего положения в финальном состоянии. Аналогично происходит для “2”, “5”, “6”. _ отличается от своего итогового положения на 2 клетки по горизонтали и на 2 по вертикали. Таким образом, общее значение для $f(n)$ получается: $1 + 1 + 1 + 1 + 2 + 2 = 8$. Функция конечной стоимости равна $8 + 0 = 8$.

Теперь, когда все возможные состояния, которые могут быть получены из исходного, найдены, мы видим, что можем переместить _ вправо или вниз.

Состояния, которые можно получить, выполнив доступные действия выглядят следующим образом:

1	_	3	4	1	3
4	2	5	_	2	5
7	8	6	7	8	6
(1)			(2)		

Снова функция стоимости для этих состояний рассчитывается с использованием метода, описанного выше, и оказывается в итоге равной 6 и 7 соответственно. Мы выбираем состояние

с наименьшей стоимостью, и им оказывается состояние 1. Следующими возможными шагами могут быть: шаг влево, шаг вправо или шаг вниз. Мы не станем делать шаг влево, так как мы пришли из этого состояния. Поэтому мы движемся либо вниз, либо вправо.

Опять находим состояния, которые могут быть получены из состояния (1).

$\begin{array}{ccc} 1 & 3 & \\ \underline{4} & 2 & 5 \\ 7 & 8 & 6 \end{array}$	$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & \underline{5} & \\ 7 & 8 & 6 \end{array}$
(3)	(4)

(3) приводит к получению значения 6, а (4) приводит к значению 4 в функции стоимости. Также будем считать, что состояние (2), найденное ранее, имеет стоимость 7. Выбираем состояние с минимальной стоимостью из всех возможных. И им становится состояние (4). Следующими возможными шагами могут быть шаги влево, вправо или вниз.

$\begin{array}{ccc} 1 & 2 & 3 \\ \underline{4} & 5 & \\ 7 & 8 & 6 \end{array}$	$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & \underline{5} & \\ 7 & 8 & 6 \end{array}$	$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 8 & 5 \\ \underline{7} & 8 & 6 \end{array}$
(5)	(6)	(7)

Получаем стоимости равные 5, 2 и 4 для (5), (6) и (7) соответственно. Также у нас есть предыдущие состояния (3) и (2) со стоимостью 6 и 7 соответственно. Выбираем состояние с наименьшей стоимостью - состояние (6). Следующие возможные шаги: вверх и вниз. И именно шаг вниз приведет нас к финальному состоянию, приравнивая значение эвристической функции к 0.

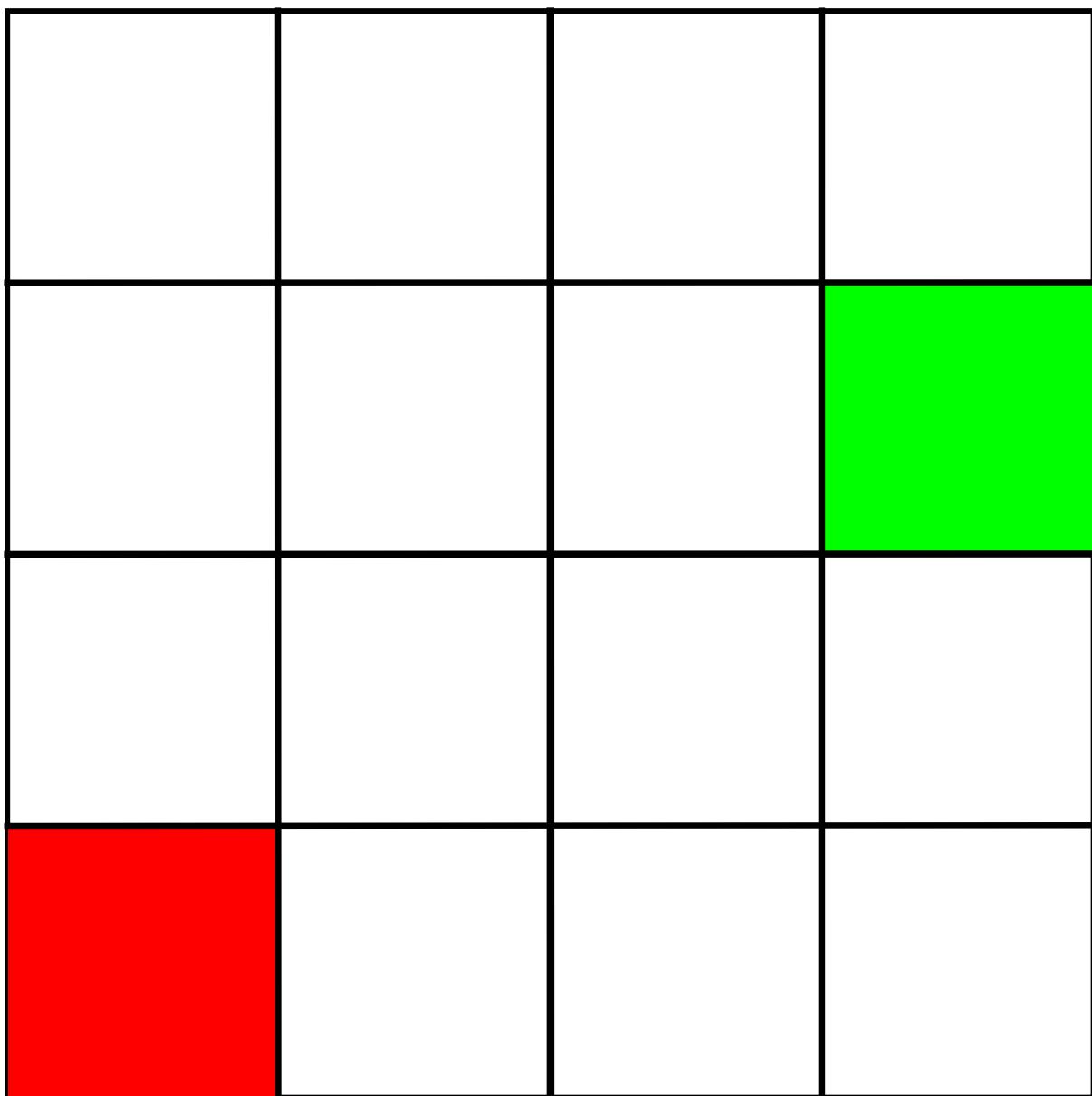
Глава 13: Алгоритм поиска пути A*

В этой теме основное внимание будет уделено алгоритму поиска пути A*, как он используется, и почему он работает.

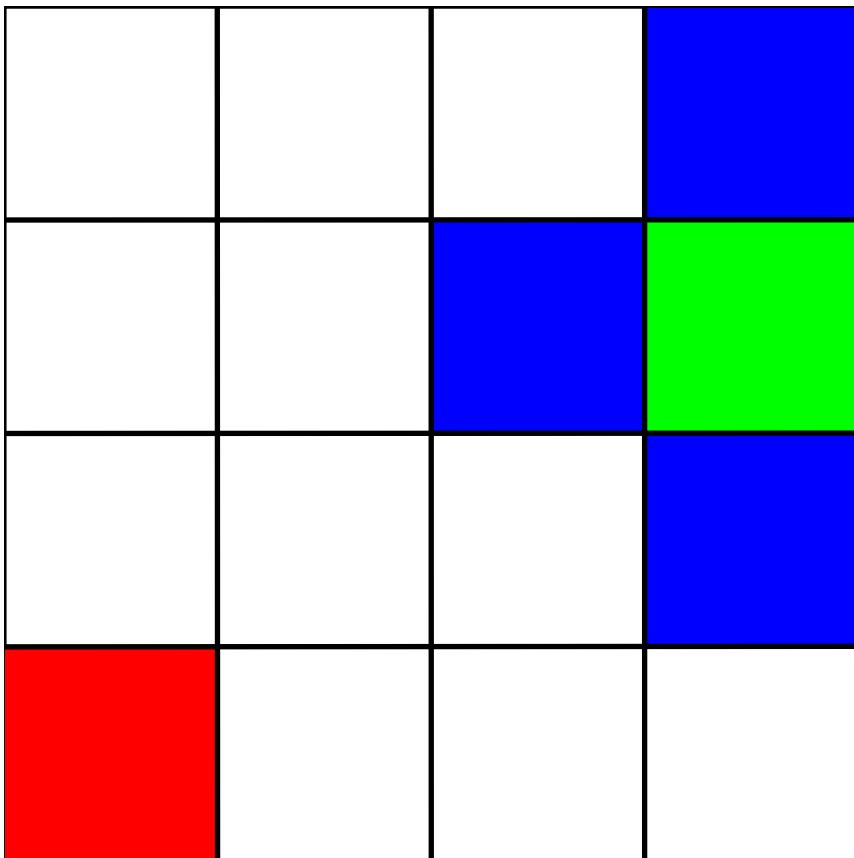
Примечание для будущих участников: я добавил пример для алгоритма A* на сетке 4x4 без каких-либо препятствий. Пример с препятствиями по-прежнему необходим.

Раздел 13.1: Простой пример алгоритма поиска пути A*: Лабиринт без препятствий

Допустим, у нас есть сетка размером 4 на 4:



Давайте предположим, что это лабиринт, хотя там нет стен или препятствий. У нас есть только начальная точка (зелёный квадрат) и конечная точка (красный квадрат). Давайте также предположим, что для того, чтобы перейти от зелёного к красному квадрату, мы не можем двигаться по диагонали. Итак, начиная с зеленого квадрата, давайте посмотрим, к каким квадратам мы можем перейти и выделим их синим цветом:



Чтобы выбрать квадрат, на который нужно перейти, необходимо учесть 2 эвристики:

1. Значение «g» - это расстояние от зеленого квадрата до этого узла.
2. Значение «h» - это то, как далеко от красного квадрата находится этот узел.
3. Значение «f» - это сумма значений «g» и «h». Это число, которое говорит нам, на какой узел перейти.

Чтобы вычислить эти эвристики, мы используем следующую формулу:
 $distance = abs(from.x - to.x) + abs(from.y - to.y)$

Эта формула известна как "[Манхэттенское расстояние](#)".

Давайте вычислим значение «g» для синего квадрата, который находится слева от зеленого:

$$abs(3 - 2) + abs(2 - 2) = 1$$

Отлично! У нас есть значение: 1. Теперь давайте попробуем вычислить значение «h»:
 $abs(2 - 0) + abs(2 - 0) = 4$

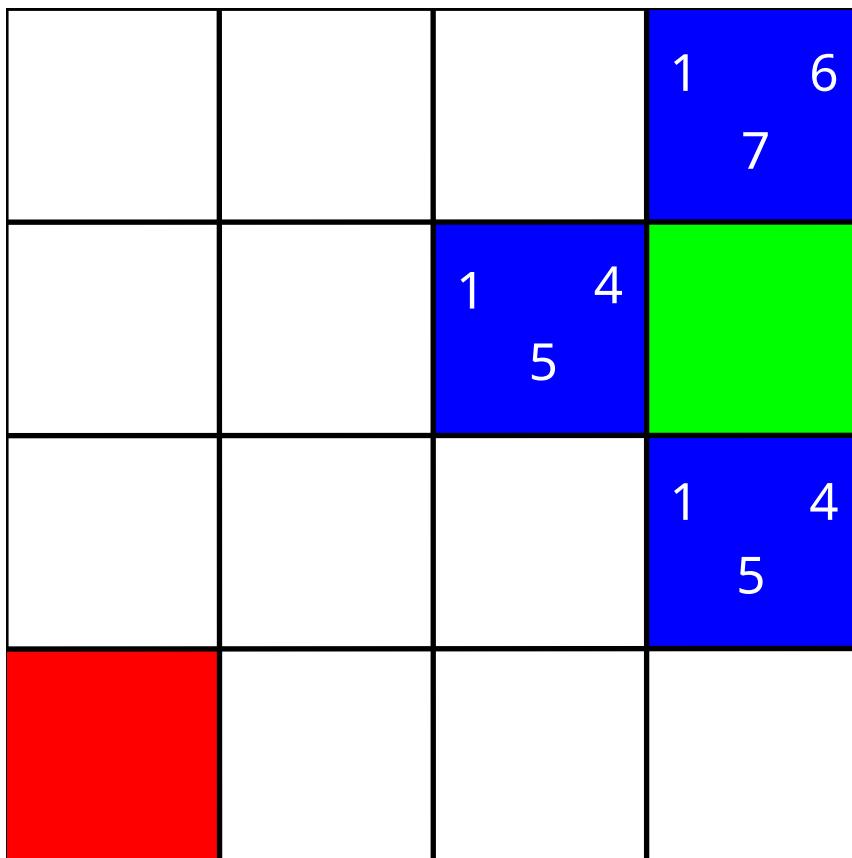
Идеально. Теперь давайте получим значение "f": $1 + 4 = 5$

Итак, окончательное значение для этого узла - «5».

Давайте сделаем то же самое для оставшихся сРаздел 14.2. Алгоритм взвешенного планирования работ Алгоритм взвешенного планирования работ также может быть обозначен как алгоритм выбора взвешенной активности.

Задача заключается в том, чтобы определить максимальную прибыль, не выполняя две работы одновременно, при этом учитывая определенные задания и время начала и окончания их выполнения, а также прибыль, которую возможно получить после завершения работы. иных

Давайте сделаем то же самое для оставшихся синих квадратов. Большое число в центре каждого квадрата - это значение «f», в верхнем левом углу -значение «g», а в верхнем правом углу - значение «h»: вадратов. Большое число в центре каждого квадрата - это значение «f», в верхнем левом углу -значение «g», а в верхнем правом углу - значение «h»:



Мы вычислили значения g, h и f для всех синих узлов. Какой мы выбираем?

Такой из них, который имеет наименьшее значение f.

Однако в этом случае у нас есть 2 узла с одинаковым значением f, равным 5. Как мы выбираем между ними?

Просто выберите случайным образом или установите приоритет. Я обычно предпочитаю ставить приоритет так: «справа> вверх> вниз> влево»

Один из узлов со значением f, равным 5, ведет нас «вниз», а другой - «влево». Так как вниз имеет более высокий приоритет, чем влево, мы выбираем квадрат, который приводит нас «вниз».

Теперь я отмечаю узлы, для которых мы вычислили эвристику, но не переместили, как оранжевый, и узел, который мы выбрали, как голубой:

			1 6 7
		1 4 5	
			1 4 5

Хорошо, теперь давайте вычислим ту же эвристику для узлов, которые расположены вокруг голубого узла:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
			2 3 5

Опять же, мы выбираем узел, идущий вниз от голубого узла, так как все опции имеют одинаковое значение f:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
			2 3 5

Давайте вычислим эвристику для единственного соседа, который имеет голубой узел:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
		3 2 5	2 3 5

Хорошо, мы будем следовать той же схеме, что и раньше:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
		3 2 5	2 3 5

Давайте еще раз вычислим эвристику для соседнего узла:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
	4 1 5	3 2 5	2 3 5

Давайте перейдем туда:

			1 6 7
		1 4 5	
		2 3 5	1 4 5
	4 1 5	3 2 5	2 3 5

Наконец, мы видим, что у нас есть выигрышная клетка, и двигаемся туда.

Глава 14: Динамическое программирование

Динамическое программирование является широко используемой концепцией и часто используется для оптимизации. Оно относится к упрощению сложной проблемы путем ее рекурсивного разбиения на более простые подзадачи, обычно снизу вверх. Есть два ключевых атрибута, которые должны содержать задачи, чтобы динамическое программирование было применимо: «Оптимальная подструктура» и «Дублирование подзадач». Для достижения оптимизации в динамическом программировании используется концепция, называемая запоминанием.

Раздел 14.1: Расстояние Левенштейна

Постановка задачи приближена к формулировке : если нам даны две строки str1 и str2, то какое минимальное количество операций может быть произведено над str1, чтобы преобразовать ее в str2.

Реализация на Java

```
public class EditDistance {  
    // первая запускаемая функция  
    public static void main(String[] args) {  
        String str1 = "march";  
        String str2 = "cart";  
        EditDistance ed = new EditDistance();  
        System.out.println(ed.getMinConversions(str1, str2));  
    }  
    // функция, возвращающая минимальное расстояние Левенштейна между двумя  
    // строками  
    public int getMinConversions(String str1, String str2) {  
        // dp[i][j] - минимальное расстояние Левенштейна между срезом первой  
        // строки до  
        // i и срезом второй строки до j  
        int dp[][] = new int[str1.length() + 1][str2.length() + 1];  
        for (int i = 0; i <= str1.length(); i++) {  
            for (int j = 0; j <= str2.length(); j++) {  
                // если длина первой строки 0, тогда нам нужно скопировать все  
                // символы из второй строки (их j)  
                if (i == 0) {  
                    dp[i][j] = j;  
                }  
                // если длина второй строки 0, тогда нам нужно скопировать все  
                // символы из первой строки (их i)  
                else if (j == 0) {  
                    dp[i][j] = i;  
                }  
                // если последние символы двух строк совпадают,  
                // тогда общее кол-во операций не меняется  
                else if (str1.charAt(i - 1) == str2.charAt(j - 1)) {  
                    dp[i][j] = dp[i - 1][j - 1];  
                }  
                // если последние символы двух строк не совпадают,  
                // тогда общее кол-во операций увеличивается на 1  
                else {  
                    dp[i][j] = Math.min(dp[i - 1][j], dp[i][j - 1]);  
                    dp[i][j]++;  
                }  
            }  
        }  
        return dp[str1.length()][str2.length()];  
    }  
}
```

```

    }
    // минимум из 3-х случаев: дописать/удалить символ к первому срезу,
    // ко второму и заменить последний символ одного среза на другой
    else {
        dp[i][j] = 1 + Math.min(dp[i - 1][j], Math.min(dp[i][j - 1],
            dp[i - 1][j - 1]));
    }
}
// ответ лежит в срезе первой строки до её длины, и срезе второй строки до
// её длины
return dp[str1.length()][str2.length()];
}
}

```

Вывод:

3

Раздел 14.2: Алгоритм взвешенного планирования работ

Алгоритм взвешенного планирования работ также может быть обозначен как алгоритм выбора взвешенной активности.

Задача заключается в том, чтобы определить максимальную прибыль, не выполняя две работы одновременно, при этом учитывая определенные задания и время начала и окончания их выполнения, а также прибыль, которую возможно получить после завершения работы.

Это похоже на задачу выбора действия с использованием жадного алгоритма, но есть ещё кое-что. То есть вместо максимизации количества выполненных работ, мы ориентируемся на получение максимальной выгоды. Количество выполненных заданий здесь не важно. Давайте посмотрим на пример:

Name	A	B	C	D	E	F
(Start Time,Finish Time)	(2,5)	(6,7)	(7,9)	(1,3)	(5,8)	(4,6)
Profit	6	4	2	5	11	5

Задания обозначены именем, временем их начала и окончания и выгодой. После нескольких итераций мы можем выяснить, что, если мы выполняем **задание-А** и **задание-Е**, мы можем получить максимальную выгоду 17. Как это выяснить с помощью алгоритма? Первое, что мы делаем, это сортируем задания по времени их окончания в порядке неубывания. Зачем мы это делаем? Потому что если мы выбираем задание, которое занимает меньше времени, то мы оставляем больше времени для выбора других работ. У нас есть:

Name	D	A	F	B	E	C
(Start Time,Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2

У нас имеется дополнительный временный массив **Acc_Prof** размера **n** (**n** здесь обозначает общее количество заданий). Он будет содержать максимальную накопленную выгоду от выполнения работ. Мы инициализируем значения массива с выгодой от каждого задания. Это означает, что **Acc_Prof [i]** сначала будет содержать прибыль от выполнения **i-го** задания.

Acc_Prof	5	6	5	4	11	2
----------	---	---	---	---	----	---

Теперь давайте обозначим **позицию 2** через **j**, а **позиция 1** будет обозначена через **j**. Наша стратегия будет состоять в том, чтобы перебрать **j** от **1** до **i-1**, и после каждой итерации мы будем увеличивать **i** на **1**, пока **i** не станет равна **n + 1**.

	j	i				
Name	D	A	F	B	E	C
(Start Time,Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2
Acc_Prof	5	6	5	4	11	2

Мы проверяем, не совпадают ли **задание[i]** и **задание[j]**, то есть, если время завершения **задания[j]** больше времени начала **задания[i]**, то эти две работы не могут быть выполнены вместе. Однако, если они не пересекаются, мы смотрим, если **Acc_Prof [j] + Profit [i] > Acc_Prof [i]**. В этом случае мы обновим **Acc_Prof [i] = Acc_Prof [j] + Profit [i]**. Это:

```

if Job[j].finish_time <= Job[i].start_time
    if Acc_Prof[j] + Profit[i] > Acc_Prof[i]
        Acc_Prof[i] = Acc_Prof[j] + Profit[i]
    endif
endif

```

Здесь **Acc_Prof [j] + Profit [i]** представляет собой накопленную выгоду от выполнения этих двух заданий вместе. Давайте проверим это для нашего примера: Здесь **задание[j]** пересекается с **заданием[i]**. Следовательно, они не могут быть выполнены вместе. Так как наш **j** равен **i-1**, мы увеличиваем значение **i** до **i + 1**, то есть до **3**. И мы устанавливаем **j = 1**.

	j		i			
Name	D	A	F	B	E	C
(Start Time,Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2
Acc_Prof	5	6	5	4	11	2

Теперь **задание[j]** и **задание[i]** не пересекаются. Общая выгода, которую мы можем получить, выбрав эти два задания: **Acc_Prof [j] + Profit [i]** = 5 + 5 = 10, что больше, чем **Acc_Prof [i]**. Поэтому мы обновляем **Acc_Prof [i]** = 10. Мы также увеличиваем **j** на 1. Мы получили,

	j		i			
Name	D	A	F	B	E	C
(Start Time,Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2
Acc_Prof	5	6	10	4	11	2

Здесь **задание[j]** перекрывается с **заданием[i]** и **j** равен **i-1**. Таким образом, мы увеличиваем **i** на 1 и устанавливаем **j = 1**. Мы получили,

	j		i			
Name	D	A	F	B	E	C
(Start Time,Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2
Acc_Prof	5	6	10	4	11	2

Теперь **задание[j]** и **задание[i]** не пересекаются, мы получаем выгоду 5 + 4 = 9, которая больше, чем **Acc_Prof [i]**. Обновим **Acc_Prof [i]** = 9 и увеличим **j** на 1.

	j		i			
Name	D	A	F	B	E	C
(Start Time,Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)

Profit	5	6	5	4	11	2
Acc_Prof	5	6	10	9	11	2

И снова **задание[j]** и **задание[i]** не пересекаются. Суммарная выгода: $6 + 4 = 10$, что больше, чем **Acc_Prof [i]**. Мы снова обновляем **Acc_Prof [i] = 10**. Увеличиваем **j** на 1. Получаем:

	j	i
Name	D	A
(Start Time,Finish Time)	(1,3)	(2,5)
Profit	5	6
Acc_Prof	5	6

	F	B	E	C
(Start Time,Finish Time)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	4	11	2
Acc_Prof	10	10	11	2

Если мы продолжим этот процесс после итерации всей таблицы с использованием **i**, наша таблица будет выглядеть примерно так:

Name	D	A	F	B	E	C
(Start Time,Finish Time)	(1,3)	(2,5)	(4,6)	(6,7)	(5,8)	(7,9)
Profit	5	6	5	4	11	2
Acc_Prof	5	6	10	14	17	8

* Несколько шагов были пропущены, чтобы сделать документ короче.

Если мы проведем итерацию по массиву **Acc_Prof**, мы увидим, что максимальная выгода равна **17!** Псевдокод:

```
// сортировка по времени окончания в неубывающем порядке
// первый цикл начинается с 2 элемента, второй --- с 1 элемента
for i -> 2 to n
    for j -> 1 to i-1
        // Если окончание работы раньше чем начало следующей
        if Job[j].finish_time <= Job[i].start_time
            // и если накопленная выгода больше текущей, то
            if Acc_Prof[j] + Profit[i] > Acc_Prof[i]
                // Изменяем текущую выгода на накопленную
                Acc_Prof[i] = Acc_Prof[j] + Profit[i]
            endif
    endif
endif
```

```

        endif
    endfor
endfor

// ищем максимальную выгоду
maxProfit = 0
for i -> 1 to n
    if maxProfit < Acc_Prof[i]
        maxProfit = Acc_Prof[i]
    endif
endfor
// возвращаем максимальную выгоду
return maxProfit

```

Сложность заполнения массива **Acc_Prof** — $O(n^2)$. Обход массива занимает $O(n)$. Таким образом, общая сложность этого алгоритма равна $O(n^2)$.

Если мы хотим выяснить, какие задания были выполнены, чтобы получить максимальную выгоду, нам нужно пройти по массиву в обратном порядке, и если **Acc_Prof** соответствует **maxProfit**, мы поместим название задания в стек и вычтем выгоду от выполнения этого задания от **maxProfit**. Мы будем делать это до тех пор, пока **maxProfit > 0** или мы не достигнем начальной точки массива **Acc_Prof**. Псевдокод будет выглядеть так:

```

Procedure FindingPerformedJobs(Job, Acc_Prof, maxProfit):
S = stack()
// обратный цикл с условием
for i -> n down to 0 and maxProfit > 0
    // если максимальная выгода равна текущей выгоде
    if maxProfit is equal to Acc_Prof[i]
        // помещаем название задания в стек
        S.push(Job[i].name)
        // вычитаем выгоду от выполнения этого задания от maxProfit
        maxProfit = maxProfit - Job[i].profit
    endif
endfor

```

Сложность этой процедуры: $O(n)$.

Следует помнить одну вещь: если есть несколько графиков работы, которые могут дать нам максимальную прибыль, мы можем найти только один график работы с помощью этой процедуры.

Раздел 14.3: Самая длинная общая подпоследовательность

Если нам даны две строки, мы должны найти самую длинную общую подпоследовательность, присутствующую в обеих из них.

Пример

Самая длинная общая подпоследовательность для входных последовательностей «ABCDGH» и «AEDFHR» — это «ADH» длины 3.

Самая длинная общая подпоследовательность для входных последовательностей «AGGTAB» и «GXTXAYB» — это «GTAB» длины 4.

Реализация на Java

```
public class LCS {
    // начало вычислений
    public static void main(String[] args) {
        // TODO Автоматически сгенерированный метод заглушки
        String str1 = "AGGTAB";
        String str2 = "GXTXAYB";
        // берем объект наибольшей общей подпоследовательности
        LCS obj = new LCS();
        // выводим наибольшую общую последовательность
        // аргументы - два слова и два значения длины слов соответственно
        System.out.println(obj.lcs(str1, str2, str1.length(), str2.length()));
        // аргументы - два слова
        System.out.println(obj.lcs2(str1, str2));
    }
    // Рекурсивная функция
    public int lcs(String str1, String str2, int m, int n){
        if(m==0 || n==0)
            return 0;
        // если символы совпадают, рекурсивно прибавляем 1 к результату
        if(str1.charAt(m-1) == str2.charAt(n-1))
            return 1 + lcs(str1, str2, m-1, n-1);
        else
            // иначе сдвиг на символ по второму слову
            // максимум между словами со сдвинутыми индексами на одну позицию
            /* первый аргумент - наибольшая общая последовательность при сдвиге
            // на символ в первом слове, второй - во втором слове */
            return Math.max(lcs(str1, str2, m-1, n), lcs(str1, str2, m, n-1));
    }
    // Итерационная функция
    public int lcs2(String str1, String str2){
        // определяем двумерный массив исходных слов
        int lcs[][] = new int[str1.length()+1] [str2.length()+1];
        // итерация по строке и перебор столбцов
        for(int i=0;i<=str1.length();i++){
            for(int j=0;j<=str2.length();j++){
                // начало отсчета
                if(i==0 || j== 0){
                    lcs[i][j] = 0;
                }
                // прибавляем к матрице счетчиков
                else if(str1.charAt(i-1) == str2.charAt(j-1)){
                    lcs[i][j] = 1 + lcs[i-1][j-1];
                }else{

```

```

        // "протягивание" к углу матрицы максимального значения
        lcs[i][j] = Math.max(lcs[i-1][j], lcs[i][j-1]);
    }
}
// возвращаем максимально значение в нижним правом углу
return lcs[str1.length()][str2.length()];
}
}

```

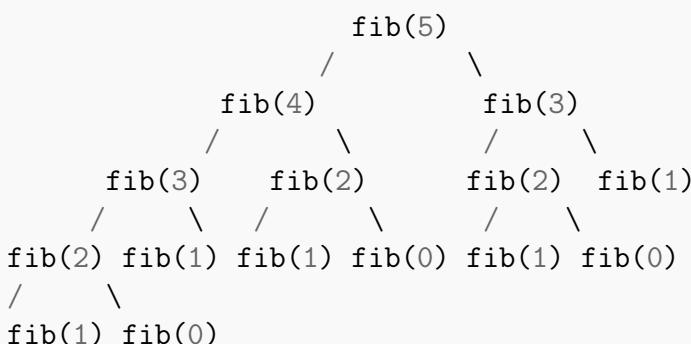
Вывод

4

Раздел 14.4. Число Фибоначчи

Подход «снизу вверх» для печати n-го числа Фибоначчи с использованием динамического программирования.

Рекурсивное дерево



Пересекающиеся подзадачи

Здесь fib(0), fib(1) и fib(3) являются пересекающимися подзадачами. Fib(0) повторяется 3 раза, fib(1) повторяется 5 раз, а fib(3) повторяется 2 раза.

Реализация

```

public int fib(int n){
    // определение массива
    int f[] = new int[n+1];
    // начальные значения
    f[0]=0; f[1]=1;
    for(int i=2;i<=n;i++){
        // динамический подсчет
        f[i]=f[i-1]+f[i-2];
    }
}

```

```
    return f[n];
}
```

Сложность выполнения

$O(n)$

Раздел 14.5: Самая длинная общая подстрока

Даны 2 строки **str1** и **str2**, мы должны найти длину самой длинной общей подстроки между ними.

Примеры

Вход: X = "abcdxyz" y = "xyzabcd" Выход: 4

Самая длинная общая подстрока — это «abcd», она имеет длину 4.

Вход: X = "zxabcdezy" y = "yzabcdez" Выход: 6

Самая длинная общая подстрока — это «abcdez», она имеет длину 6.

Реализация на Java

```
public int getLongestCommonSubstring(String str1, String str2){
    //определение двумерного массива
    int arr[][] = new int[str2.length()+1][str1.length()+1];
    // инициализируем минимальное значение для поиска максимального
    int max = Integer.MIN_VALUE;
    // итерации по матрице
    for(int i=1;i<=str2.length();i++){
        for(int j=1;j<=str1.length();j++){
            // если символы совпадают
            if(str1.charAt(j-1) == str2.charAt(i-1)){
                // записываем в соседнюю клетку, ближе к правому нижнему углу
                arr[i][j] = arr[i-1][j-1]+1;
                // запоминаем максимальное значение
                if(arr[i][j] > max)
                    max = arr[i][j];
            }
            else // если символы не равны
                arr[i][j] = 0;
        }
    }
    // возврат максимального значения
    return max;
}
```

Сложность алгоритма

$O(m*n)$

Глава 15: Применение динамического программирования

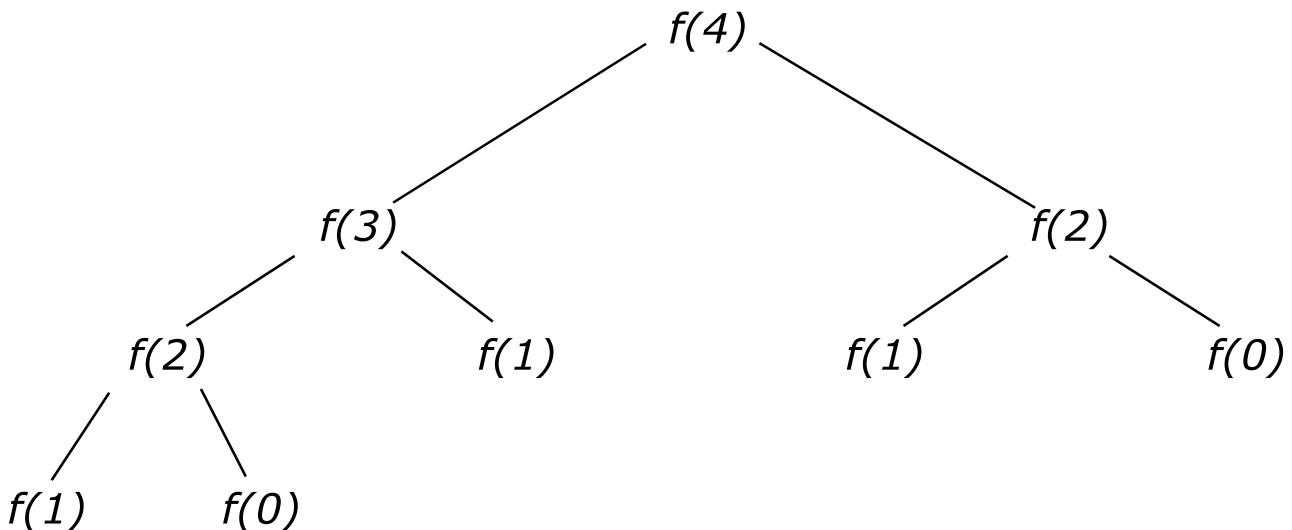
Основная идея динамического программирования заключается в разбиении трудной задачи на несколько маленьких, простых и многократно встречающихся задач. Если в задаче отчетливо заметна простая многократно решаемая подзадача, вполне возможно, что к ней можно применить данный подход.

В силу того, что заголовок этой главы содержит слово «*применение*», упор в ней сделан скорее на применения, чем на процесс создания алгоритмов.

Раздел 15.1: Числа Фибоначчи

Числа Фибоначчи прекрасно демонстрируют принцип динамического программирования, поскольку традиционный рекурсивный подход к ним задействует множество повторяющихся вычислений. В этих примерах я буду стандартно полагать $f(0) = f(1) = 1$.

Ниже представлен пример рекурсивного дерева для числа Фибоначчи (4), обратите внимание на повторяющиеся вычисления:



Нединамическое программирование $O(2^n)$ временная сложность алгоритма, $O(n)$ сложность стека

```
def fibonacci(n):      # функция по вычислению числа фибоначчи
    if n < 2: # для чисел из заданного промежутка возможен только один результат
        return 1
    return fibonacci(n-1) + fibonacci(n-2) # рекурсивный вызов функции
```

Это самый интуитивный способ написать решение. В большинстве случаев стековое пространство

ство будет $O(n)$ по мере спуска по первой рекурсивной ветви вызовов $\text{fibonacci}(n-1)$ до случая $n < 2$.

Доказательство оценки временной сложности можно посмотреть тут:

[вычислительная сложность последовательности Фибоначчи](#). Основное внимание стоит уделить тому, что сложность экспоненциальная, иными словами, время выполнения программы удваивается с каждым последующим числом, например, $f(15)$ будет вычисляться вдвое дольше, чем $f(14)$.

Мемоизированная за $O(n)$ времененная сложность алгоритма, $O(n)$ сложность пространства, $O(n)$ стековая сложность

```
memo = [] # создадим массив для записи в него вычисляемых чисел фибоначчи
# добавим в массив два первых числа, необходимых для вычисления последующих
memo.append(1) # f(1) = 1
memo.append(1) # f(2) = 1
def fibonacci(n):
    if len(memo) > n: # проверим не находили ли мы значение данного числа ранее
        return memo[n] # если мы уже находили значение текущего числа, то просто
                        # выводим его
    result = fibonacci(n-1) + fibonacci(n-2) # вычисляем результат и записываем
    # значение в массив для уменьшения количества вычислений в будущем
    memo.append(result) # f(n) = f(n-1) + f(n-2)
    return result
```

В этом случае мы вводим массив, который можно считать состоящим из всех предыдущих вызовов функции. В ячейке `memo[n]` хранится результат вызова $\text{fibonacci}(n)$. Это позволяет нам ценой сложности пространства $O(n)$ уменьшить временную сложность, поскольку нам не придется вычислять повторные вызовы лишний раз.

Итеративное динамическое программирование $O(n)$ временная сложность, $O(n)$ сложность пространства, рекурсивный стек отсутствует

```
def fibonacci(n):
    # создадим массив и добавим в него два первых числа
    memo = [1,1] # f(0) = 1, f(1) = 1
    for i in range(2, n+1):
        memo.append(memo[i-1] + memo[i-2]) # циклично вычисляем значения чисел
                                            # фибоначчи и добавляем их в массив
    return memo[n] # возвращаем искомое число
```

Сведя задачу к основаниям, читатель может заметить, что для вычисления $\text{fibonacci}(n)$ необходимо вычислить $\text{fibonacci}(n-1)$ и $\text{fibonacci}(n-2)$. Также он может заметить, что базовый случай возникает в конце того рекурсивного дерева, изображенного выше.

Со знанием этого, имеет смысл вычислять ответ "задом наперед начиная с базовых значений и поднимаясь выше. Теперь для вычисления $\text{fibonacci}(n)$ потребуется вычислить **все** числа Фибоначчи до n -го включительно.

Главным достоинством является то, что теперь мы устранили рекурсивный стек, сохранив вре-

менную сложность $O(n)$. Но, к сожалению, изменение пространственной сложности потребует отдельной заботы.

Улучшенное итеративное динамическое программирование $O(n)$ временная сложность, $O(1)$ пространственная сложность, рекурсивный стек отсутствует

```
def fibonacci(n):
    # создадим массив и добавим в него два первых числа
    memo = [1, 1] # f(0) = 1, f(1) = 1
    for i in range(2, n):
        memo[i%2] = memo[0] + memo[1] # заполняем массив числами,
                                    # предшествующими искомому
    return memo[n%2] # возвращаем искомое число
```

Как сказано выше, итеративное динамическое программирование делает свое дело, начиная с базовых случаев, поднимаясь к окончательному результату. Ключевое наблюдение, необходимое для получения пространственной сложности $O(1)$, аналогично наблюдению, проявленному в рекурсивном случае: нам необходимо знать лишь $\text{fibonacci}(n-1)$ и $\text{fibonacci}(n-2)$ для вычисления $\text{fibonacci}(n)$. Это означает, что в любой момент работы программы нам необходимо хранить в памяти лишь два предыдущих значения.

Чтобы хранить эти два значения я использую двухэлементный массив и обращаюсь к нему путем взятия остатка от деления на два индекса i , меняющегося следующим образом: $0, 1, 0, \dots, i \% 2$.

Я добавляю оба индекса массива поскольку все мы знаем, что сложение коммутативно ($5 + 6 = 11$ and $6 + 5 == 11$). Результат записывается затем в ячейку, к которой мы обращались в предпоследний раз (обозн. $i \% 2$). Окончательный ответ будет записан в позицию $n \% 2$.

Замечание

- Важно понимать, что иногда бывает правильнее использовать итеративное мемоизированное решение для функций, которые выполняют огромные повторяющиеся вычисления, поскольку в таком случае последующие обращения к этим функциям будут иметь сложность $O(1)$, как уже вычисленные ввиду кэширования ответа.

Глава 16. Алгоритм Крускала

Раздел 16.1: Оптимальное применение на основе непересекающихся множеств.

Есть два способа улучшить простые и субоптимальные не пересекающиеся множественные по-далгоритмы:

1. **Эвристика сжатия путей:** findSet не нуждается в обработке дерева, высота которого больше двух. В случае необходимости она может связать нижние узлы напрямую с корнем, оптимизируя будущие трансверсали:

```
subalgo findSet(v: a node):
    if v.parent != v
        v.parent = findSet(v.parent) # функция рекурсивно проходит от любого
                                      # узла к корню
    return v.parent
```

2. **Перемешивание, основанное на высоте:** для каждого узла хранится высота его поддерева. При перемешивании нужно сделать более высокое дерево родителем менее высокого, не увеличивая высоту того или иного дерева.

```
subalgo unionSet(u, v: nodes):
    # связем два узла с их корнями
    vRoot = findSet(v)
    uRoot = findSet(u)
    if vRoot == uRoot: # проверим, чтобы два узла не совпадали
        return
    if vRoot.height < uRoot.height: # согласно функции поиска корня при
        # выполнении данного условия один из корней будет родителем второго
        vRoot.parent = uRoot
    else if vRoot.height > uRoot.height: # согласно функции поиска корня при
        # выполнении данного условия один из корней будет родителем второго
        uRoot.parent = vRoot
    else:
        uRoot.parent = vRoot # если не выполняются оба условия будем считать
                            # один из корней родителем другого по умолчанию
    uRoot.height = uRoot.height + 1
```

Это приводит к времени выполнения $O(\alpha(n))$ для каждой операции, где α - обратная функция к быстро возрастающей функции Аккермана. Таким образом, ее собственный рост крайне замедлен и может считаться за $O(1)$ в практических целях.

Благодаря этому и начальной сортировке, для всего алгоритма Крускала верно $O(m \log m + m) = O(m \log m)$.

Замечание

Сжатие путей может уменьшить высоту дерева, что сильно усложняет сравнение высот деревьев в процессе объединения. Во избежание трудностей с хранением и вычислением высот деревьев результирующий родитель может быть выбран случайным образом:

```
subalgo unionSet(u, v: nodes):
    # соединим два узла с их корнями
    vRoot = findSet(v)
    uRoot = findSet(u)
    if vRoot == uRoot: # проверим, чтобы два узла не совпадали
        return
    if random() % 2 == 0: # выберем родителя случайным образом
        vRoot.parent = uRoot
    else:
        uRoot.parent = vRoot
```

На практике такой алгоритм в сочетании со сжатием путей для операции `findSet` будет выдавать сравнимый результат при гораздо более простом использовании.

Раздел 16.2: Простое, более детальное использование

Чтобы эффективно справляться с обнаружением циклов, будем считать каждый узел частью дерева. По мере добавления ребра определяем, являются ли компонентные узлы частями различных деревьев.

```
algorithm kruskalMST(G: a graph)
    sort Gs edges by their value
    # отсортируем Gs ребра по их значению
    MST = a forest of trees, initially each tree is a node in the graph
    # MST = лес деревьев, изначально каждое дерево является узлом графа
    for each edge e in G: # для каждого ребра в G:
        if the root of the tree that e.first belongs to is not the same
        # если корень дерева, к которому принадлежит e.first, не совпадает
            as the root of the tree that e.second belongs to:
        # как корень дерева, к которому принадлежит e.second:
            connect one of the roots to the other, thus merging two trees
        # соединить один из корней с другим, таким образом объединяя два дерева
        return MST, which now a single-tree forest
    # вернуть MST, который теперь представляет из себя лес с одним деревом
```

Раздел 16.3: Простое использование, основанное на непересекающихся множествах

Вышеупомянутая лесная методология на самом деле является системой непересекающихся множеств, использующей три главных операции:

```
subalgo makeSet(v: a node): # создаём поддерево
    v.parent = v      # создаём поддерево с корнем v

subalgo findSet(v: a node): # находим корень узла по уже известной функции
    if v.parent == v:
        return v
    return findSet(v.parent)

subalgo unionSet(v, u: nodes): # для двух корней определим иерархию
    vRoot = findSet(v)
    uRoot = findSet(u)
    uRoot.parent = vRoot

algorithm kruskalMST(G: a graph):
    sort Gs edges by their value # отсортируем Gs грани по их значениям
    for each node n in G: # для каждого узла создадим поддерево
        makeSet(n)
    for each edge e in G: # если узлы не принадлежат к одному корню, объединим
        # деревья
        if findSet(e.first) != findSet(e.second):
            unionSet(e.first, e.second)
```

Время для управления системой непересекающихся множеств занимает при наивном использовании $O(n \log n)$, таким образом время работы всего алгоритма Крускала составляет $O(m^*n \log n)$.

Раздел 16.4: Простое высокоуровневое использование

Сортируем ребра по величине и добавляем их в MST в отсортированном порядке, если они не образуют цикл.

```
algorithm kruskalMST(G: a graph)
    sort Gs edges by their value # отсортируем Gs ребра по их значению
    MST = an empty graph # MST = пустой граф
    for each edge e in G: # для каждого ребра в G:
        if adding e to MST does not create a cycle:
            # если добавление e в MST не создает цикл:
                add e to MST # добавим e к MST
    return MST
```

Глава 17: Жадные алгоритмы

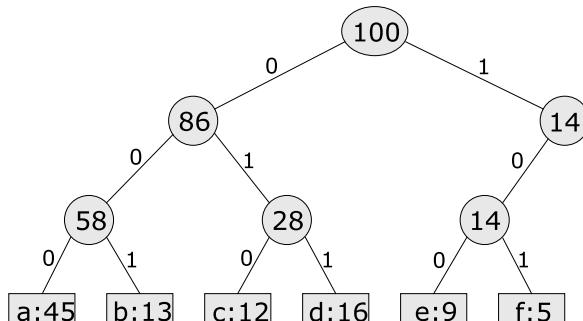
Раздел 17.1: Кодировка Хаффмана

Код Хаффмана — это особый тип оптимального префиксного кода, который повсеместно используется для сжатия данных без потерь. Он сжимает данные крайне эффективно, экономя от 20% до 90% памяти, в зависимости от характеристики сжимаемых данных. Здесь мы считаем данными символные последовательности. Жадный алгоритм Хаффмана использует таблицу частот возникновения того или иного символа в последовательности, оптимально представляя каждый символ в виде бинарной строки. Код Хаффмана был предложен [Дэвидом А. Хаффманом](#) в 1951 году.

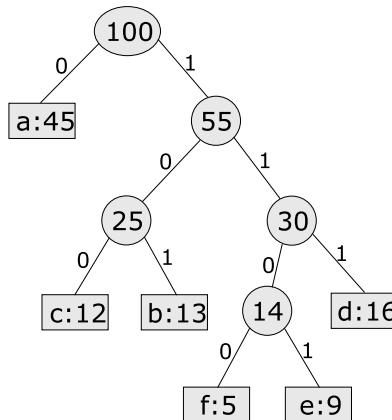
Предположим, у нас есть файл с данными в виде 100000 символов, который мы хотели бы сжать. Предположим, что в файле встречаются только шесть различных символов. Ниже приведена частота их появления:

Character	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

Мы располагаем многочисленными способами представить такой файл. Здесь мы рассмотрим задачу составления *бинарного символического кода*, в котором каждый символ будет представляться бинарной строкой, называемой **кодовым словом**.



Кодовое слово постоянной длины



Кодовое слово переменной длины

Благодаря построенному дереву мы видим, что:

Character	a	b	c	d	e	f
Fixed-length Codeword	000	001	010	011	100	101
Variable-length Codeword	0	101	100	111	1101	1100

Если мы используем **код фиксированной длины**, нам потребуется 3 бита для представления 6 символов. Такой метод потребует 300000 бит для кодирования всего файла. А можем ли мы улучшить положение? **Код переменной длины** может работать гораздо лучше кода фиксированной длины. Частым символам присваиваются короткие кодовые слова, а нечастым символам — длинные. Этот код требует $(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000 = 224000$ бит для представления файла, что экономит приблизительно 25% памяти.

Следует помнить одно: мы рассматриваем здесь только те коды, в которых ни одно кодовое слово не является началом (префиксом) другого. Такие коды называются *префиксными*. Для кодирования с переменной длиной мы кодируем 3-символьный файл abc как 0.101.100 = 0101100, где “.” означает конкатенацию.

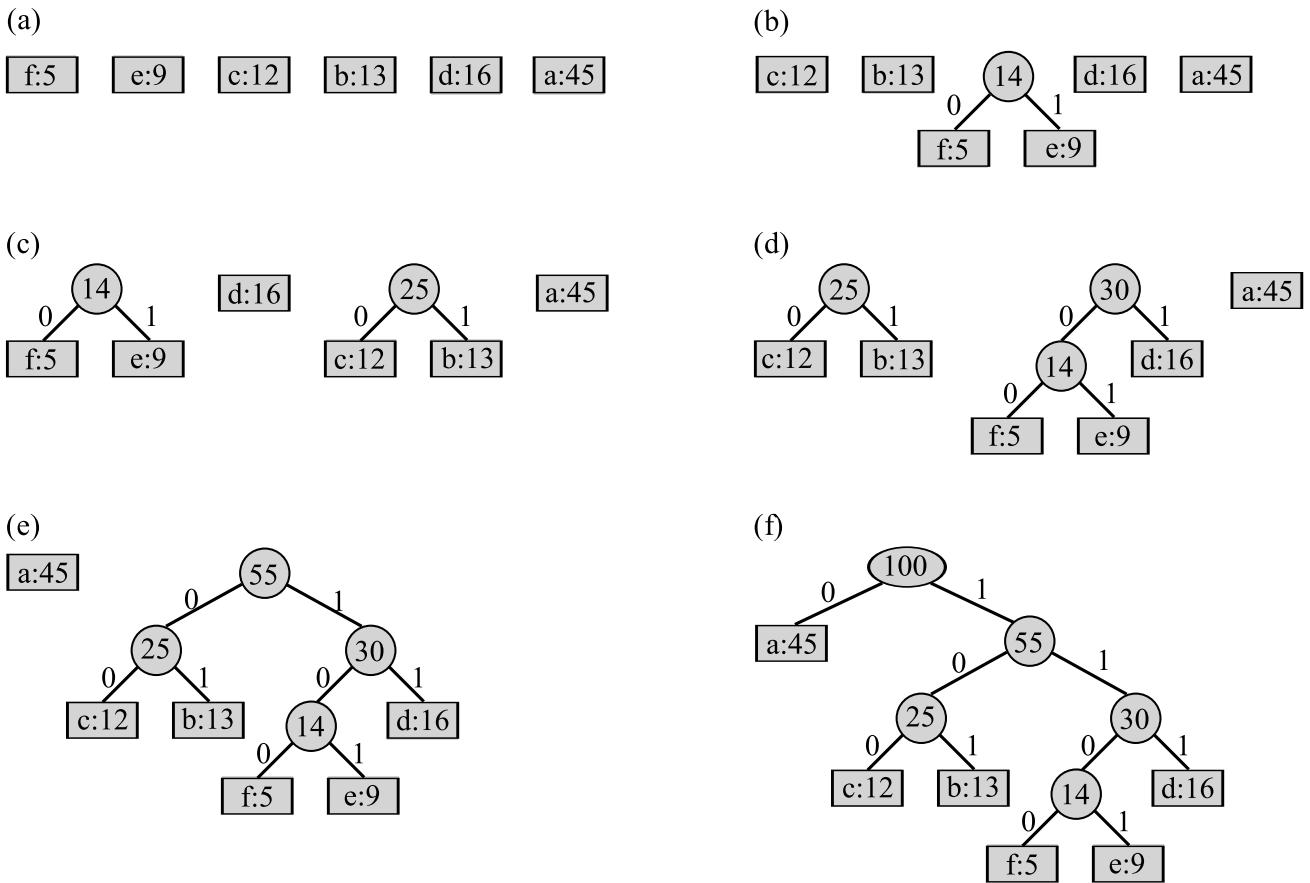
Желательно использовать префиксные коды, поскольку они упрощают декодирование. Так как ни одно кодовое слово не является префиксом другого, то кодовое слово, с которого начинается файл, декодируется однозначно. Мы можем легко идентифицировать исходное кодовое слово, декодировать его и повторить этот процесс на оставшейся части закодированного файла. Например, 001011101 однозначно представляется как 0.0.101.1101, что декодируется как aabe. Короче говоря, все комбинации двоичных представлений уникальны. Например, если одна буква обозначается как 110, то никакая другая буква не может обозначаться как 1101 или 1100. Это сделано для того, чтобы у вас не было неопределенности: выбрать 110 или продолжить конкатенацию следующих битов и выбрать более длинное кодовое слово.

Техника сжатия

Эта техника работает путем создания двоичного дерева узлов. Они могут храниться в обычном массиве, размер которого зависит от количества символов **n**. Узел может быть или *листовым*, или *внутренним*. Изначально все узлы — листья и содержат сам символ, его частоту и, возможно, ссылку на его сыновей. Как правило, битом ‘0’ обозначают левого сына, а ‘1’ — правого. Для хранения узлов используется очередь с *приоритетом*, где на первом месте находится узел с наименьшей частотой. Этот процесс описан ниже:

1. Создайте узел для каждого символа и добавьте его в очередь с приоритетом.
2. Пока в очереди хранится более одного узла:
 1. Удалите из очереди два узла с наибольшим приоритетом.
 2. Создайте новый внутренний узел с этими двумя узлами в качестве сыновей и с частотой, равной сумме частот этих двух узлов.
 3. Добавьте новый узел в очередь.
3. Оставшийся узел будет корнем. Дерево Хаффмана построено.

Для нашего примера:



Псевдокод выглядит так:

```
# C - это набор из n символов и соответствующей информации #
Procedure Huffman(C):
    n = C.size
    Q = priority_queue()
    for i = 1 to n
        n = node(C[i])
        Q.push(n)
    end for
    # пока размер информации не равен 1
    while Q.size() is not equal to 1
        # создаем новый внутренний узел
        Z = new node()
        # устанавливаем узлы в качестве потомков
        Z.left = x = Q.pop
        Z.right = y = Q.pop
        # частота, равная сумме частот двух узлов
        Z.frequency = x.frequency + y.frequency
        Q.push(Z)
    end while
    Return Q
```

В общем случае для использования этого алгоритма требуется предварительная сортировка.

Поскольку сортировка в общем случае занимает $O(n * \log(n))$ времени, оба метода имеют одинаковую сложность.

Поскольку n здесь — это число символов в алфавите, которое обычно очень мало (по сравнению с длиной закодированного сообщения), временная сложность не особо важна при выборе алгоритма.

Техника декомпрессии:

Процесс декомпрессии — это просто перевод потока префиксных кодов в отдельное байтовое значение, обычно, путем обхода дерева Хаффмана по мере считывания каждого бита из входного потока. Достигение листового узла обязательно завершает поиск конкретного значения байта. Значение листа представляет собой требуемый символ. Обычно дерево Хаффмана строится с использованием статистически скорректированных данных по каждому циклу компрессии, поэтому реконструкция довольно проста. В противном случае информация для восстановления дерева должна быть отправлена отдельно. Псевдокод:

```
Procedure HuffmanDecompression(root, S):
    # root - представляет корень дерева Хаффмана S относится к потоку битов,
    # который нужно распаковать
    n := S.length
    for i := 1 to n
        current = root
        # пока не спустились по дереву
        while current.left != NULL and current.right != NULL
            # переход влево, если попадается 0, иначе - вправо
            if S[i] is equal to '0'
                current := current.left
            else
                current := current.right
            endif
            # следующий символ, перехватываем индекс i
            i := i+1
        endwhile
        # индекс i снова перебирается другим циклом
        print current.symbol
    endfor
```

Объяснение жадности алгоритма: Кодирование Хаффмана рассматривает вхождение каждого символа и сохраняет его в виде двоичной строки оптимальным образом. Идея состоит в том, чтобы присваивать входным символам коды переменной длины, длина присваиваемых кодов основана на частоте соответствующих символов. Мы создаем двоичное дерево и работаем с ним “снизу вверх” так, чтобы как минимум два часто встречающихся символа были как можно дальше от корня. Таким образом, самый частый символ получает наименьший код, а наименее частый — наибольший.

Рекомендации:

- Алгоритмы. Построение и анализ — Чарльз Лейзерсон, Клиффорд Штайн, Рональд Ривест, и Томас Кормен
- [Код Хаффмана](#) — Википедия

- Discrete Mathematics and Its Applications — Кеннет Розен

Раздел 17.2: Проблема выбора вида деятельности

Проблема

У вас есть набор задач, которые нужно выполнить. Каждое задание имеет время начала и время окончания. Вам не разрешается выполнять более одной задачи за раз. Ваша задача — найти способ выполнить максимальное количество заданий.

Например, предположим, что вам нужно выбрать уроки из списка.

№ задачи	Начало	Конец
1	10.20 А.М.	11.00 А.М.
2	10.30 А.М.	11.30 А.М.
3	11.00 А.М.	12.00 А.М.
4	10.00 А.М.	11.30 А.М.
5	9.00 А.М.	11.00 А.М.

Помните, что вы не можете присутствовать на двух занятиях одновременно. Это означает, что вы не можете выбрать уроки 1 и 2, потому что у них общее время с 10:30 до 11:00. Однако, вы можете выбрать занятия 1 и 3, потому что они проходят в разное время. Итак, ваша задача состоит в том, чтобы выбрать максимально возможное число уроков без накладок. Как вы можете это сделать?

Анализ

Давайте подумаем над решением с помощью жадного подхода. Прежде всего мы случайным образом выбираем какой-нибудь подход и проверяем, сработает он или нет.

- **Сортировка задач по времени начала.** Это означает, что задания, которые начинаются раньше всего, мы будем выполнять в первую очередь. Затем будем перебирать задачи по порядку и проверять, будут ли они пересекаться по времени с предыдущим выбранным заданием или нет. Если текущая работа не пересекается с ранее выбранной, то мы будем ее выполнять, а в противном случае не будем. Этот подход сработает для некоторых случаев. Например:

№ задачи	Начало	Конец
1	11.00 А.М.	1.30 Р.М.
2	11.30 А.М.	12.00 Р.М.
3	1.30 Р.М.	2.00 Р.М.
4	10.00 А.М.	11.00 А.М.

Порядок сортировки будет $4 \rightarrow 1 \rightarrow 2 \rightarrow 3$ и только четвертая задача будет выполнена, но ответом может быть последовательность действий $1 \rightarrow 3$ или $2 \rightarrow 3$, которые могут быть выполнены. Таким образом, наш подход не будет работать в приведенном выше случае. Давайте попробуем другой подход.

- **Сортировка задач по длительности.** То есть сначала выполняются наименее длительные действия. Это поможет решить предыдущую проблему. Однако сама проблема в целом до конца не решена. Есть еще несколько случаев, в которых такой подход не сработает. Применим этот подход к следующему случаю.

№ задачи	Начало	Конец
1	6.00 А.М.	11.40 А.М.
2	11.30 А.М.	12.00 Р.М.
3	11.40 А.М.	2.00 Р.М.

Если мы отсортируем задания по длительности их выполнения, то получим порядок 2->3->1. И если мы сначала выполним задачу 2, то больше ничего мы выполнить не сможем. Но ответом будет выполнение задачи 1, а затем задачи 3. Таким образом, мы можем выполнить максимум два задания. Так что такой подход не может быть решением данной проблемы. Нам нужно искать другой.

Анализ

- **Сортировка задач по времени их окончания.** Это означает, что та задача, которая заканчивается раньше, располагается первее. Алгоритм приведен ниже.

1. Отсортируйте задания по времени их окончания.
2. Если задача по времени не пересекается с ранее выполненными, то выполните ее.

Давайте проанализируем первый пример

№ задачи	Начало	Конец
1	10.20 А.М.	11.00 А.М.
2	10.30 А.М.	11.30 А.М.
3	11.00 А.М.	12.00 А.М.
4	10.00 А.М.	11.30 А.М.
5	9.00 А.М.	11.00 А.М.

Отсортируем задачи по времени их окончания. Итак, порядок сортировки 1 --> 5 --> 2 --> 4 --> 3. Ответ 1 --> 3 — эти две задачи будут выполнены. И это действительно правильный ответ. Вот алгоритм для построения кода:

1. Сортировать: задачи
2. Выполнить первую задачу из отсортированного списка
3. Текущая задача := первая задача
4. Время окончания := время окончания текущей задачи
5. Перейти к следующей задаче, если она существует. Иначе завершить работу
6. Если время начала текущей задачи <= Время окончания : выполнить задачу и перейти к п. 4
7. Иначе : перейти к п. 5

помощь с кодом: <http://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/>

Раздел 17.3: Проблема выдачи сдачи

При данной денежной системе, можно ли выдать определенную сумму денег и как найти минимальный набор монет, соответствующий заданной сумме?

Канонические денежные системы. Для некоторых денежных систем, таких как те, которые мы используем в реальной жизни, “интуитивное” решение работает идеально. Например, если имеются монеты и банкноты номиналом в 1€, 2€, 5€, 10€, то, выдавая самую крупную монету или купюру, пока не дойдем до нужной суммы, и повторяя данную процедуру, мы получим минимальный набор монет.

Мы можем сделать это рекурсивно на языке OCaml:

```
(* при условии, что денежная система отсортирована в порядке убывания *)
let change_make money_system amount =
  (* устанавливает сдачу после пересчета *)
  let rec loop given amount =
    if amount = 0 then
      given
    else
      (* мы находим первое значение, которое меньше или равно
         оставшемуся количеству *)
      let coin = List.find ((>=) amount) money_system in
      loop (coin::given) (amount - coin)
  in loop [] amount
```

Такие системы сделаны так, что набрать сдачу легко. Проблема усложняется, когда речь заходит о произвольных денежных системах.

Общий случай. Как выдать 99€, имея монеты номиналом 10€, 7€ и 5€? Здесь выдача монет по 10€ до тех пор, пока не останется выдать 9€, очевидно, не приводит ни к какому решению. Хуже того, решение может и не существовать. На самом деле данная задача является NP-трудной, но существуют приемлемые решения, основанные на **жадных алгоритмах** и **мемоизации**. Идея состоит в том, чтобы исследовать все возможные варианты и выбрать тот, в котором имеется минимальное количество монет.

Чтобы выдать сумму $X > 0$, мы выбираем произвольное P , а затем решаем подзадачу для выдачи суммы $X - P$. Мы попробуем сделать это для всех возможных P . Решение, если оно существует, является наименьшим путем, который привел нас к 0.

Вот рекурсивная функция, соответствующая данному методу, написанная на OCaml. Если решения не существует, то она возвращает None.

```
(* вариант утилиты *)
let optmin x y =
  (* если в паре есть один элемент None, то получить соседнее значение *)
  match x, y with
  | None, a | a, None -> a
  | Some x, Some y -> Some (min x y)
```

```

(* инкремент входного аргумента, за исключением None *)
let optsucc = function
| Some x -> Some (x+1)
| None -> None

(* Проблема изменения *)
let change_make money_system amount =
let rec loop n =
  let onepiece acc piece =
    (* если разница долга и сдачи равна 0, то *)
    match n - piece with
    | 0 -> (* проблема решена одной монетой *)
      Some 1
    | x -> if x < 0 then
      (* если не достигаем 0, то отбрасываем это решение *)
      None
    else
      (* ищем наименьший путь, отличный от None с остальными частями *)
      optmin (optsucc (loop x)) acc
  in
  (* называем onepiece для всех частей *)
  List.fold_left onepiece None money_system
in loop amount

```

Примечание: можно заметить, что эта процедура может вычислять набор монет для одного и того же значения несколько раз. На практике, чтобы избежать таких повторений, используется техника мемоизации. Использование такой техники приводит к более быстрому (намного более быстрому) результату.

Глава 18: Применение жадной стратегии

Раздел 18.1: Оффлайн кэширование

Проблема кэширования возникает из-за ограниченности конечного пространства. Предположим, что в нашем кэше C есть k страниц. Теперь мы хотим обработать последовательность m запросов элементов, которые должны быть помещены в кэш перед их обработкой. Конечно, если $m \leq k$, то мы просто поместим все элементы в кэш, и это будет работать, но обычно $m >> k$.

Мы говорим, что запрос является **кэш-попаданием**, когда элемент уже находится в кэше, в противном случае он называется **кэш-промахом**. В этом случае мы должны внести запрошенный элемент в кэш и вытеснить другой, предполагая, что кэш заполнен. Цель - это список вытеснений, который **минимизирует количество вытеснений**.

Существует множество жадных стратегий для решения этой проблемы. Давайте рассмотрим некоторые из них:

1. **Первым пришел - первым вышел (FIFO)**: Самая старая страница вытесняется
2. **Последним пришел - первым вышел (LIFO)**: Самая новая страница вытесняется
3. **Последняя посещаемая страница (LRU)**: Вытесняется страница, последний доступ к которой был самым ранним
4. **Наименее часто запрашиваемая страница (LFU)**: Вытесняется страница, которая запрашивалась реже всех
5. **Самое длинное прямое расстояние (LFD)**: Вытесняется страница, которая не будет запрашиваться дольше всех в будущем

Внимание: В следующих примерах мы вытесняем страницу с наименьшим индексом, если может быть вытеснено больше одной страницы.

Пример (FIFO)

Пусть размер кэша $k=3$, начальный кэш a, b, c и запрос $a, a, d, e, b, b, a, c, f, d, e, a, f, b, e, c$:

Запрос	a a d e b b a c f d e a f b e c
кэш 1	a a d d d a a a d d d f f f b
кэш 2	b b b e e e e c c c e e e b b b
кэш 3	c c c c b b b b f f f a a a e e
кэш-промах	x x x x x x x x x x x x x x x x

Тринадцать кэш-промахов на шестнадцать запросов звучит не очень оптимально, давайте попробуем такой же пример с другой стратегией

Пример (LFD)

Пусть размер кэша $k=3$, начальный кэш a, b, c и запрос $a, a, d, e, b, b, a, c, f, d, e, a, f, b, e, c$:

Запрос	a a d e b b a c f d e a f b e c
кэш 1	a a d e e e e e e e e e e e c
кэш 2	b b b b b b a a a a a a f f f f f
кэш 3	c c c c c c c f d d d b b b
кэш-промах	x x x x x x x x x

Восемь кэш-промахов, уже намного лучше.

Задание: Сделайте пример для LIFO, LFU, RFU и посмотрите, что произойдет.

Следующий пример программы (написанной на C++) состоит из двух частей: Каркас - это программа, которая решает проблему в зависимости от выбранной жадной стратегии:

```
#include <iostream>
#include <memory>

using namespace std;

const int cacheSize = 3;
const int requestLength = 16;

const char request[] = {'a','a','d','e','b','b','a','c','f','d','e','a','f','b','e','c',
                      'a','f','b','e','c'};
char cache[] = {'a','b','c'};

char originalCache[] = {'a','b','c'};

class Strategy {

public:
    Strategy(std::string name) : strategyName(name) {}
    virtual ~Strategy() = default;

    virtual int apply(int requestIndex) = 0;

    virtual void update(int cachePlace, int requestIndex, bool cacheMiss) = 0;

    const std::string strategyName;
};

bool updateCache(int requestIndex, Strategy* strategy)
{
    int cachePlace = strategy->apply(requestIndex);

    bool isMiss = request[requestIndex] != cache[cachePlace];

    strategy->update(cachePlace, requestIndex, isMiss);

    cache[cachePlace] = request[requestIndex];

    return isMiss;
}
```

```

}

int main()
{
    Strategy* selectedStrategy[] = { new FIFO, new LIFO,new LRU,
                                    new LFU, new LFD };

    for (int strat=0; strat < 5; ++strat)
    {

        for (int i=0; i < cacheSize; ++i) cache[i] = originalCache[i];

        cout <<"\nStrategy: " << selectedStrategy[strat]->strategyName << endl;

        cout << "\nCache initial: (";
        for (int i=0; i < cacheSize-1; ++i) cout << cache[i] << ",";
        cout << cache[cacheSize-1] << ") \n\n";

        cout << "Request\t";
        for (int i=0; i < cacheSize; ++i) cout << "cache " << i << "\t";
        cout << "cache miss" << endl;

        int cntMisses = 0;

        for(int i=0; i<requestLength; ++i)
        {

            bool isMiss = updateCache(i, selectedStrategy[strat]);
            if (isMiss) ++cntMisses;
            cout << " " << request[i] << "\t";
            for (int l=0; l < cacheSize; ++l)
            cout << " " << cache[l] << "\t";
            cout << (isMiss ? "x" : "") << endl;
        }

        cout<< "\nTotal cache misses: " << cntMisses << endl;
    }

    for (int i=0; i<5; ++i) delete selectedStrategy[i];
}

```

Основная идея проста: для каждого запроса у меня есть два вызова моей стратегии:

- применить:** Стратегия должна сообщить вызывающей стороне, какую страницу использовать
- обновить:** После того, как вызывающая сторона использует место, она сообщает стратегии, было ли оно пропущено или нет. Тогда стратегия может обновить свои внутренние данные. Стратегия **LFU**, например, должна обновлять частоту попаданий к страницам кэша, в то время как стратегия **LFD** должна пересчитывать расстояние для страниц кэша.

Теперь давайте рассмотрим примеры реализации наших пяти стратегий:

FIFO

```
class FIFO : public Strategy {
public:
    FIFO() : Strategy("FIFO")
    {
        for (int i=0; i<cacheSize; ++i) age[i] = 0;
    }

    int apply(int requestIndex) override
    {
        int oldest = 0;

        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;

            else if(age[i] > age[oldest])
                oldest = i;
        }

        return oldest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        if(!cacheMiss)
            return;
        for(int i=0; i<cacheSize; ++i)
        {
            if(i != cachePos)
                age[i]++;
            else
                age[i] = 0;
        }
    }

private:
    int age[cacheSize];
};
```

FIFO просто нужна информация о том, как долго страница находится в кэше (и, конечно, только относительно других страниц). Так что единственное, что нужно сделать - это дождаться промаха, а затем создать страницы, которые не будут вытеснять старые. Приведём решение для нашего примера, написанного сверху:

Strategy: FIFO

Cache initial: (a,b,c)

Request	cache 0	cache 1	cache 2	cache miss
a	a	b	c	
a	a	b	c	
d	d	b	c	x
e	d	e	c	x
b	d	e	b	x
b	d	e	b	
a	a	e	b	x
c	a	c	b	x
f	a	c	f	x
d	d	c	f	x
e	d	e	f	x
a	d	e	a	x
f	f	e	a	x
b	f	b	a	x
e	f	b	e	x
c	c	b	e	x

Total cache misses: 13

Это и есть решение вышенаписанного.

LIFO

```
class LIFO : public Strategy {
public
    LIFO() : Strategy("LIFO")
    {
        for (int i=0; i<cacheSize; ++i) age[i] = 0;
    }

    int apply(int requestIndex) override
    {
        int newest = 0;
        for(int i=0; i<cacheSize; ++i)
        {

            if(cache[i] == request[requestIndex])
                return i;

            else if(age[i] < age[newest])
                newest = i;
        }

        return newest;
    }

    void update(int cachePos, int requestIndex, bool cacheMiss) override
    {
        if(!cacheMiss)
            return;
    }
}
```

```

    for(int i=0; i<cacheSize; ++i)
    {
        if(i != cachePos)
            age[i]++;
        else
            age[i] = 0;
    }
}

private:
    int age[cacheSize];
};

```

Реализация **LIFO** более или менее аналогична **FIFO**, но мы вытесняем самую младшую, а не самую старую страницу. Результаты программы:

Strategy: LIFO

Cache initial: (a,b,c)

Request	cache 0	cache 1	cache 2	cache miss
a	a	b	c	
a	a	b	c	
d	d	b	c	x
e	e	b	c	x
b	e	b	c	
b	e	b	c	
a	a	b	c	x
c	a	b	c	
f	f	b	c	x
d	d	b	c	x
e	e	b	c	x
a	a	b	c	x
f	f	b	c	x
b	f	b	c	
e	e	b	c	x
c	e	b	c	

Total cache misses: 9

LRU

```

class LRU : public Strategy {
public:
    LRU() : Strategy("LRU")
    {
        for (int i=0; i<cacheSize; ++i) age[i] = 0;
    }
}

```

```

int apply(int requestIndex) override
{
    int oldest = 0;

    for(int i=0; i<cacheSize; ++i)
    {

        if(cache[i] == request[requestIndex])
            return i;

        else if(age[i] > age[oldest])
            oldest = i;
    }
    return oldest;
}

void update(int cachePos, int requestIndex, bool cacheMiss) override
{
    for(int i=0; i<cacheSize; ++i)
    {

        if(i != cachePos)
            age[i]++;
        else
            age[i] = 0;
    }
}

private:
    int age[cacheSize];
};

```

В случае **LRU** стратегия не зависит от того, что находится на странице кэша. Единственным, что ее интересует, является последнее использование. Результат программы:

Strategy: LRU

Cache initial: (a,b,c)

Request	cache 0	cache 1	cache 2	cache miss
a	a	b	c	
a	a	b	c	
d	a	d	c	x
e	a	d	e	x
b	b	d	e	x
b	b	d	e	
a	b	a	e	x
c	b	a	c	x
f	f	a	c	x
d	f	d	c	x
e	f	d	e	x
a	a	d	e	x
f	a	f	e	x
b	a	f	b	x
e	e	f	b	x
c	e	c	b	x

Total cache misses: 13

LFU

```
class LFU : public Strategy {
public:
    LFU() : Strategy("LFU")
    {
        for (int i=0; i<cacheSize; ++i) requestFrequency[i] = 0;
    }

    int apply(int requestIndex) override
    {

        int least = 0;

        for(int i=0; i<cacheSize; ++i)
        {

            if(cache[i] == request[requestIndex])
                return i;

            else if(requestFrequency[i] < requestFrequency[least])
                least = i;
        }

        return least;
    }
}
```

```

void update(int cachePos, int requestIndex, bool cacheMiss) override
{
    if(cacheMiss)
        requestFrequency[cachePos] = 1;

    else
        ++requestFrequency[cachePos];
}

private:
    int requestFrequency[cacheSize];
};

```

LFU вытесняет страницу, которая используется реже всего. Поэтому стратегия обновления заключается просто в подсчете каждого доступа. Конечно, после промаха счет сбрасывается. Результаты программы:

Strategy: LFU

Cache initial: (a,b,c)

Request	cache 0	cache 1	cache 2	cache miss
a	a	b	c	
a	a	b	c	
d	a	d	c	x
e	a	d	e	x
b	a	b	e	x
b	a	b	e	
a	a	b	e	
c	a	b	c	x
f	a	b	f	x
d	a	b	d	x
e	a	b	e	x
a	a	b	e	
f	a	b	f	x
b	a	b	f	
e	a	b	e	x
c	a	b	c	x

Total cache misses: 10

LFD

```

class LFD : public Strategy {
public:
    LFD() : Strategy("LFD")
    {
        for (int i=0; i<cacheSize; ++i)
            nextUse[i] = calcNextUse(-1, cache[i]);
    }
}

```

```

int apply(int requestIndex) override
{
    int latest = 0;

    for(int i=0; i<cacheSize; ++i)
    {
        if(cache[i] == request[requestIndex])
            return i;

        else if(nextUse[i] > nextUse[latest])
            latest = i;
    }
    return latest;
}

void update(int cachePos, int requestIndex, bool cacheMiss) override
{
    nextUse[cachePos] = calcNextUse(requestIndex, cache[cachePos]);
}

private:

int calcNextUse(int requestPosition, char pageItem)
{
    for(int i = requestPosition+1; i < requestLength; ++i)
    {
        if (request[i] == pageItem)
            return i;
    }
    return requestLength + 1;
}

int nextUse[cacheSize];
};

```

Стратегия **LFD** отличается от всех ранее написанных. Это единственная стратегия, которая использует будущие запросы для принятия решения, кого вытеснять. Для реализации используется функция `calcNextUse`, чтобы получить страницу, следующее использование которой в будущем будет наиболее отдалённым. Программное решение равносильно вышеннаписанному решению:

Strategy: LFD

Cache initial: (a,b,c)

Request	cache 0	cache 1	cache 2	cache miss
a	a	b	c	
a	a	b	c	
d	a	b	d	x
e	a	b	e	x
b	a	b	e	
b	a	b	e	
a	a	b	e	
c	a	c	e	x
f	a	f	e	x
d	a	d	e	x
e	a	d	e	
a	a	d	e	
f	f	d	e	x
b	b	d	e	x
e	b	d	e	
c	c	d	e	x

Total cache misses: 8

Жадная стратегия **LFD** действительно является оптимальной стратегией из пяти представленных. Доказательство довольно длинное и может быть найдено [здесь](#) или в книге Джона Клейнберга и Евы Тардос (см. Источники в примечаниях ниже).

Алгоритм против реальности

Стратегия **LFD** является оптимальной, но есть большая проблема. Это оптимальное **оффлайн** решение. На практике кэширования обычно возникают **онлайн**-задачи, это означает, что стратегия бесполезна, потому что мы не можем знать, когда в следующий раз нам понадобится конкретный элемент. Другие четыре стратегии являются **онлайн**-стратегиями. Для онлайн-задач нам нужен другой подход.

Раздел 18.2: Автомат билетов

Первый простой пример:

У вас есть автомат по продаже билетов, который даёт возможность обменивать монеты со значениями 1, 2, 5, 10 и 20. Раздачу можно рассматривать как серию падающих монет до тех пор, пока не будет выдано нужное значение. Мы говорим, что раздача является **оптимальной**, когда **количество монет минимально** для её стоимости.

Пусть M из интервала $[1, 50]$ - цена за билет T , а P из интервала $[1, 50]$ - деньги, которые кто-то заплатил за T , $P \geq M$. Пусть $D = P - M$. Мы определяем **выгоду** сделанного шага как разницу между D и $D - c$, где c - монета из автомата, выданная на этом шаге.

Жадная техника обмена - это следующий псевдо алгоритмический подход:

- Шаг 1: пока $D > 20$ выдать монету со значением 20 и установить $D = D - 20$
- Шаг 2: пока $D > 10$ выдать монету со значением 10 и установить $D = D - 10$
- Шаг 3: пока $D > 5$ выдать монету со значением 5 и установить $D = D - 5$

Шаг 4: пока $D > 2$ выдать монету со значением 2 и установить $D = D - 2$

Шаг 5: пока $D > 1$ выдать монету со значением 1 и установить $D = D - 1$

После этого сумма всех монет явно равна D . Это **жадный алгоритм**, потому что после каждого шага и после каждого повторения шага выгода максимизируется. Мы не можем выдать другую монету и получить большую выгоду.

Реализация автомата билетов на C++:

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

std::vector<unsigned int> readInCoinValues();

int main()
{
    std::vector<unsigned int> coinValues;
    int ticketPrice;
    int paidMoney;

    coinValues = readInCoinValues();

    cout << "ticket price: ";
    cin >> ticketPrice;

    cout << "money paid: ";
    cin >> paidMoney;

    if(paidMoney <= ticketPrice)
    {
        cout << "No exchange money" << endl;
        return 1;
    }

    int diffValue = paidMoney - ticketPrice;

    std::vector<unsigned int> coinCount;

    for(auto coinValue = coinValues.begin();
        coinValue != coinValues.end(); ++coinValue)
    {
        int countCoins = 0;
        while (diffValue >= *coinValue)
        {
            diffValue -= *coinValue;
            countCoins++;
        }
        coinCount.push_back(countCoins);
    }
}
```

```

cout << "the difference " << paidMoney - ticketPrice
    << " is paid with: " << endl;
for(unsigned int i=0; i < coinValues.size(); ++i)
{
    if(coinCount[i] > 0)
        cout << coinCount[i] << " coins with value "
            << coinValues[i] << endl;
}
return 0;
}

std::vector<unsigned int> readInCoinValues()
{
    std::vector<unsigned int> coinValues;

    coinValues.push_back(1);

    while(true)
    {

        int coinValue;

        cout << "Coin value (<1 to stop): ";
        cin >> coinValue;

        if(coinValue > 0)
            coinValues.push_back(coinValue);

        else
            break;
    }

    sort(coinValues.begin(), coinValues.end(), std::greater<int>());
    auto last = std::unique(coinValues.begin(), coinValues.end());
    coinValues.erase(last, coinValues.end());

    cout << "Coin values: ";

    for(auto i : coinValues)
        cout << i << " ";
    cout << endl;
    return coinValues;
}

```

Имейте в виду, что теперь есть проверка ввода, чтобы упростить пример. Пример вывода:

```

Coin value (<1 to stop): 2
Coin value (<1 to stop): 4
Coin value (<1 to stop): 7
Coin value (<1 to stop): 9
Coin value (<1 to stop): 14
Coin value (<1 to stop): 4
Coin value (<1 to stop): 0
Coin values: 14 9 7 4 2 1
ticket price: 34
money paid: 67
the difference 33 is paid with:
2 coins with value 14
1 coins with value 4
1 coins with value 1

```

Пока есть монета со значением 1, мы знаем, что алгоритм будет завершен, потому что:

- D строго уменьшается с каждым шагом
- D никогда не будет >0 и меньше, чем самая маленькая монета 1, одновременно

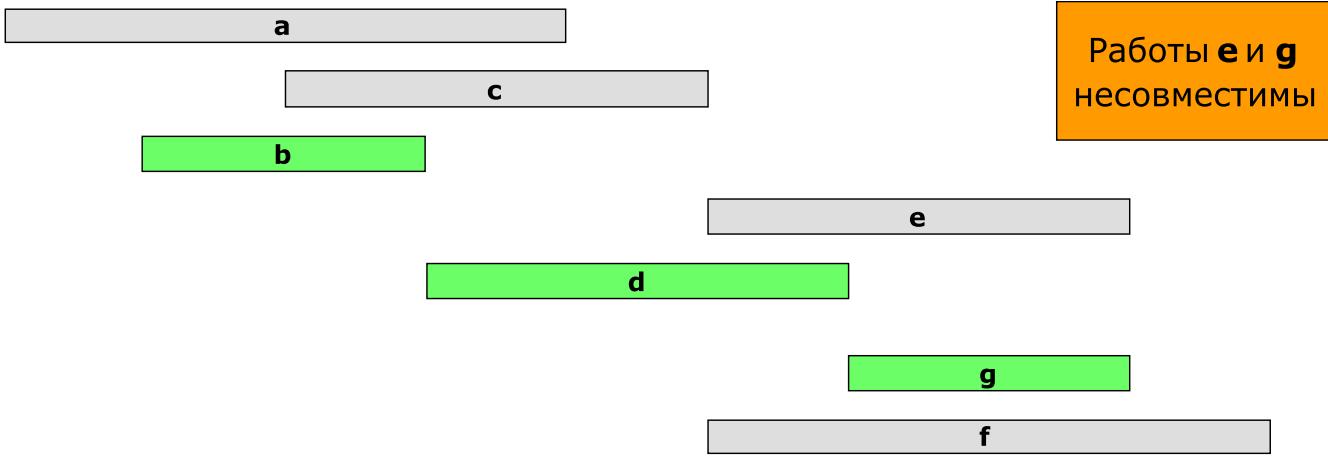
Но алгоритм имеет два подводных камня:

1. Пусть C будет наибольшим значением монеты. Время исполнения является полиномиальным только до тех пор, пока D/C является полиномиальным, потому что представление D использует только $\log D$ бит, а в D/C время исполнения является по меньшей мере линейным.
2. На каждом этапе наш алгоритм выбирает локальный оптимум. Но этого недостаточно, чтобы сказать, что алгоритм находит глобальное оптимальное решение (см. больше информации [здесь](#) или в книге [Корти и Вайджена](#)).

Простой пример счётчика: монеты 1,3,4 и $D=6$. Оптимальным решением, безусловно, являются две монеты номиналом 3, но жадный выбирает 4 на первом шаге, поэтому он должен выбрать 1 на втором и третьем шагах. Так что это не даёт оптимального решения. Возможный оптимальный алгоритм для данного примера основан на [динамическом программировании](#).

Раздел 18.3: Интервальное планирование

У нас есть набор работ $J=a,b,c,d,e,f,g$. Пусть j из J будет работой, которая начинается в s_j и заканчивается в f_j . Две работы совместимы, если они не пересекаются. Пример:



Цель состоит в том, чтобы найти **максимальное подмножество взаимно совместимых работ**. Есть несколько жадных подходов к этой проблеме:

1. **Ранний старт**: Рассматриваем задания в порядке возрастания s_j
2. **Раннее завершение**: Рассматриваем задания в порядке возрастания f_j
3. **Кратчайший интервал**: Рассматриваем задания в порядке возрастания $f_j - s_j$
4. **Наименьшее количество конфликтов**: Для каждого задания j подсчитать количество конфликтующих заданий c_j

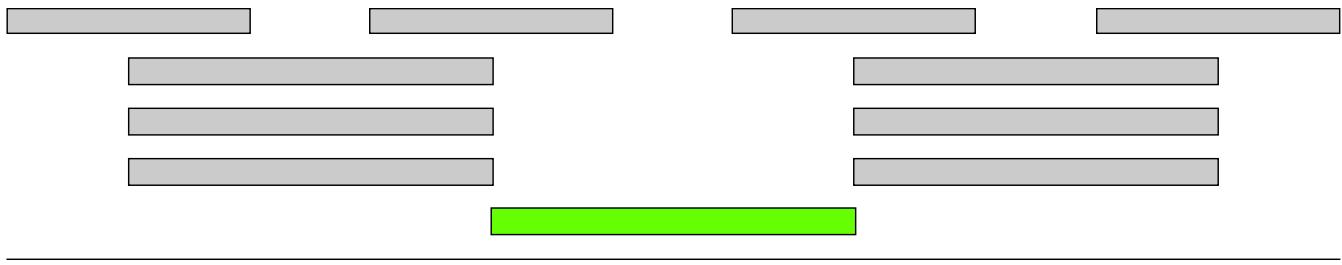
Вопрос сейчас состоит в том, какой подход действительно успешен. **Ранний старт** - определенно нет, вот контрпример:



Подход кратчайшего интервала также не является самым оптимальным,



в то время как **наименее конфликтный подход** действительно может показаться оптимальнее, однако существует пример, показывающий обратное:



Что вынуждает нас использовать подход **Наименьших временных затрат**. Псевдокод достаточно прост:

1. Отсортировать задачи по времени окончания так, чтобы $f_1 \leq f_2 \leq \dots \leq f_n$
2. Пусть A — пустое множество
3. для j от 1 до n : если j совместимо со **всеми** задачами во множестве A , тогда множество $A = A + j$
4. множество A — это **максимальное подмножество взаимно совместимых задач**

Или программа на C++:

```
#include <iostream>
#include <utility>
#include <tuple>
#include <vector>
#include <algorithm>
const int jobCnt = 10;
const int startTimes[] = { 2, 3, 1, 4, 3, 2, 6, 7, 8, 9};
const int endTimes[] = { 4, 4, 3, 5, 5, 5, 8, 9, 9, 10};
using namespace std;
int main()
{
    vector<pair<int,int>> jobs;
    for(int i=0; i<jobCnt; ++i)
        jobs.push_back(make_pair(startTimes[i], endTimes[i]));
    sort(jobs.begin(), jobs.end(), [] (pair<int,int> p1, pair<int,int> p2)
         { return p1.second < p2.second; });
    vector<int> A;
    for(int i=0; i<jobCnt; ++i)
    {
        auto job = jobs[i];
        bool isCompatible = true;
        for(auto jobIndex : A)
        {
            if(job.second >= jobs[jobIndex].first &&
               job.first <= jobs[jobIndex].second)
            {
                isCompatible = false;
                break;
            }
        }
        if(isCompatible)
            A.push_back(i);
    }
}
```

```

        }
    }
    if(isCompatible)
        A.push_back(i);
}
cout << "Compatible: ";
for(auto i : A)
    cout << "(" << jobs[i].first << "," << jobs[i].second << ")";
cout << endl;
return 0;
}

```

Вывод для этого примера: Compatible: (1,3) (4,5) (6,8) (9,10)

Явно выражена сложность реализованного алгоритма — $\theta(n^2)$. Также здесь представлена реализация за $\theta(n \log n)$ (пример на языке Java), заинтересованный читатель может прочесть его ниже.

Теперь у нас есть жадный алгоритм для интервального планирования, но оптимален ли он?

Утверждение: Самое раннее время завершения жадного алгоритма — оптимальное.

Доказательство (от обратного):

Допустим жадный алгоритм не оптимален и i_1, i_2, \dots, i_k обозначают множество всех выбранных жадным алгоритмом задач. Пусть j_1, j_2, \dots, j_m обозначают множество в **оптимальном** решении при $i_1 = j_1, i_2 = j_2, \dots, i_r = j_r$ для **наибольшего** возможного значения r .

Задача $i_{(r+1)}$ существует и заканчивается до $j_{(r+1)}$ (самое раннее выполнение). Но $j_1, j_2, \dots, j_r, i_{(r+1)}, j_{(r+2)}, \dots, j_m$ также **оптимальное** решение и для всех k в промежутке от 1 до $r+1$ включительно выполнено $j_k = i_k$. Это **противоречит** условию максимизации r . Что и следовало доказать.

Второй пример демонстрирует то, что обычно существует много способов задания жадного алгоритма, но только некоторые могут найти оптимальное решение в каждом примере или, что вполне возможно, ни один не сможет найти оптимальное решение к данной задаче в каком-либо примере.

Ниже представлена программа на языке Java, которая работает за $\theta(n \log n)$

```

import java.util.Arrays;
import java.util.Comparator;
class Job
{
    int start, finish, profit;
    Job(int start, int finish, int profit)
    {
        this.start = start;
        this.finish = finish;
        this.profit = profit;
    }
}

```

```

    }
}

class JobComparator implements Comparator<Job>
{
    public int compare(Job a, Job b)
    {
        return a.finish < b.finish ? -1 : a.finish == b.finish ? 0 : 1;
    }
}
public class WeightedIntervalScheduling
{
    static public int binarySearch(Job jobs[], int index)
    {
        int lo = 0, hi = index - 1;
        while (lo <= hi)
        {
            int mid = (lo + hi) / 2;
            if (jobs[mid].finish <= jobs[index].start)
            {
                if (jobs[mid + 1].finish <= jobs[index].start)
                    lo = mid + 1;
                else
                    return mid;
            }
            else
                hi = mid - 1;
        }
        return -1;
    }
    static public int schedule(Job jobs[])
    {
        Arrays.sort(jobs, new JobComparator());
        int n = jobs.length;
        int table[] = new int[n];
        table[0] = jobs[0].profit;
        for (int i=1; i<n; i++)
        {
            int inclProf = jobs[i].profit;
            int l = binarySearch(jobs, i);
            if (l != -1)
                inclProf += table[l];
            table[i] = Math.max(inclProf, table[i-1]);
        }
        return table[n-1];
    }
    public static void main(String[] args)
    {
        Job jobs[] = {new Job(1, 2, 50), new Job(3, 5, 20),
                     new Job(6, 19, 100), new Job(2, 100, 200)};
        System.out.println("Optimal profit is " + schedule(jobs));
    }
}

```

}

И ожидаемый вывод:

Optimal profit is 250

Раздел 18.4: Минимизация задержки

Существует множество проблем минимизации задержки, в данном случае мы имеем один исполнитель, который может обрабатывать только одну задачу в одно и то же время. Задание j требует единиц времени обработки t_j и должно быть выполнено в момент времени d_j . Если задача j начнется в момент времени s_j , тогда время окончания обработки будет равно $f_j = s_j + t_j$. Мы объявляем задержку, как $L = \max(0, f_j - d_h)$ для всех j . Цель состоит в том, чтобы минимизировать максимальную задержку L .

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	10	11

Задача	3	2	2	5	5	5	4	4	4	4	1	1	1	6	6
Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
l_j	-8	-5		-4					1			7		4	

Очевидно, что решение $L=7$ не является оптимальным. Давайте посмотрим некоторые способы задания жадного алгоритма:

1. **Первоочередный вызов с наименьшим временем обработки:** спланировать задачи по возрастанию времени обработки j
2. спланировать задачи по возрастанию срока окончания задачи d_j
3. **Первоочередный вызов с наименьшей разницей:** спланировать задачи по возрастанию разницы $d_j - t_j$

Легко заметить, что **первоочередный вызов с наименьшим временем обработки** неоптимальный, показательным примером для этого является

	1	2
t_j	1	5
d_j	10	5

решение **первоочередного вызова с наименьшей разницей** имеет схожие недостатки

1	2
t_j	1 5
d_j	3 5

последняя стратегия похожа на правильную, поэтому давайте начнём с псевдо-кода:

1. Отсортировать n задач по времени так, чтобы $d_1 \leq d_2 \leq \dots \leq d_n$
2. Положим $t = 0$
3. Для j от 1 до n :
 - Назначить задание j на интервал $[t, t + t_j]$
 - Положим $s_j = t$ и $f_j = t + t_j$
 - Положим $t = t + t_j$
4. Возвратить интервалы $[s_1, f_1], [s_2, f_2], \dots, [s_n, f_n]$

И реализация на C++:

```
#include <iostream>
#include <utility>
#include <tuple>
#include <vector>
#include <algorithm>
const int jobCnt = 10;
const int processTimes[] = { 2, 3, 1, 4, 3, 2, 3, 5, 2, 1};
const int dueTimes[] = { 4, 7, 9, 13, 8, 17, 9, 11, 22, 25};
using namespace std;
int main()
{
    vector<pair<int,int>> jobs;
    for(int i=0; i<jobCnt; ++i)
        jobs.push_back(make_pair(processTimes[i], dueTimes[i]));
    sort(jobs.begin(), jobs.end(), [] (pair<int,int> p1, pair<int,int> p2)
         { return p1.second < p2.second; });
    int t = 0;
    vector<pair<int,int>> jobIntervals;
    for(int i=0; i<jobCnt; ++i)
    {
        jobIntervals.push_back(make_pair(t,t+jobs[i].first));
        t += jobs[i].first;
    }
    cout << "Intervals:\n" << endl;
    int lateness = 0;
    for(int i=0; i<jobCnt; ++i)
    {
        auto pair = jobIntervals[i];
        lateness = max(lateness, pair.second-jobs[i].second);
        cout << "(" << pair.first << "," << pair.second << ")"
    }
}
```

```

    << "Lateness: " << pair.second-jobs[i].second << std::endl;
}
cout << "\nmaximal lateness is " << lateness << endl;
return 0;
}

```

И вывод этой программы:

```

Intervals:
(0,2) Lateness:-2
(2,5) Lateness:-2
(5,8) Lateness: 0
(8,9) Lateness: 0
(9,12) Lateness: 3
(12,17) Lateness: 6
(17,21) Lateness: 8
(21,23) Lateness: 6
(23,25) Lateness: 3
(25,26) Lateness: 1
maximal lateness is 8

```

Рабочий цикл алгоритма очевидно имеет сложность $\theta(n \log n)$, потому что сортировка является преобладающей операцией алгоритма. Сейчас нам нужно показать, что он оптимальный. Три-виально у самой оптимальной планировки **нет времени простоя**. **Первоочередный вызов с наиболее ранним дедлайном** также не имеет времени простоя.

Давайте предполагать, что задачи пронумерованы так, что $d_1 \leq d_2 \leq \dots \leq d_n$. Мы говорим, что **инверсия** расписания — это пара заданий i и j таких, что $i < j$, но j запланировано раньше i . Следуя этому определению **первоочередный вызов с наиболее ранним дедлайном** не имеет инверсий. Конечно, если расписание имеет инверсию, то оно также имеет пару обращенных к друг другу задач (inverted jobs), запланированных последовательно.

Утверждение: Замена двух смежных, обращенных друг к другу задач уменьшает количество инверсий **на одно и не увеличивает** максимальную задержку.

Доказательство: пусть L будет задержкой перед заменой, а M после. Т.к. обмен двумя соседними задачами не сдвигает другие задачи с их позиций, то для всех $k \neq i, j$ выполняется равенство $L_k = M_k$.

Очевидно, что $M_i \leq L_i$, т.к. задача i была запланирована раньше. Если задача j запаздывает, то следуем определению:

$$\begin{aligned}
M_j &= f_i - d_j \text{ (definition)} \\
&\leq f_i - d_i \text{ (since } i \text{ and } j \text{ are exchanged)} \\
&\leq L_i
\end{aligned}$$

Это означает, что задержка после перестановки меньше или равна той, что была до неё. Что и следовало доказать.

Утверждение: Первочередный вызов с наиболее ранним дедлайном является оптимальной стратегией S .

Доказательство(от обратного):

Предположим, S^* это оптимальное расписание с **наименьшим возможным** количеством инверсий. Мы можем предположить, что у S^* нет времени простоя. Если у S^* нет инверсий, тогда $S = S^*$ и проблема решена. Если у S^* есть инверсия, следовательно, в данном расписании имеется смежная инверсия. В последнем утверждении говорится, что мы можем менять местами соседние инверсии, не увеличивая задержку, но уменьшая эти самые инверсии. Это противоречит определению S^* .

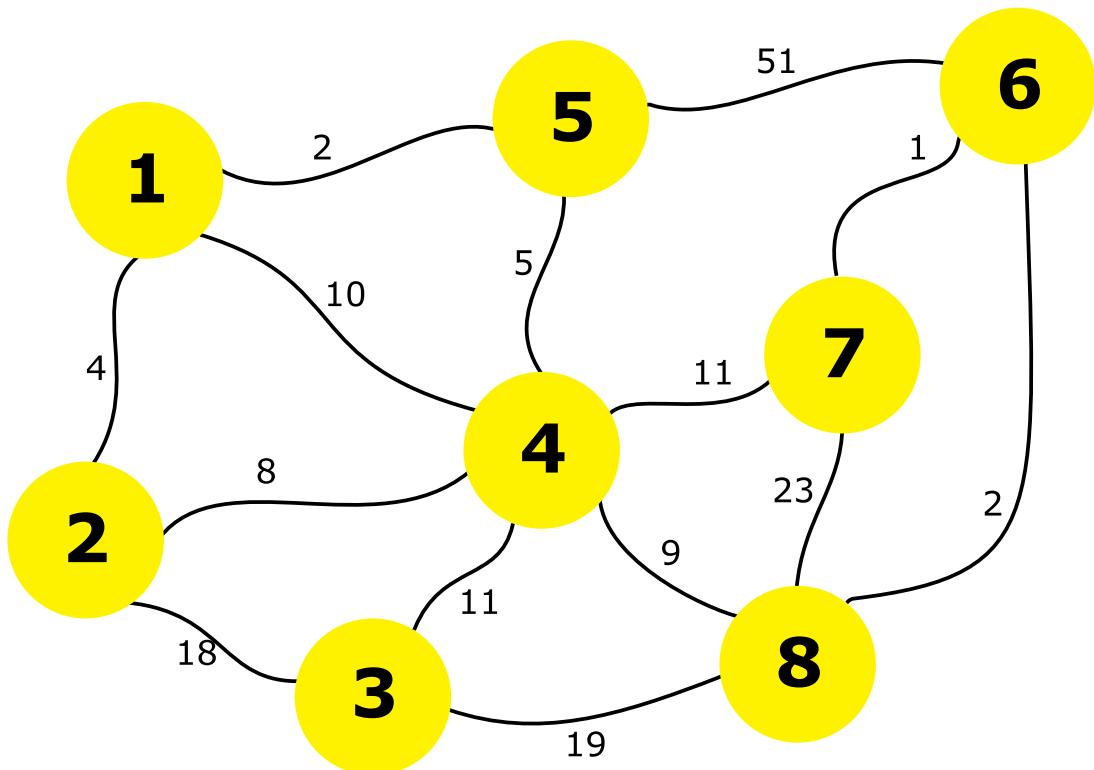
Проблема минимизации задержки и смежные ей проблемы минимизации суммарного времени выполнения, где задается вопрос о минимальном сроке, имеет множество применений в реальном мире. Но обычно вы работаете не с одной машиной, и всё множество машин обрабатывают одинаковую задачу с разной скоростью. Эти NP-полные проблемы решаются очень быстро.

Еще один интересный вопрос возникает, если не рассматривать проблему в **автономном режиме**, где у нас есть все задачи и данные при себе, а рассматривать как способ поступления данных **онлайн**, где задачи появляются во время работы программы.

Глава 19: Алгоритм Прима

Раздел 19.1: Введение в алгоритм Прима

Допустим, у нас есть 8 домов. Вы хотим установить телефонные линии между этими домами. Ребро между домами представляет из себя стоимость установки телефонной линии между этими двумя домами.



Наша задача установить линии таким способом, чтобы все дома были соединены, и чтобы стоимость всей установки связи была минимальная. И как мы решим эту проблему? Мы можем использовать **алгоритм Прима**.

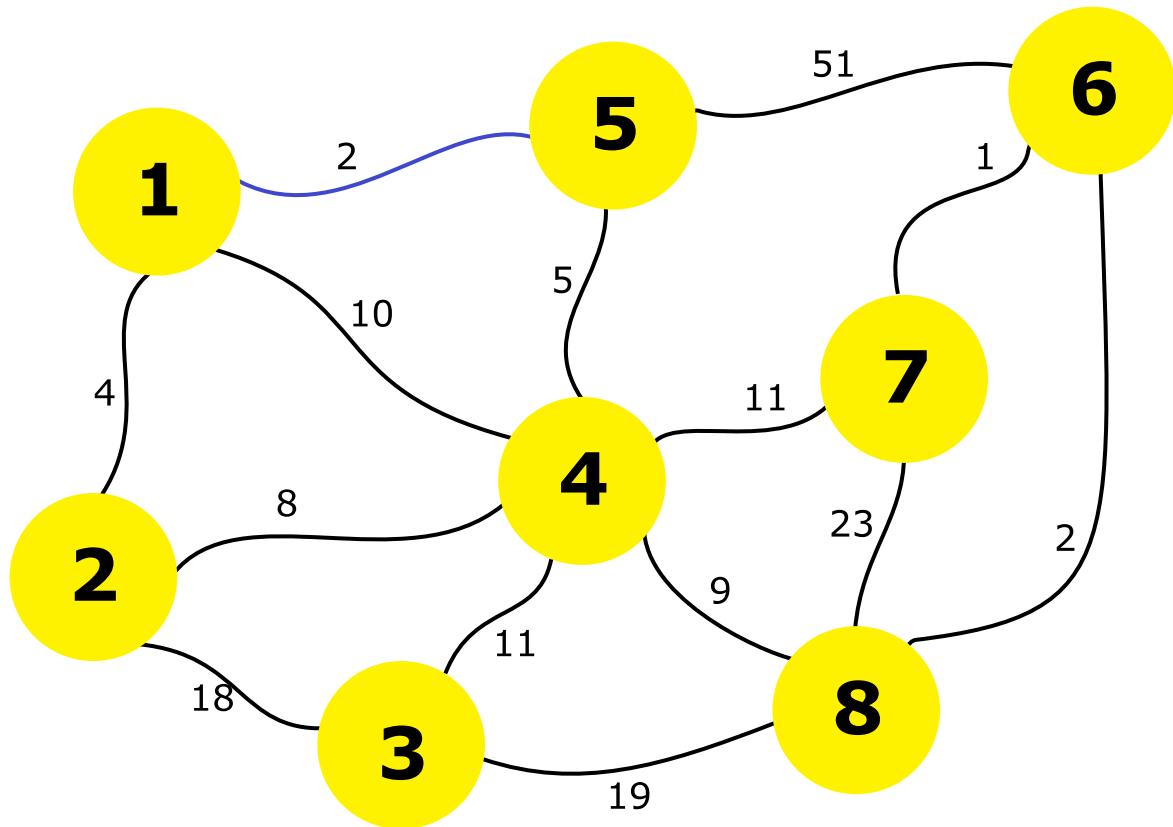
Алгоритм Прима это жадный алгоритм, который находит минимальное остворное дерево для взвешенного неориентированного графа. Это означает, что он находит подмножество ребер, таких, что формируют дерево, которое включает каждый узел, где общий вес всех ребер в дереве минимизирован. Алгоритм был разработан в 1930 году чешским математиком Ярником Войтехомом и затем переоткрыт и заново опубликован учёным компьютерных наук Робертом Примом в 1957 году и Эдсгером Вибе Дейкстрой в 1959. Этот алгоритм также известен, как алгоритм DJP, алгоритм Ярника, алгоритм Прим-Ярника или алгоритм Прим-Дейкстры.

А теперь первым делом давайте посмотрим на техническую составляющую алгоритма. Если мы создаем граф S , используя некоторые узлы и ребра неориентированного графа G , тогда S называется **подграфом** графа G . Здесь и далее S будет называться **остовным деревом** если и только если:

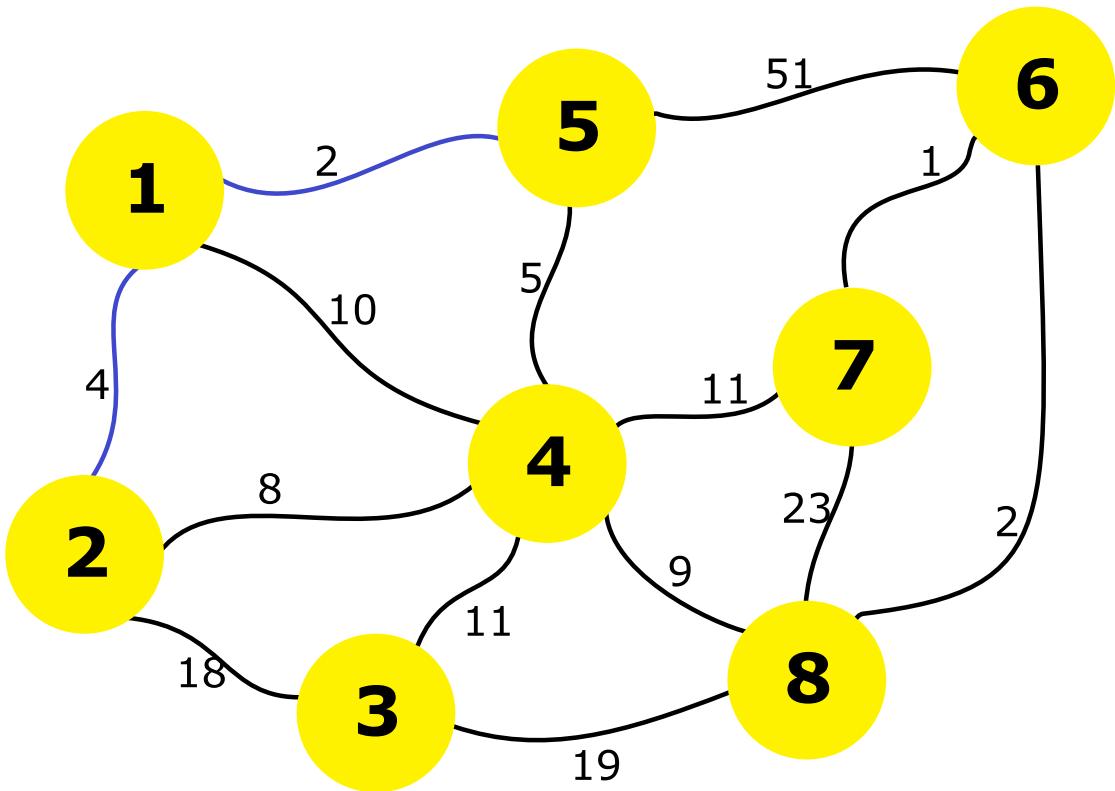
- Он содержит все узлы графа G .
- Это дерево, что значит, что в нем нет циклов и все узлы соединены между собой.
- В дереве ровно $(n - 1)$ узлов, где n это количество узлов в G .

В графе может быть несколько **остовных деревьев**. **Минимальное оствовное дерево** взвешенного неориентированного графа — это такое дерево, что сумма весов на ребрах минимальна. Сейчас мы воспользуемся **алгоритмом Прима**, чтобы найти минимальное оствовное дерево, или же найдем способ настроить телефонные линии в нашем графе-примере таким образом, чтобы стоимость установки была минимальной.

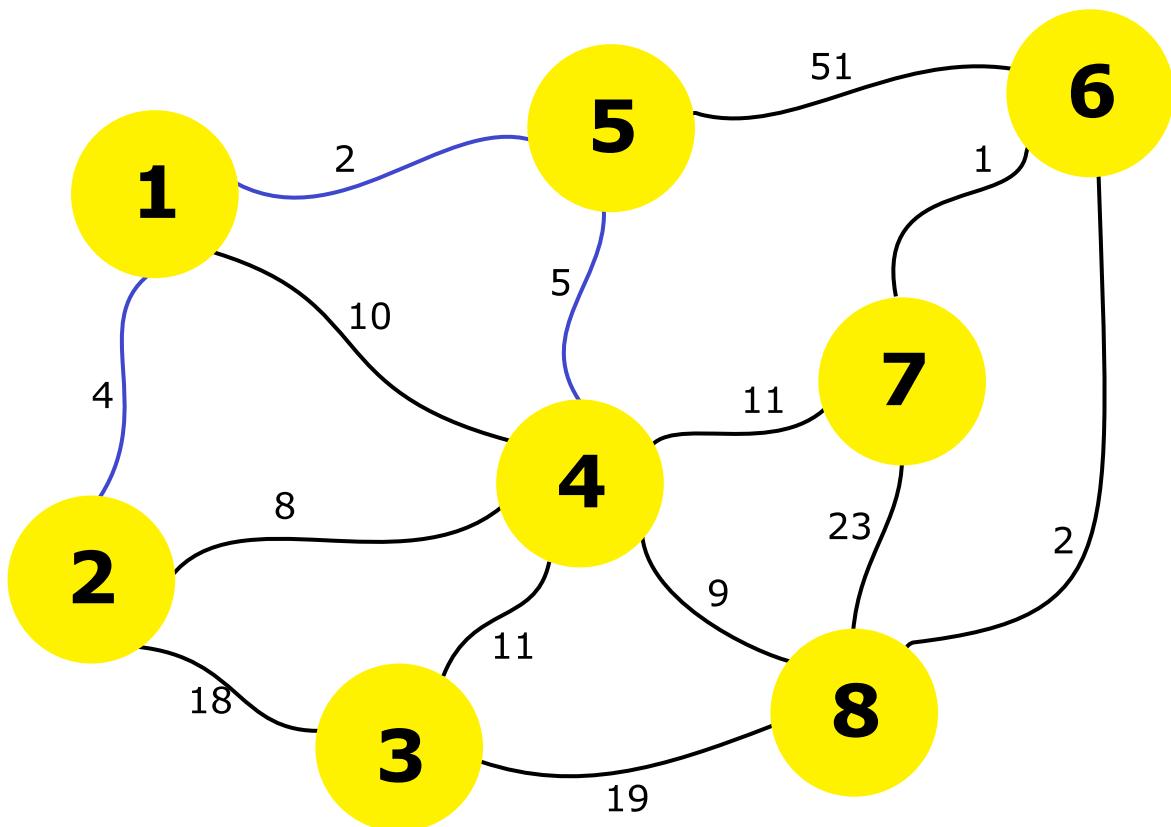
Для начала вы выберем узел-**источник**. Допустим, что **node-1** это наш **источник**. Теперь мы добавим ребро из **node-1** так, чтобы к нашему подграфу добавилась как можно меньший вес. Тут мы отметим ребро, которое находится в подграфе, используя **синий** цвет. Здесь **1-5** это наше желаемое ребро.



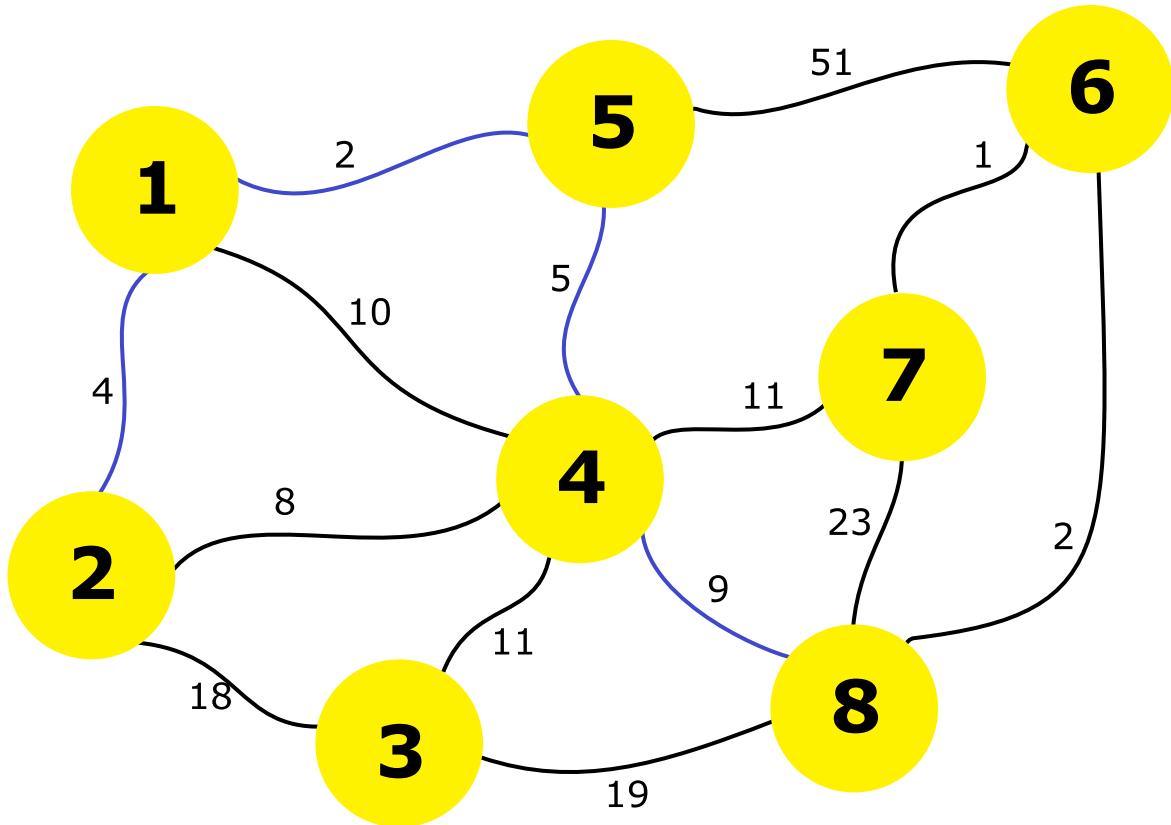
Теперь мы рассмотрим все ребра от узла 1 и узла 5 и возьмем минимум. Так как 1-5 уже отмечены, мы возьмем 1-2.



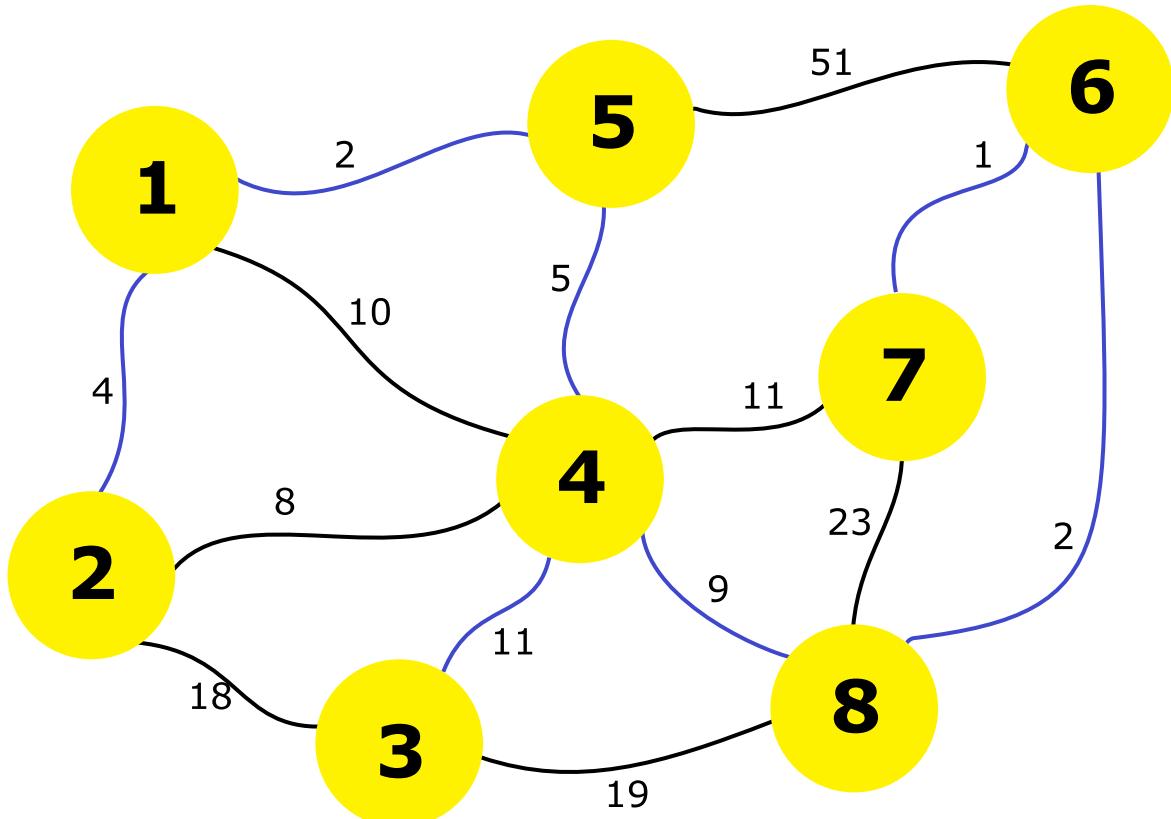
На этот раз мы рассмотрим узел 1, узел 2 и узел 5 и возьмем минимальное ребро 5-4.



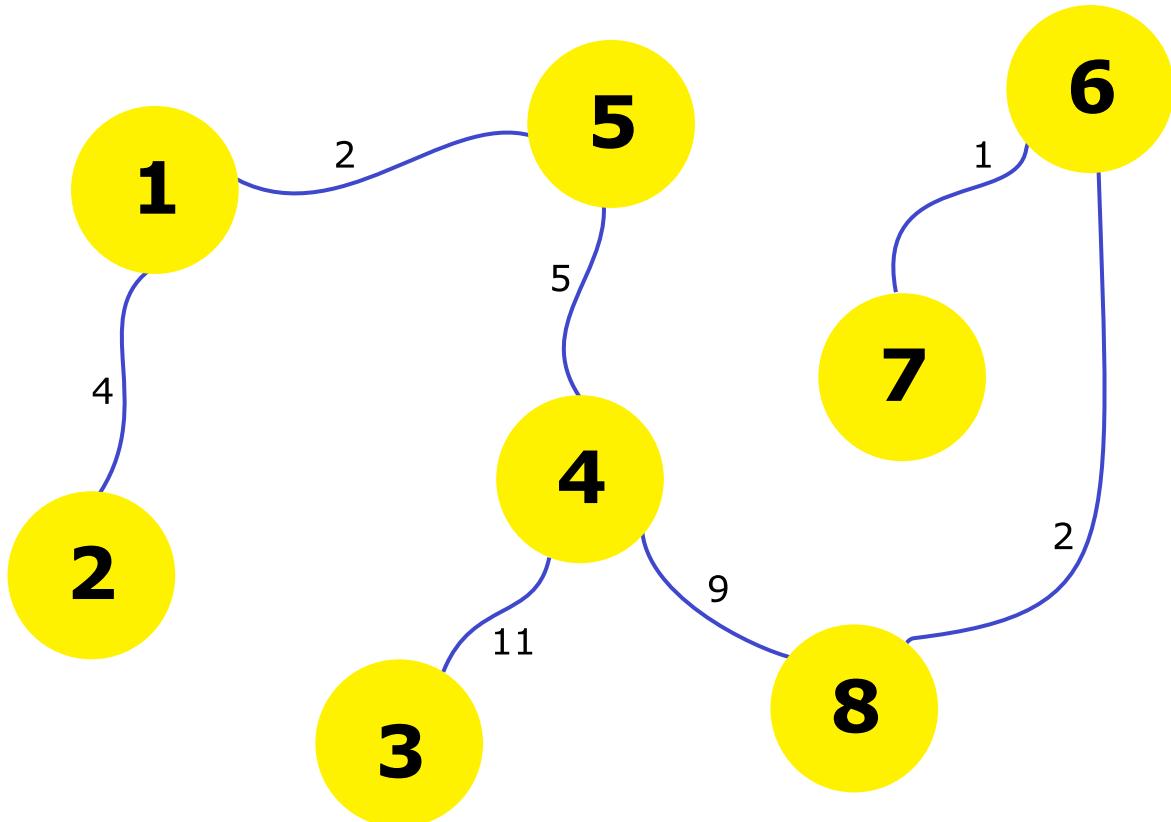
Следующий шаг важен. От **узла-1**, **узла-2**, **узла-5** и **узла-4** наименьшее ребро - это **2-4**. Но если мы выберем его, то создадим цикл в нашем подграфе. Это из-за того, что **узел-2** и **узел-4** уже находятся в нашем подграфе. Таким образом, добавление ребра **2-4** не принесет нам никакой пользы. *Мы выберем ребра такие, чтобы они добавили новый узел в наш подграф.* Поэтому мы выберем ребро **4-8**.



Если мы продолжим тем же образом, то выберем ребра 8-6, 6-7 и 4-3. Наш подграф будет выглядеть так:



Это подграф, который нам нужен, который даст нам дерево с минимальным охватом. Если мы уберем ребра, которые не выбирали, то получим:



Это наше **дерево с минимальным охватом** (MST). Таким образом, стоимость настройки телефонных соединений: $4 + 2 + 5 + 11 + 9 + 2 + 1 = 34$. А набор домов и их связей показаны на граfe. На граfe может быть несколько MST. Это зависит от выбранного нами **исходного** узла.

Ниже приведен псевдокод алгоритма:

```

Procedure PrimsMST(Graph):
Vnew[] = {x}
Enew[] = {}
while Vnew is not equal to V
    u -> a node from Vnew
    v -> a node that is not in Vnew such that edge u-v has the minimum cost
    add v to Vnew
    add edge (u, v) to Enew
end while
Return Vnew and Enew

```

Сложность:

Временная сложность вышеуказанного наивного подхода – $O(V^2)$. В нем используется матрица смежности. Мы можем уменьшить сложность, используя очередь приоритетов. Когда мы добавляем новый узел к V_{new} , то мы можем добавить его соседние ребра в очередь с приоритетами. Затем вытаскиваем из него ребро с минимальным весом. Тогда сложность будет: $O(E \log E)$, где E – это количество ребер. Также можно построить двоичную кучу, чтобы уменьшить сложность до $O(E \log V)$.

Псевдокод, использующий очередь с приоритетами, приведен ниже:

```

Procedure MSTPrim(Graph, source):
for each u in V
    key[u] := inf
    parent[u] := NULL
end for
key[source] := 0
Q = Priority_Queue()
Q = V
while Q is not empty
    u -> Q.pop
    for each v adjacent to i
        if v belongs to Q and Edge(u,v) < key[v]
            parent[v] := u
            key[v] := Edge(u, v)
        end if
    end for
end while

```

Здесь **key[]** хранит минимальную стоимость обхода **node-v**. **parent[]** используется для хранения родительского узла. Это полезно для обхода и вывода дерева.

Ниже приведена простая программа на Java:

```

import java.util.*;

public class Graph
{
    private static int infinite = 9999999;
    int[][] LinkCost;
    int NNodes;
    Graph(int[][] mat)
    {
        int i, j;
        NNodes = mat.length;
        LinkCost = new int[NNodes][NNodes];
        for ( i=0; i < NNodes; i++)
        {
            for ( j=0; j < NNodes; j++)
            {
                LinkCost[i][j] = mat[i][j];
                if (LinkCost[i][j] == 0)
                {
                    LinkCost[i][j] = infinite;
                }
            }
        }
        for ( i = 0; i < NNodes; i++)
        {
            for (j = 0; j < NNodes; j++)
                if (LinkCost[i][j] <infinite)
                    System.out.print( " " + LinkCost[i][j] + " " );
                else
                    System.out.print(" * ");
        }
    }
}

```

```

        System.out.print();
    }
}

public int unReached(boolean[] r)
{
    boolean done = true;
    for (int i = 0; i < r; i++)
        if (r[i] == false)
            return i;
    return -1;
}
public void Prim( )
{
    int i, j, k, x, y;
    boolean[] Reached = new boolean[NNodes];
    Reached[0] = true;
    for (k = 1; k < NNodes; k++)
    {
        Reached[k] = false;
    }
    predNode[0] = 0;
    printReachSet( Reached);
    for(k=1; k < NNodes; k++)
    {
        x= y=0;
        for( i=0; i< NNodes; i++)
            for( j=0; j< NNodes; j++)
            {
                if (Reached[i] &&!Reached[j] &&
                    LinkCost[i][j] < LinkCost[x][y])
                {
                    x= i;
                    y= j;
                }
            }
        System.out.println("Min cost edge:
                           (" + x + "," + y + ")" + "cost = " + LinkCost[x][y]);
        predNode[y]= x;
        Reached[y]= true;
        printReachSet( Reached);
        System.out.println();
    }
    int[] a= predNode;
    for( i=0; i < NNodes; i++)
        System.out.println( a[i]+ " --> "+ i);
}
void printReachSet(boolean[] Reached)
{
    System.out.print("ReachSet = ");
    for(int i=0; i< Reached.length; i++)
        if( Reached[i])

```

```

        System.out.print( i+ " ");
    }
    public static void main(String[] args)
    {
        int[][] conn={{0,3,0,2,0,0,0,0,4}, // 0
                      {3,0,0,0,0,0,0,4,0}, // 1
                      {0,0,0,6,0,1,0,2,0}, // 2
                      {2,0,6,0,1,0,0,0,0}, // 3
                      {0,0,0,1,0,0,0,0,8}, // 4
                      {0,0,1,0,0,0,8,0,0}, // 5
                      {0,0,0,0,0,8,0,0,0}, // 6
                      {0,4,2,0,0,0,0,0,0}, // 7
                      {4,0,0,0,8,0,0,0,0} // 8
                    };
        Graph G= new Graph(conn);
        G.Prim();
    }
}

```

Скомпилируем код выше, используя `javac Graph.java`

На выходе получим:

```

$ java Graph
* 3 * 2 * * * * 4
3 * * * * * 4 *
* * * 6 * 1 * 2 *
2 * 6 * 1 * * * *
* * * 1 * * * * 8
* * 1 * * * 8 * *
* * * * * 8 * * *
* 4 2 * * * * * *
4 * * * 8 * * * *

ReachSet = 0 Min cost edge: (0,3)cost = 2
ReachSet = 0 3
Min cost edge: (3,4)cost = 1
ReachSet = 0 3 4
Min cost edge: (0,1)cost = 3
ReachSet = 0 1 3 4
Min cost edge: (0,8)cost = 4
ReachSet = 0 1 3 4 8
Min cost edge: (1,7)cost = 4
ReachSet = 0 1 3 4 7 8
Min cost edge: (7,2)cost = 2
ReachSet = 0 1 2 3 4 7 8
Min cost edge: (2,5)cost = 1
ReachSet = 0 1 2 3 4 5 7 8
Min cost edge: (5,6)cost = 8
ReachSet = 0 1 2 3 4 5 6 7 8
0 --> 0
0 --> 1

```

```
7 --> 2
0 --> 3
3 --> 4
2 --> 5
5 --> 6
1 --> 7
0 --> 8
```

Глава 20: Алгоритм Беллмана-Форда

Раздел 20.1: Алгоритм поиска кратчайшего пути от одной вершины

Перед прочтением данного примера необходимо иметь краткое представление о релаксации ребер графа. Вы можете узнать об этом отсюда

Алгоритм [Беллмана-Форда](#) вычисляет кратчайшие пути из одной исходной вершины до всех остальных вершин во взвешенном графе. Несмотря на то, что он медленнее, чем алгоритм Дейкстры, он работает в тех случаях, когда вес ребра отрицательный, а также находит в графе цикл с отрицательным весом. Проблема с алгоритмом Дейкстры в том, что если имеется отрицательный цикл, то он продолжает проходить через цикл снова и снова и продолжает сокращать расстояние между двумя вершинами.

Идея данного алгоритма заключается в том, чтобы пройти все ребра этого графа одно за другим в некотором случайном порядке. Это может быть любой случайный порядок. Но вы должны убедиться, что если $u-v$ (где u и v - две вершины графа) является одним из ваших порядков, то должно иметься ребро, соединяющее u и v . Обычно оно берется непосредственно из порядка, заданного входными данными. Опять же, любой случайный порядок будет работать.

После выбора порядка, мы *релаксируем* ребра в соответствии с формулой релаксации. Для данного ребра $u-v$, проходя от u до v , формула следующая:

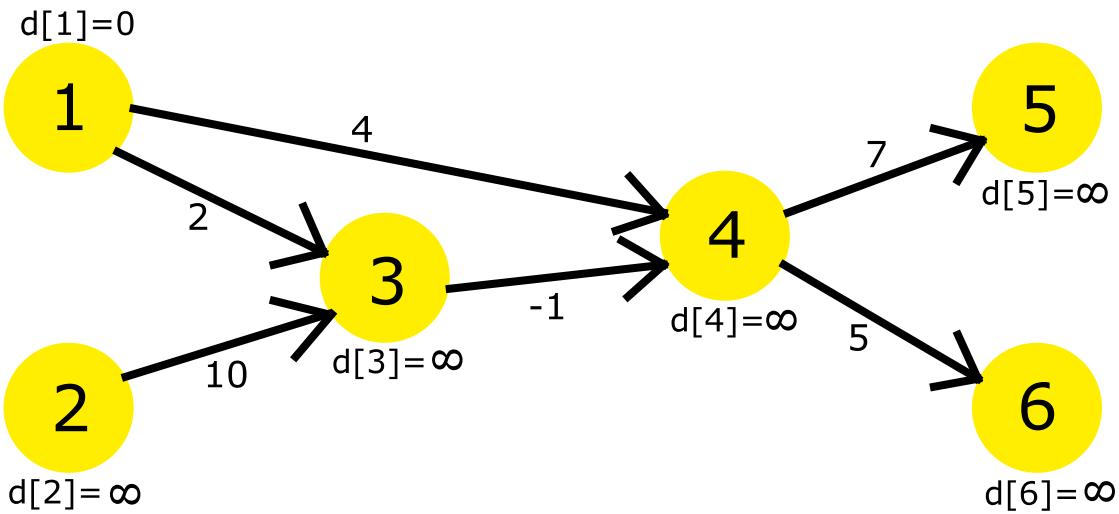
```
if distance[u] + cost[u][v] < d[v]
    d[v] = d[u] + cost[u][v]
```

То есть, если расстояние от **источника** до любой вершины u + вес **ребра** $u-v$ меньше, чем расстояние от источника до другой вершины v , мы обновляем расстояние от **источника** до v . Нам нужно *релаксировать* ребра не более чем ($V-1$) раз, где V - число ребер графа. Вы спросите, почему ($V-1$)? Мы объясним это в другом примере. Также мы будем отслеживать родительскую вершину любой вершины, т.е. когда мы релаксируем ребро, мы установим:

```
parent[v] = u
```

Это означает, что мы нашли еще один короткий путь для достижения v через u . Он понадобится нам позже, чтобы вывести кратчайший путь от **источника** к нужной вершине.

Давайте рассмотрим пример. У нас есть граф:



В качестве **исходной** вершины мы выбрали **1**. Мы хотим найти кратчайший путь от **источника** до всех остальных вершин.

Сначала $d[1] = 0$, потому что это источник. А остальные - *бесконечность*, потому что мы еще не знаем их расстояния.

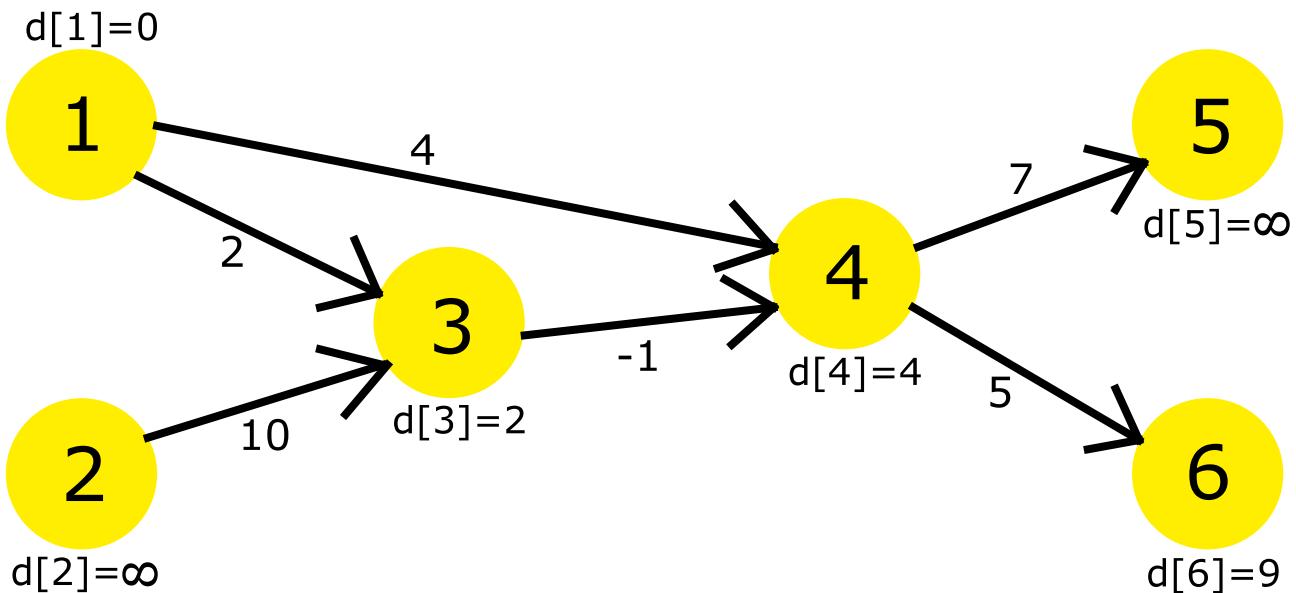
Мы будем релаксировать ребра в этой последовательности:

Serial	1	2	3	4	5	6
Edge	4->5	3->4	1->3	1->4	4->6	2->3

Вы можете взять любую последовательность. Если мы *прорелаксируем* ребра один раз, то что мы получим? Мы получим расстояние от **источника** до всех остальных вершин пути, которые используют максимум 1 ребро. Теперь давайте ослабим ребра и обновим значения $d[]$. Мы получим:

1. $d[4] + \text{cost}[4][5] = \text{infinity} + 7 = \text{infinity}$. Мы не можем обновить это значение.
2. $d[2] + \text{cost}[3][4] = \text{infinity}$. Мы не можем обновить это значение.
3. $d[1] + \text{cost}[1][3] = 0 + 2 = 2 < d[2]$. Поэтому $d[3] = 2$. Также $\text{parent}[1] = 1$.
4. $d[1] + \text{cost}[1][4] = 4$.
5. $d[4] + \text{cost}[4][6] = 9$. $d[6] = 9 < d[6]$. $\text{parent}[6] = 4$.
6. $d[2] + \text{cost}[2][3] = \text{infinity}$. Мы не можем обновить это значение.

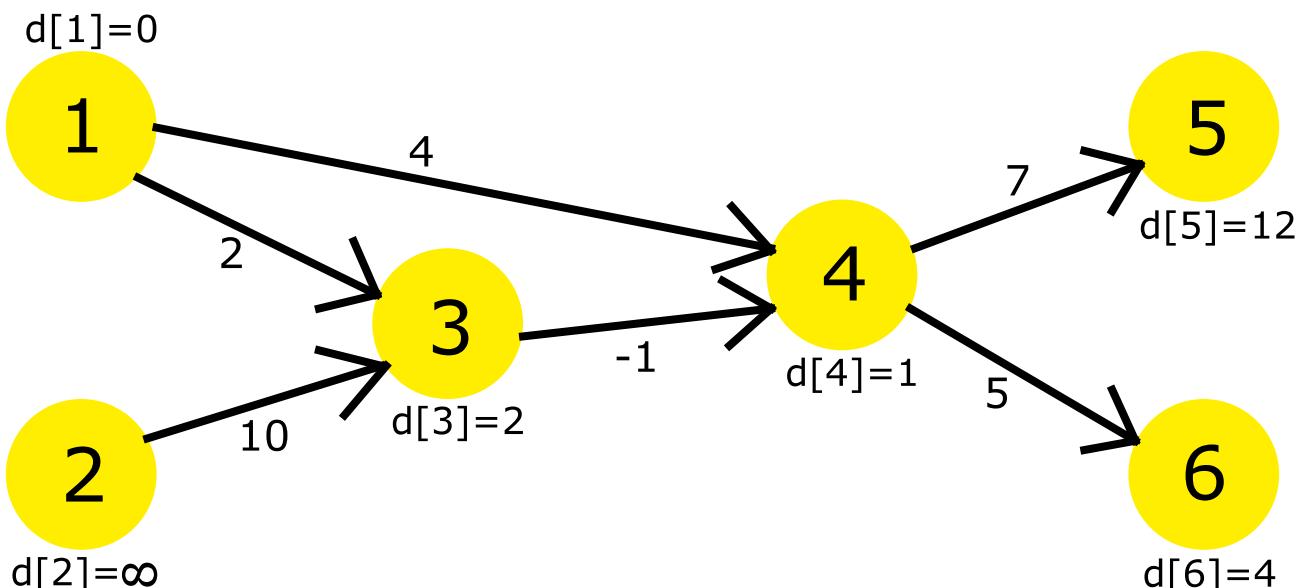
Мы не смогли обновить некоторые вершины, потому что условие $d[u] + \text{cost}[u][v] < d[v]$ не совпадало. Как мы уже говорили ранее, мы нашли пути от **исходных** вершин к другим узлам, используя максимум 1 ребро.



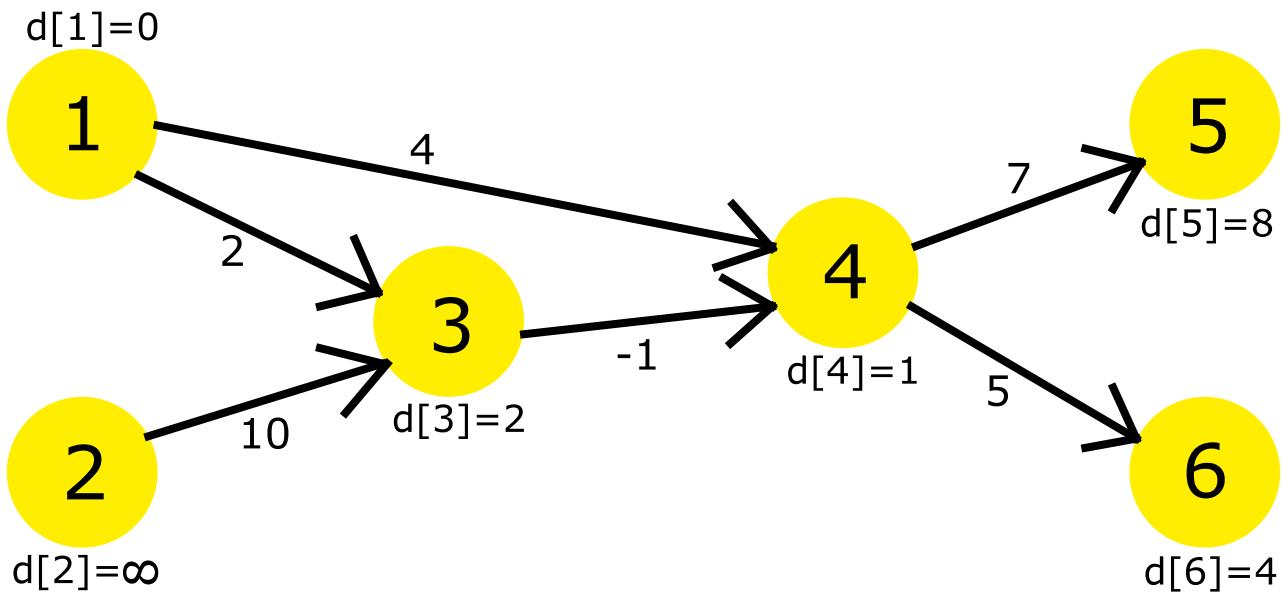
Вторая итерация предоставит нам путь с использованием 2 узлов. Мы получим:

1. $d[4] + \text{cost}[4][5] = 12 < d[5]$. $d[5] = 12$. $\text{parent}[5] = 4$.
2. $d[3] + \text{cost}[3][4] = 1 < d[4]$. $d[4] = 1$. $\text{parent}[4] = 3$.
3. $d[3]$ остается без изменений.
4. $d[4]$ остается без изменений.
5. $d[4] + \text{cost}[4][6] = 6 < d[6]$. $d[6] = 6$. $\text{parent}[6] = 4$.
6. $d[3]$ остается без изменений.

Наш граф будет выглядеть так:



Наша 3-я итерация обновит только **вершину 5**, где $d[5]$ будет 8. Наш граф будет выглядеть так:



После этого, сколько бы итераций мы ни делали, у нас будут одинаковые расстояния. Поэтому у нас будет флаг, который проверяет, происходит обновление или нет. Если нет, то мы просто прервем цикл. Наш псевдокод будет выглядеть так:

```

Procedure Bellman-Ford(Graph, source):
n := number of vertices in Graph
for i from 1 to n
    d[i] := infinity
    parent[i] := NULL
end for
d[source] := 0
for i from 1 to n-1
    flag := false
    for all edges from (u,v) in Graph
        if d[u] + cost[u][v] < d[v]
            d[v] := d[u] + cost[u][v]
            parent[v] := u
            flag := true
        end if
    end for
    if flag == false
        break
    end for
Return d

```

Чтобы отслеживать отрицательные циклы, мы можем модифицировать наш код, используя описанную здесь процедуру. Наш конечный псевдокод будет выглядеть так:

```

Procedure Bellman-Ford-With-Negative-Cycle-Detection(Graph, source):
n := number of vertices in Graph
for i from 1 to n

```

```

d[i] := infinity
parent[i] := NULL
end for
d[source] := 0
for i from 1 to n-1
    flag := false
    for all edges from (u,v) in Graph
        if d[u] + cost[u][v] < d[v]
            d[v] := d[u] + cost[u][v]
            parent[v] := u
            flag := true
        end if
    end for
    if flag == false
        break
end for
for all edges from (u,v) in Graph
    if d[u] + cost[u][v] < d[v]
        Return "Negative Cycle Detected"
    end if
end for
Return d

```

Вывод пути:

Чтобы вывести кратчайший путь до вершины, мы будем возвращаться назад к ее родителю до тех пор, пока не встретим **NULL**, и затем выведем вершины. Псевдокод будет выглядеть так:

```

Procedure PathPrinting(u)
v := parent[u]
if v == NULL
    return
PathPrinting(v)
print -> u

```

Сложность:

Так как нам необходимо ослабить ребра максимум ($V-1$) раз, то времененная сложность этого алгоритма будет равна $O(V * E)$, где E - количество ребер, если мы используем список смежности для отображения графа. Однако, если для представления графа используется матрица смежности, то времененная сложность будет равна $O(V^3)$. Причина в том, что при использовании списка смежности мы можем выполнить итерацию по всем ребрам во времени $O(E)$, но при использовании матрицы смежности требуется время $O(V^2)$.

Раздел 20.2. Обнаружение отрицательного цикла в графе

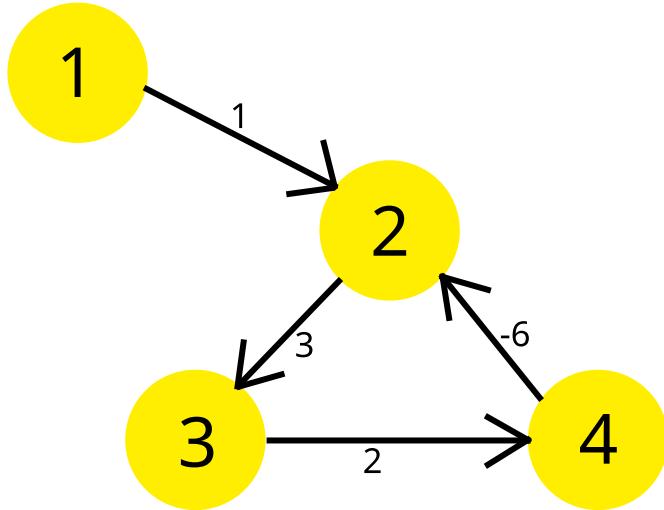
Чтобы понять этот пример, рекомендуется иметь краткое представление об алгоритме Беллмана-Форда, которое можно найти здесь

Используя алгоритм Беллмана-Форда, мы можем определить, есть ли отрицательный цикл на нашем графике. Мы знаем, что для того, чтобы узнать кратчайший путь, нам нужно ослабить все ребра графа ($V-1$) раз, где V - количество вершин в графе. Мы уже видели, что в этом примере после ($V-1$) итераций мы не можем обновить $d[]$, независимо от того, сколько

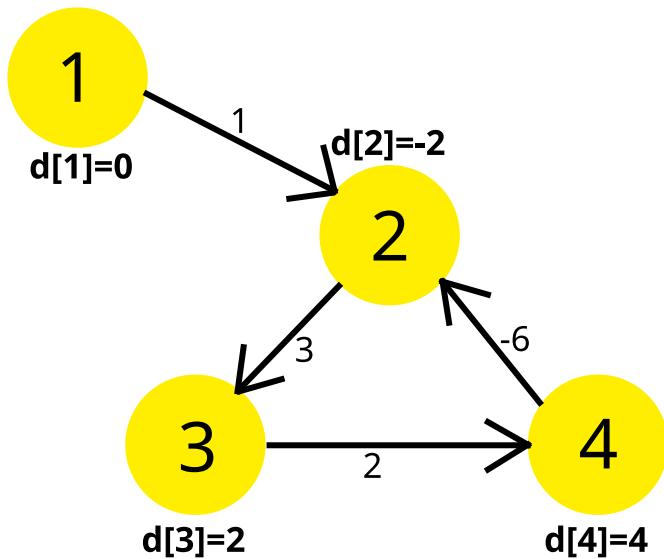
итерации мы делаем. Или можем?

Если в графе есть отрицательный цикл, даже после $(V - 1)$ итераций, мы можем обновить $d[]$. Это происходит потому, что для каждой итерации, проходящей через отрицательный цикл, всегда уменьшается стоимость кратчайшего пути. Вот почему алгоритм Беллман-Форда ограничивает количество итераций до $(V - 1)$. Если бы мы использовали алгоритм Дейкстры здесь, мы бы застряли в бесконечном цикле. Однако давайте сосредоточимся на поиске отрицательного цикла.

Предположим, у нас есть график:



Давайте выберем **вершину 1** в качестве **источника**. После применения алгоритма кратчайшего пути Беллмана-Форда к графу, мы узнаем расстояния от **источника** до всех остальных вершин.



Так выглядит график после $(V - 1) = 3$ итераций. Это должно быть результатом, так как есть 4 ребра, нам нужно больше 3 итераций, чтобы узнать кратчайший путь. Так что либо это ответ, либо есть отрицательный цикл в графе. Чтобы выяснить это, после $(V - 1)$ итераций мы делаем еще одну заключительную итерацию и если расстояние продолжает уменьшаться, то это означает, что на графике, определенно, отрицательный весовой цикл.

Для этого примера: если мы проверим $2 - 3$, $d[2] + \text{cost}[2][3]$ даст нам 1, что меньше, чем $d[3]$. Таким образом, мы можем сделать вывод, что на нашем графике есть отрицательный цикл.

Итак, как мы можем узнать отрицательный цикл? Мы немного изменим процедуру Беллмана-Форда:

```
Procedure NegativeCycleDetector(Graph, source):
n := number of vertices in Graph
# задаем начальные бесконечные расстояния до вершин из source
for i from 1 to n
    # d - массив расстояний от source
    d[i] := infinity
end for
# Очевидно, что от source до source расстояние 0

d[source] := 0
# Улучшаем расстояние до других вершин. Сделаем n-1 итераций
for i from 1 to n-1
    # если flag - true, то мы нашли меньшее расстояние до другой вершины
    flag := false
    # Проходим по всем ребрам
    for all edges from (u,v) in Graph
        # Если расстояние от вершины u до v меньше, чем было - обновим
        # расстояние на более оптимальное
        if d[u] + cost[u][v] < d[v]
            d[v] := d[u] + cost[u][v]
            # изменения есть
            flag := true
        end if
    end for
    # А если изменений не было - нет смысла прогонять ещё итерации
    if flag == false
        break
    end for
# Проведем последнюю итерацию
# Если расстояния до других вершин из source уменьшились
# То это укажет на наличие отрицательного цикла
for all edges from (u,v) in Graph
    if d[u] + cost[u][v] < d[v]
        # Уменьшение расстояний произошло
        Return "Negative Cycle Detected"
    end if
end for
Return "No Negative Cycle"
```

Вот как мы узнаем, существует ли отрицательный цикл в графе. Мы также можем изменить алгоритм Беллмана-Форда, чтобы сохранить путь отрицательных циклов.

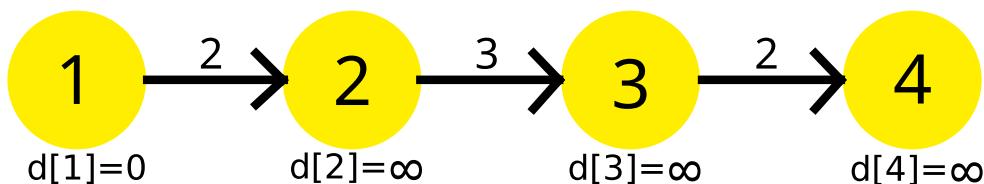
Раздел 20.3: Почему нам нужно ослабить все ребра в ($V-1$) раз

Чтобы понять этот пример, рекомендуется иметь краткое представление об алгоритме Беллмана-Форда с одним источником кратчайшего пути который можно найти здесь.

В алгоритме Беллмана-Форда, чтобы найти кратчайший путь, нам нужно ослабить все ребра графа. Этот процесс повторяется не более ($V-1$) раз, где V - количество вершин в графе.

Количество итераций, необходимое для определения кратчайшего пути от **источника** ко всем остальным вершинам, зависит от порядка, который мы выбираем, чтобы ослабить ребра.

Давайте посмотрим на пример:



Здесь **исходная** вершина равна 1. Выясним кратчайшее расстояние между **источником** и всеми остальными вершинами. Мы можем ясно видеть, что для достижения **вершины 4** в худшем случае потребуется ($V - 1$) ребер. Теперь в зависимости от доступности, какие ребра обнаружены, это может занять ($V - 1$) раз, чтобы обнаружить вершину 4. Не поняли? Давайте использовать алгоритм Беллман-Форда поиска кратчайшего пути здесь:

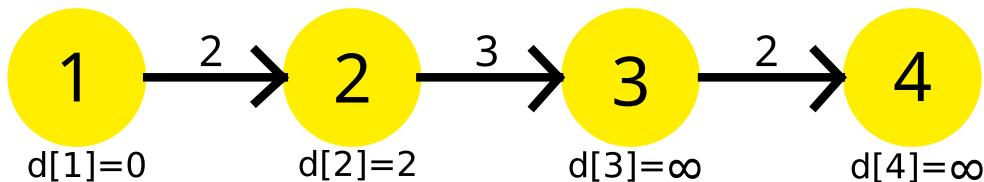
Мы собираемся использовать эту последовательность:

	1	2	3
Номер	1	2	3
Грань	3->4	2->3	1->2

Для нашей первой итерации:

1. $d[3] + \text{cost}[3][4] = \text{infinity}$. Это ничего не изменит.
2. $d[2] + \text{cost}[2][3] = \text{infinity}$. Это ничего не изменит.
3. $d[1] + \text{cost}[1][2] = 2 < d[2]$. $d[2] = 2$. $\text{parent}[2] = 1$.

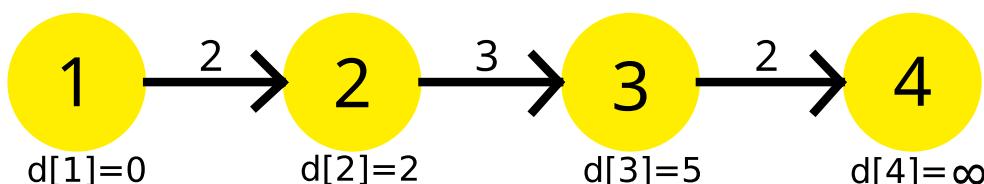
Мы видим, что наш процесс ослабления изменился только в $d[2]$. Наш график будет выглядеть так:



Вторая итерация:

1. $d[3] + \text{cost}[3][4] = \text{infinity}$. Это ничего не изменит.
2. $d[2] + \text{cost}[2][3] = 5 < d[3]$. $d[3] = 5$. $\text{parent}[3] = 2$.
3. Это не будет изменено.

На этот раз процесс ослабления изменился на $d[3]$. Наш график будет выглядеть так:



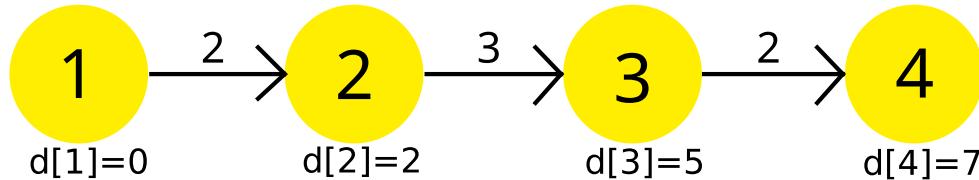
Третья итерация:

1. $d[3] + \text{cost}[3][4] = 7 < d[4]$. $d[4] = 7$. $\text{parent}[4] = 3$.

2. Это не будет изменено.

3. Это не будет изменено.

Наша третья итерация наконец нашла кратчайший путь до **4** из **1**. Наш график будет выглядеть так:



Итак, потребовалось **3** итерации, чтобы найти кратчайший путь. После этого, сколько бы раз мы ни *расслабляли* края, значения в $d[]$ останутся прежними. Теперь, если мы рассмотрим другую последовательность:

Номер	1	2	3
Грань	1->2	2->3	3->4

Мы бы получили:

1. $d[1] + \text{cost}[1][2] = 2 < d[2]$. $d[2] = 2$.

2. $d[2] + \text{cost}[2][3] = 5 < d[3]$. $d[3] = 5$.

3. $d[3] + \text{cost}[3][4] = 7 < d[4]$. $d[4] = 5$.

Наша самая первая итерация нашла кратчайший путь от **источника** ко всем остальным узлам. Другая последовательность **1-> 2, 3-> 4, 2-> 3**, возможно, это даст нам кратчайший путь после **2** итераций. Мы можем прийти к решению, независимо от того, как мы упорядочим последовательность, это займет не более **3** итераций, чтобы найти кратчайший путь от **источника** в этом примере.

Мы можем заключить, что в лучшем случае потребуется **1** итерация, чтобы найти кратчайший путь от **источника**. Для худшего случая потребуется **(V-1)** итераций, поэтому мы повторяем процесс ослабления **(V-1)** раз.

Глава 21: Линейный алгоритм

Рисование линии осуществляется путем расчета промежуточных положений вдоль пути между двумя указанными конечными позициями. Затем устройство вывода направляется для заполнения этих позиций между конечными точками.

Раздел 21.1: Алгоритм рисования линий Брезенхэма

Теория фона: алгоритм рисования линий Брезенхэма - это эффективная и точная генерация растровых линий алгоритмом, разработанным Брезенхэмом. Он включает в себя только целочисленные вычисления, поэтому он точный и быстрый. Он также может быть расширен для отображения кругов разных характеристик.

Алгоритм рисования линий Брезенхэма:

При наклоне $|m| < 1$:

Либо значение x увеличивается

ИЛИ оба значения x и y увеличиваются с помощью параметра решения.

При наклоне $|m| > 1$:

Либо значение y увеличивается

ИЛИ оба значения x и y увеличиваются с помощью параметра решения.

Алгоритм для наклона $|m| < 1$:

1. Ввести две точки: (x_1, y_1) и (x_2, y_2)
2. Нарисовать точку (x_1, y_1) : $\text{plot}(x_1, y_1)$.

3. Вычислить

$$\text{Delx} = |x_2 - x_1|$$

$$\text{Dely} = |y_2 - y_1|$$

4. Получить начальный параметр решения:

$$p = 2 * \text{dely} - \text{delx}$$

5. Для i от 0 до delx с шагом 1:

Если $p < 0$, то

$$x_1 = x_1 + 1$$

$\text{plot}(x_1, y_1)$

$$p = p + 2\text{dely}$$

Иначе,

$$x_1 = x_1 + 1$$

$$y_1 = y_1 + 1$$

$\text{plot}(x_1, y_1)$

$$p = p + 2\text{dely} - 2 * \text{delx}$$

Конец Если

Конец Для

6. Конец

Исходный код на Си:

```
/* Программа на Си с реализацией алгоритма Брезенхэма для рисования линий для |m|<1 */
#include<stdio.h>
#include<conio.h>
#include <graphics.h>
#include <math.h>
int main()
{
    int gdriver=DETECT,gmode;
    int x1,y1,x2,y2,delx,dely,p,i;
    // Инициализируем график
    initgraph(&gdriver,&gmode,"C:\\TC\\BGI");
    printf("Enter the initial points: ");
    scanf("%d",&x1);
    scanf("%d",&y1);
    printf("Enter the end points: ");
    scanf("%d",&x2);
    scanf("%d",&y2);
    // Рисуем начальную точку красным цветом
    putpixel(x1,y1,RED);
    // Вычисляем расстояние по Ox и Oy от начальной точки до конечной
    delx=fabs(x2-x1);
    dely=fabs(y2-y1);
    // Вычисляем параметр решения
    p=(2*dely)-delx;
    // Проходим по всем x от 0 до delx.
    // Не по dely, потому что наклон |m|<1, то есть наклон пологий
    for(i=0;i<delx;i++){
        // В зависимости от параметра p рисуем пиксель
        // либо по Ox, либо еще со смещением по Oy
        if(p<0)
        {
            x1=x1+1;
            putpixel(x1,y1,RED);
            p=p+(2*dely);
        }
        else
        {
            x1=x1+1;

            y1=y1+1;
            putpixel(x1,y1,RED);
            p=p+(2*dely)-(2*delx);
        }
    }
    // Задержка графика, график закроется нажатием любой клавиши
    getch();
    // Закрытие графика. Окончание программы
    closegraph();
    return 0;
}
```

Алгоритм для наклона $|m|>1$:

1. Ввести две точки: (x_1, y_1) и (x_2, y_2)
2. Нарисовать точку (x_1, y_1) : $\text{plot}(x_1, y_1)$.
3. Вычислить
$$\text{Delx} = |x_2 - x_1|$$
$$\text{Dely} = |y_2 - y_1|$$
4. Получить начальный параметр решения:
$$p = 2 * \text{delx} - \text{dely}$$
5. Для i от 0 до $dely$ с шагом 1:
Если $p < 0$, то
 $x_1 = x_1 + 1$
 $\text{plot}(x_1, y_1)$
 $p = p + 2\text{delx}$
Иначе,
 $x_1 = x_1 + 1$
 $y_1 = y_1 + 1$
 $\text{plot}(x_1, y_1)$
 $p = p + 2\text{delx} - 2 * \text{dely}$
Конец Если
Конец Для
6. Конец

Исходный код на Си:

```
/* Программа на Си с реализацией алгоритма Брезенхэма для рисования линий для |m|<1 */
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<math.h>
int main()
{
    int gdriver=DETECT,gmode;

    int x1,y1,x2,y2,delx,dely,p,i;
    // Инициализируем график
    initgraph(&gdriver,&gmode,"C:\\TC\\BGI");
    printf("Enter the intial points: ");
    scanf("%d",&x1);
    scanf("%d",&y1);
    printf("Enter the ent points: ");
    scanf("%d",&x2);
    scanf("%d",&y2);
    // Рисуем начальную точку красным цветом
    putpixel(x1,y1,RED);
    // Вычисляем расстояние по Ox и Oy от начальной точки до конечной
    delx=fabs(x2-x1);
    dely=fabs(y2-y1);
    // Вычисляем параметр решения
    p=(2*delx)-dely;
    // Проходим по всем y от 0 до dely.
```

```

// Не по delx, потому что наклон |m|>1, то есть наклон крутой
for(i=0;i<dely;i++){
    // В зависимости от параметра p рисуем пиксель
    // либо по Dy, либо еще со смещением по Dx
    if(p<0)
    {
        y1=y1+1;
        putpixel(x1,y1,RED);
        p=p+(2*delx);
    }
    else
    {
        x1=x1+1;
        y1=y1+1;
        putpixel(x1,y1,RED);
        p=p+(2*delx)-(2*dely);
    }
}
// Задержка графика
// График закроется нажатием любой клавиши
getch();
// Закрытие графика. Окончание программы
closegraph();
return 0;
}

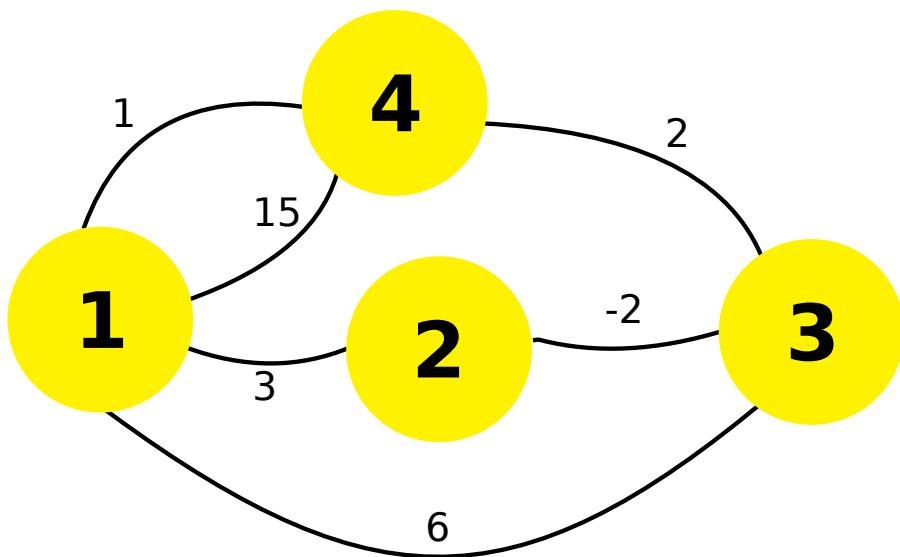
```

Глава 22: Алгоритм Флойда- Уоршелла

Раздел 22.1. Алгоритм кратчайшего пути для всех пар

Алгоритм Флойда-Уоршелла предназначен для поиска кратчайших путей во взвешенном графе с положительным или отрицательными весом ребер. Однократное выполнение алгоритма найдет длины (суммарные веса) кратчайших путей между всеми парами вершин. С небольшими вариациями он может распечатать кратчайший путь и обнаружить отрицательные циклы на графике. Флойд-Уоршелл - это алгоритм динамического программирования.

Давайте посмотрим на пример. Мы собираемся применить алгоритм Флойда-Уоршелла на этом графике:



Для начала возьмем 2 2D матрицы. Это матрицы смежности. Размер матриц будет общим числом вершин. Для нашего графа мы возьмем матрицы $4 * 4$. **Матрица расстояний** будет хранить минимальное расстояние, найденное между двумя вершинами в настоящее время. Впервые, для ребер, если есть ребро между $u-v$ и w является расстоянием, мы будем хранить: $\text{distance}[u][v] = w$. Для всех ребер, которые не существуют, мы ставим бесконечность. **Матрица путей** предназначена для восстановления пути минимального расстояния между двумя вершинами. Итак, изначально, если есть путь между u и v , мы будем кладь $\text{path}[u][v] = u$. Это означает, что лучший способ добраться до **вершины-v** из **вершины-u** - это использовать ребро, соединяющее v с u . Если между двумя вершинами пути нет, мы добавим туда N , указывая, что пути сейчас нет. Две таблицы для нашего графика будут выглядеть так:

	distance					path			
	1 2 3 4		1 2 3 4						
+-----+	-----+-----+-----+	+-----+	-----+-----+-----+	+-----+					
1 0 3 6 15	1 N 1 1 1								
+-----+	-----+-----+-----+	+-----+	-----+-----+-----+	+-----+					
2 inf 0 -2 inf	2 N N 2 N								
+-----+	-----+-----+-----+	+-----+	-----+-----+-----+	+-----+					
3 inf inf 0 2	3 N N N 3								
+-----+	-----+-----+-----+	+-----+	-----+-----+-----+	+-----+					
4 1 inf inf 0	4 4 N N N								
+-----+	-----+-----+-----+	+-----+	-----+-----+-----+	+-----+					
distance		path							

Поскольку петли нет, диагонали установлены на **N**. Расстояние от самой вершины равно **0**. Чтобы применить алгоритм Флойда-Уоршелла, мы выберем среднюю вершину **k**. Затем для каждой вершины **i** мы собираемся проверить, можем ли мы перейти от **i** к **k**, а затем от **k** к **j**, где **j** - другая вершина, и минимизировать стоимость перехода от **i** к **j**. Если текущее **distance[i][j]** больше, чем **distance[i][k] + distance[k][j]**, мы сделаем **distance[i][j]** равным сумме этих двух расстояний. И **path[i][j]** будет присвоено **path[k][j]**, как лучше перейти от **i** к **k**,

а затем от **k** к **j**. Все вершины будут выбраны как **k**. У нас будет 3 вложенных цикла: для **k** - от 1 до 4, для **i** - от 1 до 4, а для **j** - от 1 до 4. Мы собираемся проверить:

```

if distance[i][j] > distance[i][k] + distance[k][j]
    distance[i][j] := distance[i][k] + distance[k][j]
    path[i][j] := path[k][j]
end if

```

Ну что ж, в основном мы проверяем, *получаем ли для каждой пары вершин более короткое расстояние, проходя через другую вершину?* Общее количество операций для нашего графа будет $4 * 4 * 4 = 64$. Это означает, что мы собираемся сделать эту проверку 64 раза. Когда **k = 1, i = 2** и **j = 3**, **distance[i][j]** равно **-2**, которое не больше чем **distance[i][k] + distance[k][j] = -2 + 0 = -2**. Так что это останется без изменений. Опять же когда **k = 1, i = 4** и **j = 2**, **distance[i][j] = infinity**, которое больше чем **distance[i][k] + distance[k][j] = 1 + 3 = 4**. Итак, мы ставим **distance[i][j] = 4**, и мы положили **path[i][j] = path[k][j] = 1**. Это означает, что для перехода от **вершины-4** к **вершине-2** путь **4-> 1-> 2** короче существующего пути. Так мы заполняем обе матрицы. Расчет для каждого шага показан здесь. После внесения необходимых изменений наши матрицы будут выглядеть так:

	distance					path			
	1	2	3	4		1	2	3	4
+-----+-----+-----+-----+	+-----+-----+-----+-----+								
1 0 3 1 3	1 N 1 2 3								
+-----+-----+-----+-----+	+-----+-----+-----+-----+								
2 1 0 -2 0	2 4 N 2 3								
+-----+-----+-----+-----+	+-----+-----+-----+-----+								
3 3 6 0 2	3 4 1 N 3								
+-----+-----+-----+-----+	+-----+-----+-----+-----+								
4 1 4 2 0	4 4 1 2 N								
+-----+-----+-----+-----+	+-----+-----+-----+-----+								

Это наша матрица кратчайшего расстояния. Например, самое короткое расстояние от **1** до **4** равно **3**, а самое короткое расстояние от **4** до **3** - **2**. Наш псевдокод будет:

```
Procedure Floyd-Warshall(Graph):
for k from 1 to V // V denotes the number of vertex
    for i from 1 to V
        for j from 1 to V
            if distance[i][j] > distance[i][k] + distance[k][j]
                distance[i][j] := distance[i][k] + distance[k][j]
                path[i][j] := path[k][j]
            end if
            end for
        end for
    end for
```

Печать пути:

Чтобы напечатать путь, мы проверим матрицу **пути**. Чтобы напечатать путь от **u** до **v**, мы **path[u][v]**. Мы будем продолжать изменять **v = path[u][v]**, пока не найдем **path[u][v] = u** и не поместим все значения **path[u][v]** в стек. Найдя **u**, мы распечатаем **u** и начнем вытаскивать элементы из стопки и распечатывать их. Это работает, потому что матрица путей хранит значение вершины, которая разделяет кратчайший путь к **v** от любого другого узла. Псевдокод будет:

```
Procedure PrintPath(source, destination):
s = Stack()
S.push(destination)
while Path[source][destination] is not equal to source
S.push(Path[source][destination])
destination := Path[source][destination]
end while
print -> source
while S is not empty
print -> S.pop
end while
```

Нахождение отрицательного цикла в графе:

Чтобы выяснить, существует ли отрицательный цикл в графе, нам нужно проверить основную диагональ матрицы **расстояний**. Если какое-либо значение на диагонали отрицательно, это означает, что в графе есть отрицательный цикл.

Сложность:

Сложность алгоритма Флойда-Уоршелла составляет $O(V^3)$, а сложность пространства: $O(V^2)$.

Глава 23: Числа Каталана

Раздел 23.1: Числа Каталана. Основная информация

Алгоритм каталонских чисел - это алгоритм динамического программирования.

В комбинаторной математике каталонские числа образуют последовательность натуральных чисел, которые встречаются в различных задачах счета, часто связанных с участием рекурсивно определенных объектов. Каталонские числа на неотрицательных целых числах n представляют собой набор чисел, возникающих в задачах перечисления деревьев типа: «Сколькоими способами можно разделить регулярный n -угольник на $n-2$ треугольника, если разные ориентации учитываются отдельно?»

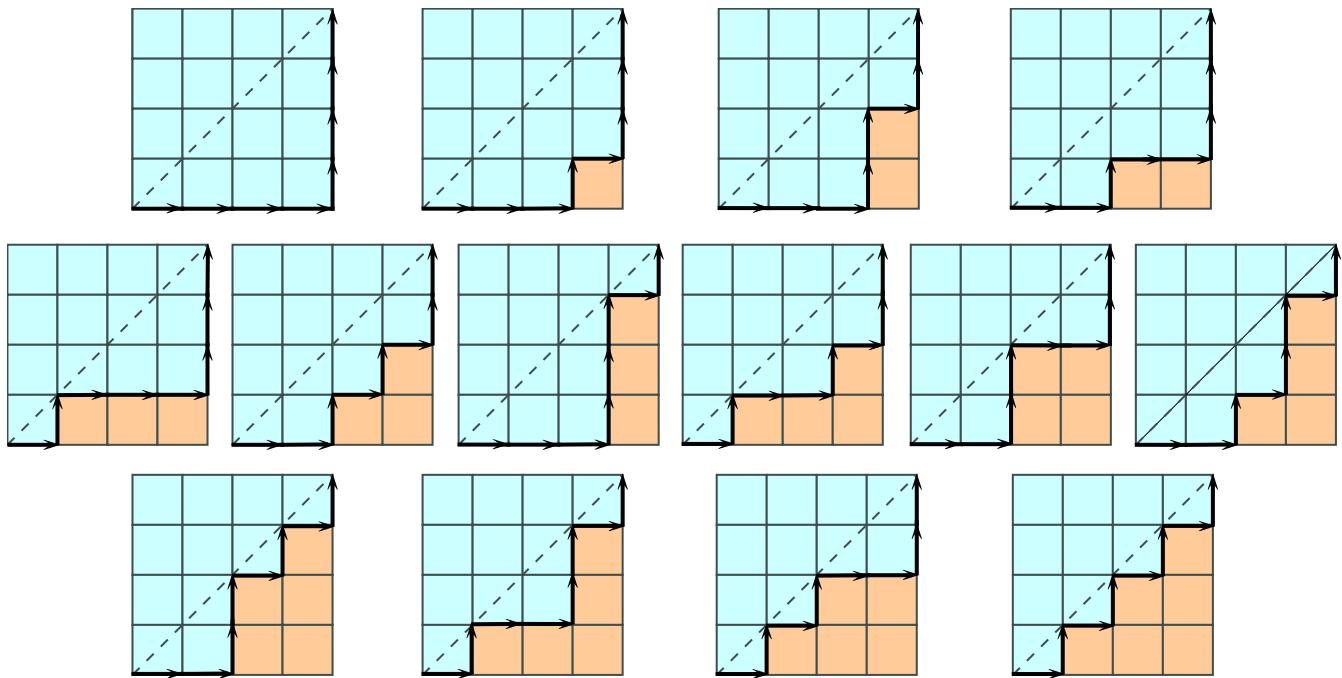
Применение алгоритма каталонского числа:

- Количество способов сложить монеты в нижнем ряду, который состоит из n последовательно лежащих монет в плоскости, так что монеты не могут быть размещены на двух сторонах нижних монет, и каждая дополнительная монета должна быть выше двух других монет, составляет n -ый каталонский номер.
- Число способов сгруппировать строку из n пар круглых скобок, чтобы каждая открытая скобка имела совпадающую закрытую скобку, является n -м каталонским числом.
- Число способов разрезать $n + 2$ -сторонний выпуклый многоугольник на плоскости на треугольники, соединяя вершины прямыми не пересекающимися линиями, является n -м каталонским числом. Это применения, которым интересовался Эйлер.

Используя нумерацию с нуля, n -е каталонское число дается непосредственно в терминах биномиальных коэффициентов следующим уравнением.

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} = \prod_{k=2}^n \frac{n+k}{k} \quad n \geq 0$$

Пример каталонского номера: Здесь значение $n = 4$. (Лучший пример - из Википедии)



Вспомогательное пространство: $O(n)$

Сложность по времени: $O(n^2)$

Глава 24:Многопоточные алгоритмы

Примеры некоторых многопоточных алгоритмов.

Раздел 24.1:Многопоточное умножение квадратной матрицы

```
multiply-square-matrix-parallel(A, B)
#функция умножения двух квадратных матриц
#переменная хранящая количество линий в матрице A
n = A.lines
#создаем матрицу размера n*n
C = Matrix(n,n)
#создаем n*n параллельных циклов, каждый из которых выполняет по одной итерации
parallel for i = 1 to n
    parallel for j = 1 to n
        #обнуляем элемент матрицы
        C[i][j] = 0
        #проходим k от 1 до n
        pour k = 1 to n
            #прибавляя произведение k-го элемента i строк матрицы A и k-го элемента
            #j столбца матрицы B
            C[i][j] = C[i][j] + A[i][k]*B[k][j]
#возвращаем матрицу C
return C
```

Раздел 24.2:Многопоточное умножение вектора матрицы

```
matrix-vector(A,x)
#создаем переменную количества строк матрицы
n = A.lines
y = Vector(n) #создаем вектор длины n
#запускаем параллельно n итераций цикла
parallel for i = 1 to n
    #обнуляем все элементы вектора
    y[i] = 0
#запускаем параллельно n итераций цикла
parallel for i = 1 to n
    #в каждом из них запускаем цикл от 1 до n проходящий по строке i
    for j = 1 to n
        #в каждой итерации прибавляем к элементу вектора y произведение
        #элемента i строки, j столбца и j элемента вектора x
        y[i] = y[i] + A[i][j]*x[j]
#возвращаем вектор y
return y
```

Раздел 24.3:Многопоточная сортировка слиянием

А – это массив, а р и q – индексы массива, например, если вы собираетесь отсортировать подмассив А[р..r], то В - это подмассив, который будет заполнен сортировкой.

Вызов p-merge-sort(A, p, r, B, s) сортирует элементы из A [p..r] и помещает их в B [s..s + r-p].

```
p-merge-sort(A,p,r,B,s)
#многопоточная сортировка подмассива
#находим длину подмассива
n = r-p+1
#если длина равна 1, то не сортируя помещаем элемент массива A в
#массив B На позицию s
if n==1
    B[s] = A[p]
else
    #создаем массив T длины n
    T = new Array(n)
    #находим середину отрезка массива, округляя до целого числа
    q = floor((p+r)/2)
    #находим длину половины отрезка
    q_prime = q-p+1
    #запускаем параллельно сортировку двух половин массива,
    #помещая результат в массив T
    spawn p-merge-sort(A,p,q,T,1)
    p-merge-sort(A,q+1,r,T,q_prime+1)
    sync
    #соединяем две отсортированные половины массива
    p-merge(T,1,q_prime,q_prime+1,n,B,s)
```

Вот вспомогательная функция, которая выполняет слияние параллельно. P-merge предполагает, что два подмассива для слияния находятся в одном массиве, но не предполагает, что они смежны в массиве. Вот почему нам нужны p1, r1, p2, r2.

```
p-merge(T,p1,r1,p2,r2,A,p3)
#длина первого отрезка
n1 = r1-p1+1
#длина второго отрезка
n2 = r2-p2+1

if n1<n2
    #меняем местами надала отрезков
    permute p1 and p2
    #меняем местами концы отрезков
    permute r1 and r2
    #меняем местами длины отрезка
    permute n1 and n2
if n1==0
    #массив пуст? тогда пусты оба массива
    return
else
    #округленная середина первого отрезка
    q1 = floor((p1+r1)/2)
    #ищем положение элемента с таким же значением во втором цикле
    q2 = dichotomic-search(T[q1],T,p2,r2)
    #положение элемента в конечном массиве
    q3 = p3 + (q1-p1) + (q2-p2)
    #помещаем элемент в результирующий массив
```

```

A[q3] = T[q1]
#параллельно вызываем функции слияния, для левых и
#правых половин отрезков массива
spawn p-merge(T,p1,q1-1,p2,q2-1,A,p3)
p-merge(T,q1+1,r1,q2,r2,A,q3+1)
sync

```

А вот вспомогательная функция дихотомического поиска.

x - это ключ для поиска в подмассиве $T[p..r]$.

```

dichotomic-search(x,T,p,r)
    #позиция наименьшего элемента массива
    inf = p
    #позиция наименьшего элемента массива
    sup = max(p,r+1)
    #пока позиция наименьшего элемента меньше позиции наибольшего
    while inf<sup
        #находим середину отрезка между наименьшим и наибольшим элементом
        half = floor((inf+sup)/2)
        #если ключ поиска меньше или равен значению середины отрезка
        if x<=T[half]
            #приравниваем супремум положению среднего элемента
            sup = half
        else
            #иначе, приравниваем инфимум положению среднего элемента
            inf = half+1
    return sup

```

Глава 25: Алгоритм Кнута-Морриса-Пратта (КМР)

КМР – это алгоритм сопоставления шаблонов, который ищет вхождения "слова" W в главной "текстовой строке" S , используя наблюдение, то есть при возникновении несоответствия у нас есть достаточно информации, чтобы определить, где может начаться следующее совпадение. Мы используем эту информацию, чтобы избежать совпадения с символами, которые, как мы знаем, в любом случае будут совпадать. В худшем случае сложность поиска шаблона уменьшается до $O(n)$.

Раздел 25.1: Пример КМР

Алгоритм

Этот алгоритм состоит из двух этапов. Сначала мы создаем вспомогательный массив $lps[]$, а затем используем его для поиска шаблона.

Предварительная обработка:

1. Мы предварительно обрабатываем шаблон и создаем вспомогательный массив $lps[]$, который используем для пропуска символов при соответствии.
2. Здесь $lps[]$ указывает на самый длинный собственный префикс, который также является суффиксом. Собственный префикс – это префикс, в который не включена вся строка. Например, префиксами строки **ABC** являются «», «**A**», «**AB**» и «**ABC**». Собственными префиксами являются «», «**A**» и «**AB**». Суффиксами строки являются «», «**C**», «**BC**» и «**ABC**».

Поиск

1. Мы сохраняем совпадающие символы $txt[i]$ и $pat[j]$ и продолжаем увеличивать i и j до тех пор, пока $pat[j]$ и $txt[i]$ совпадают.
2. Когда мы видим несоответствие, мы знаем, что символы $pat[0..j-1]$ совпадают с $txt[i-j+1..i-1]$. Мы также знаем, что $lps[j-1]$ – это количество символов из $pat[0..j-1]$, которые являются как собственным префиксом, так и суффиксом. Из этого мы можем сделать вывод, что нам не нужно сопоставлять символы $lps[j-1]$ с $txt[i-j..i-1]$, потому что мы знаем, что эти символы будут совпадать в любом случае.

Реализация на Java

```
public class KMP {  
  
    public static void main(String[] args) {  
        //строка данный над которой мы будем работать  
        String str = "abcababdabc";  
        //паттерн, являющийся префиксом и суффиксом  
        String pattern = "abc";  
        //создаем новый объект класс  
        KMP obj = new KMP();  
        //выводим строку данных и паттерн  
        System.out.println(obj.patternExistKMP(str.toCharArray(),  
    }  
  
    public int[] computeLPS(char[] str){  
        //создаем вспомогательный массив длина которого, равна длины строки  
        int lps[] = new int[str.length];  
        //обнуляем первый элемент массива и счетчик  
        lps[0] = 0;  
        int j = 0;  
        //обходим всю строку  
        for(int i =1;i<str.length;i++){  
            //если j элемент строки равен i элементе  
            if(str[j] == str[i]){  
                //кладем значение j+1 в i элемент вспомогательного массива  
                lps[i] = j+1;  
                //увеличиваем j и i  
                j++;  
                i++;  
            }else{  
                //иначе, если j не равен нулю, то j равен  
                //j-1 элементу вспомогательного массива  
                if(j!=0){  
                    j = lps[j-1];  
                }else{  
                    //если j равен нулю, то приравниваем i  
                    //элемент вспомогательного массива значению j+1  
                    lps[i] = j+1;  
                    //увеличиваем i  
                    i++;  
                }  
            }  
        }  
        //возвращаем вспомогательный массив  
        return lps;  
    }  
  
    public boolean patternExistKMP(char[] text,char[] pat){  
        //заполняем вспомогательный массив  
        int[] lps = computeLPS(pat);  
    }  
}
```

```

//обнуляем переменные
int i=0,j=0;
//пока длина строки больше i и длина паттерна больше j
while(i<text.length && j<pat.length){
    //если i буква строки равна J букве паттерна
    if(text[i] == pat[j]){
        //увеличиваем i и j на единицу
        i++;
        j++;
    }else{
        //иначе, если j не равен нулю, то присваиваем ему значение
        //элемента j-1 из вспомогательного массива в
        if(j!=0){
            j = lps[j-1];
        }else{
            //иначе увеличиваем i на единицу
            i++;
        }
    }
}
//если j равен длине паттерна возвращаем истину, иначе ложь
if(j==pat.length)
    return true;
return false;
}

```

Глава 26: Алгоритм изменения динамического расстояния

Раздел 26.1: Преобразования строки 1 в строку 2

Постановка задачи выглядит следующим образом: если нам даны две строки str1 и str2, то такое наименьшее количество операций может быть выполнено над str1, чтобы преобразовать ее в str2. Операции могут быть:

1. Вставить
2. Удалить
3. Заменить

Для примера

Input: str1 = "geek" , str2 = "gesek"

Output: 1

We only need to insert s in first string

Input: str1 = "march" , str2 = "cart"

Output: 3

We need to replace m with c and remove character c and then replace h with t

Чтобы решить эту проблему, мы будем использовать двумерный массив $dp[n+1][m+1]$, где n – длина первой строки, а m – длина второй строки. Для нашего примера, если str1 – это **azcef**, а str2 – это **abcdef**, тогда наш массив будет $dp[6][7]$, а окончательный ответ – $dp[5][6]$.

	(a) (b) (c) (d) (e) (f)
+---+---+---+---+---+---+---+	
	0 1 2 3 4 5 6
+---+---+---+---+---+---+---+	
(a)	1
+---+---+---+---+---+---+---+	
(z)	2
+---+---+---+---+---+---+---+	
(c)	3
+---+---+---+---+---+---+---+	
(e)	4
+---+---+---+---+---+---+---+	
(f)	5
+---+---+---+---+---+---+---+	

Для $dp[1][1]$ мы должны проверить, что мы можем сделать, чтобы преобразовать a в a . Это будет **0**. Для $dp[1][2]$ мы должны проверить, что мы можем сделать, чтобы преобразовать a в ab . Это будет **1**, потому что мы должны **вставить** b . Поэтому после 1-й итерации наш массив будет выглядеть так.

```
| | (a) | (b) | (c) | (d) | (e) | (f) |
+---+---+---+---+---+---+---+
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
|(a)| 1 | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
|(z)| 2 |   |   |   |   |   |
+---+---+---+---+---+---+---+
|(c)| 3 |   |   |   |   |   |
+---+---+---+---+---+---+---+
|(e)| 4 |   |   |   |   |   |
+---+---+---+---+---+---+---+
|(f)| 5 |   |   |   |   |   |
+---+---+---+---+---+---+---+
```

Для итерации 2

Для $dp[2][1]$ мы должны проверить, что для преобразования az в a нам нужно удалить z , поэтому $dp[2][1]$ будет **1**. Аналогично для $dp[2][2]$, где нам нужно заменить z с b , следовательно, $dp[2][2]$ будет **1**. Так что после 2-й итерации наш массив $dp[]$ будет выглядеть так.

```
| | (a) | (b) | (c) | (d) | (e) | (f) |
+---+---+---+---+---+---+---+
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
|(a)| 1 | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
|(z)| 2 | 1 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
|(c)| 3 |   |   |   |   |   |
+---+---+---+---+---+---+---+
|(e)| 4 |   |   |   |   |   |
+---+---+---+---+---+---+---+
|(f)| 5 |   |   |   |   |   |
+---+---+---+---+---+---+---+
```

Наша формула будет выглядеть так:

```
if characters are same
    //если символы равны
    //приравниваем элемент верхнему левому по диагонали
    dp[i][j] = dp[i-1][j-1];
else
    //иначе, приравниваем единице плюс минимальный из левого, верхнего и
    //верхнего левого по диагонали
    dp[i][j] = 1 + Min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])
```

После последней итерации наш массив $dp[]$ будет выглядеть так.

```

|   | (a) | (b) | (c) | (d) | (e) | (f) |
+---+---+---+---+---+---+---+
|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
+---+---+---+---+---+---+---+
|(a)| 1 | 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
|(z)| 2 | 1 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+---+
|(c)| 3 | 2 | 2 | 1 | 2 | 3 | 4 |
+---+---+---+---+---+---+---+
|(e)| 4 | 3 | 3 | 2 | 2 | 2 | 3 |
+---+---+---+---+---+---+---+
|(f)| 5 | 4 | 4 | 2 | 3 | 3 | 3 |
+---+---+---+---+---+---+---+

```

Реализация на Java

```

public int getMinConversions(String str1, String str2){
    //функция поиска количества элементарных изменений в строке 1
    //для приведения к строке 2
    //создаем массив из с количеством строк: длина первой строки плюс один,
    //и количеством столбцов: длина второй строки плюс один
    int dp[][] = new int[str1.length()+1][str2.length()+1];
    //цикл, количество итераций которого равно длине первой строки
    for(int i=0;i<=str1.length();i++){
        //вложенный цикл, количество итераций которого равно длине второй строки
        for(int j=0;j<=str2.length();j++){
            //если это первая итерация внешнего цикла
            if(i==0)
                //заполняем первую строку номерами столбцов
                dp[i][j] = j;
            //если это первый столбец
            else if(j==0)
                //заполняем первый столбец номерами строк
                dp[i][j] = i;
            //иначе если i-1 символ первой строки и j-1 символ второй
            //строки равны
            else if(str1.charAt(i-1) == str2.charAt(j-1))
                // приравниваем элемент верхнему левому по диагонали
                dp[i][j] = dp[i-1][j-1];
            //иначе
            else{
                //иначе, приравниваем единице плюс минимальный из левого,
                //верхнего и верхнего левого по диагонали
                dp[i][j] = 1 + Math.min(dp[i-1][j], Math.min(dp[i][j-1],
                    dp[i-1][j-1]));
            }
        }
    }
    //возвращаем нижний правый элемент массива
    return dp[str1.length()][str2.length()];
}

```

Глава 27: Онлайн алгоритмы

Теория:

Определение 1: Задача оптимизации Π состоит из набора экземпляров $\sum \Pi$. Для каждого экземпляра $\sigma \in \sum \Pi$ существует множество $Z\sigma$ решений и целевая функция $f\sigma : Z\sigma \rightarrow \mathcal{R} \geq 0$, которая присваивает каждому решению положительное вещественное значение. Мы говорим, что $OPT(\sigma)$ - это значение оптимального решения, $A(\sigma)$ - это решение алгоритма A для задачи Π и $wA(\sigma) = f\sigma(A(\sigma))$ - его значение.

Определение 2: Онлайн алгоритм A для задачи минимизации Π имеет конкурентное соотношение $r \geq 1$, если имеется постоянная $\tau \in \mathcal{R}$ такая, что

$$wA(\sigma) = f\sigma(A(\sigma)) \leq r \cdot OPT(\sigma) + \tau$$

для всех экземпляров $\sigma \in \sum \Pi$. A называется **r-конкурентным** онлайн алгоритмом. Если даже

$$wA(\sigma) \leq r \cdot OPT(\sigma)$$

для всех экземпляров $\sigma \in \sum \Pi$, тогда A называется **строго r-конкурентным** онлайн алгоритмом.

Теорема 1.3: LRU и FWF являются алгоритмами маркировки.

Доказательство: В начале каждой фазы (кроме первой) **FWF** имеет кэш-промах и очищает кэш. Это означает, что у нас есть k пустых страниц. В каждой фазе максимум k различных запрашиваемых страниц, так что на этой фазе будет исключение. Значит, **FWF** - это алгоритм маркировки.

Предположим, что **LRU** не является алгоритмом маркировки. Тогда есть экземпляр σ , где **LRU** помеченная страница x в фазе i исключена. Пусть σt сделает запрос в фазе i , где x исключена. Так как x отмечен, то должен быть более ранний запрос σt^* для x в той же фазе, так что $t^* < t$. После того, как t^*x стала самой новой страницей кэша, для того, чтобы она была исключена из t , последовательность $\sigma t^* + 1, \dots, \sigma t$ должна запрашивать по крайней мере k из x разных страниц. Это означает, что фаза i запросила по крайней мере $k+1$ разных страниц, что противоречит определению фазы. Поэтому **LRU** должен быть алгоритмом маркировки.

Теорема 1.4: Каждый алгоритм маркировки строго k-конкурентный.

Доказательство: Пусть σ - экземпляр для задачи пейджирования и l - число фаз для σ . Если $l = 1$, то каждый алгоритм маркировки оптимален, и оптимальный автономный алгоритм не может быть лучше.

Мы предполагаем, что $l \geq 2$. Стоимость каждого алгоритма разметки, например σ , ограничена сверху $l \cdot k$, потому что на каждой фазе алгоритм разметки не может исключать более k страниц без удаления одной помеченной страницы.

Теперь мы пытаемся показать, что оптимальный автономный алгоритм исключает, по крайней

мере, $k+l-2$ страницы для σ , k на первой фазе и, по крайней мере, по одной странице на каждой последующей фазе, кроме последней. Для доказательства определим $l-2$ дизъюнктивные подпоследовательности σ .

Последовательность $i \in 1, \dots, l-2$ начинается со второй позиции фазы $i+1$ и завершается первой позицией фазы $i+2$.

Пусть x будет первой страницей фазы $i+1$. В начале подпоследовательности i находится страница x и не более чем $k-1$ различных страниц в кэше оптимальных автономных алгоритмов. В подпоследовательности i k -й запрос страницы отличается от x , поэтому оптимальный автономный алгоритм должен исключить хотя бы одну страницу для каждой подпоследовательности. Так как на начальной фазе 1 кэш все еще пустой, оптимальный автономный алгоритм вызывает k исключений на первой фазе. Это показывает, что

$$wA(\sigma) \leq l \cdot k \leq (k+l-2)k \leq OPT(\sigma) \cdot k$$

Следствие 1.5: LRU и FWF являются строго k -конкурентными.

Если нет константы r , для которой онлайн алгоритм A является r -конкурентным, мы называем A неконкурентным.

Теорема 1.6: LFU и LIFO не являются конкурентными.

Доказательство: Пусть $l \geq 2$ - постоянная, $k \geq 2$ - размер кэша. Различные страницы кэша имеют нумерацию $1, \dots, k+1$. Мы смотрим на следующую последовательность:

$$\sigma = (1^l, 2^l, \dots, (k-1)^l, (k, k+1)^{l-1})$$

Первая страница 1 запрашивается l раз, потом страница 2 и так далее. В конце концов у нас есть $(l-1)$ чередующихся запросов для страницы k и $k+1$.

LFU и **LIFO** заполняют свой кэш страницами $1-k$. Когда запрашивается страница $k+1$ страница k исключается и наоборот. Это означает, что каждый запрос подпоследовательности $(k, k+1)^{l-1}$ вызывает одну страницу. Кроме того, при первом использовании страниц $1-(k-1)$ происходит кэширование $k-1$. Таким образом, **LFU** и **LIFO** исключают $k-1+2(l-1)$ страниц.

Теперь нужно показать, что для каждой константы $\tau \in \mathcal{R}$ и каждой константы $r \leq 1$ существует l такое, что

$$w_{LFU}(\sigma) = w_{LIFO}(\sigma) > r \cdot OPT(\sigma) + \tau$$

что эквивалентно

$$k-1+2(l-1) > r(k+1) + \tau \Leftrightarrow l \geq 1 + \frac{r \cdot (k+1) + \tau - k + 1}{2}$$

Чтобы это неравенство выполнилось, придется выбрать достаточно большое l . Таким образом, **LFU** и **LIFO** не являются конкурентными.

Теорема 1.7: Не существует r -конкурентного детерминированного онлайн алгоритма для пейджирования с $r < k$.

Используемые источники

Основной материал

1. Онлайн алгоритмы скриптов (на немецком языке), Хайко Реглин, Боннский университет.
2. Алгоритм замены страницы

Для дальнейшего чтения

1. Онлайн-вычисления и конкурентный анализ Аллана Бородина и Ран Эль-Янива

Исходный код

1. Исходный код автономного копирования
2. Исходный код игры оппонента

Раздел 27.1: Пейджирование (онлайн кэширование)

Предисловие

Вместо того, чтобы начинать с формальных определений, подход к теме будет сопровождаться рядом примеров, а определения будут вводиться по пути. Раздел примечаний **Теория** будет состоять из всех определений, теорем и следствий, чтобы дать вам всю информацию для более быстрого поиска конкретных аспектов.

Источники в разделе примечаний состоят из базового материала, используемого для написания главы и дополнительного материала для дальнейшего чтения. В добавок вы здесь также найдете примеры исходных кодов. Пожалуйста, обратите внимание, что для того, чтобы исходный код приведенных примеров был менее громоздкий, он воздерживается от таких вещей как обработка ошибок и т.д. Он также опускает некоторые специфические особенности языка, которые могут испортить ясность примера, такие как широкое использование расширенных библиотек и т.д.

Пейджирование

Проблема пейджирования возникает из-за ограничения конечного пространства. Предположим, что в нашем кэше C есть k страниц. Теперь мы хотим обработать последовательность из m страниц запросов, которые должны были быть помещены в кэш перед их обработкой. Конечно, если $m \leq k$, то мы просто помещаем все элементы в кэш, и это будет работать, но обычно мы имеем дело с $m >> k$.

Мы говорим, что запрос является **кэш-попаданием**, когда страница уже находится в кэше, в противном случае, его называют **кэш-промахом**. В этом случае мы должны ввести запрашиваемую страницу в кэш и исключить другую, предполагая, что кэш заполнен. Результатом будет граф исключений, который **минимизирует количество исключений**.

Существует множество стратегий для решения этой проблемы, давайте посмотрим на некоторые:

1. **Первым вошел, первым вышел (FIFO)**: Самая первая страница исключается
2. **Последним вошел, первым вышел (LIFO)**: Самая последняя страница исключается

3. **Наименее раннее пользование (LRU)**: Исключение страницы, чей самый последний доступ был самым ранним
4. **Наименее запрашиваемый (LFU)**: Исключение наименее запрашиваемой страницы (страницы, которая запрашивалась реже остальных)
5. **Наибольшее расстояние вперед (LFD)**: Исключить страницу в кэше, которая не запрашивается до дальнейшего будущего
6. **Очистка при заполнении (FWF)**: Очистка кэша происходит как только произошел кэш-промах

Есть два способа решить эту проблему:

1. **автономный**: последовательность запросов страниц известна заранее.
2. **онлайн**: последовательность запросов страниц неизвестна заранее

Автономный подход

Для первого случая рассмотрим применение жадной стратегии. Этот третий пример **автономного кэширования** рассматривает все вышеприведенные стратегии и дает хорошую отправную точку для дальнейшего понимания.

Пример программы был дополнен стратегией FWF:

```
class FWF : public Strategy {
public:
    FWF() : Strategy("FWF") //вызов конструктора класса Strategy
    {
    }

    /*возвращение индекса кэша, с которого можно вставлять элементы*/
    int apply(int requestIndex) override
    {
        for(int i=0; i<cacheSize; ++i)
        {
            if(cache[i] == request[requestIndex])
                return i;
        }
        // после первой пустой страницы, все остальные тоже должны быть пусты
        else if(cache[i] == emptyPage)
            return i;
    }
    // нет пустых страниц
    return 0;
}

void update(int cachePos, int requestIndex, bool cacheMiss) override {
    // нет пустых страниц -> пропустить -> очистить кэш
    if(cacheMiss && cachePos == 0)
    {
        for(int i = 1; i < cacheSize; ++i)
            cache[i] = emptyPage;
    }
}
```

Полный исходный код доступен [здесь](#). Если мы повторно используем пример из заголовка, то мы получим следующий вывод:

Strategy: FWF

Cache initial: (a,b,c)

Request	cache 0	cache 1	cache 2	cache miss
a	a	b	c	
a	a	b	c	
d	d	X	X	x
e	d	e	X	
b	d	e	b	
b	d	e	b	
a	a	X	X	x
c	a	c	X	
f	a	c	f	
d	d	X	X	x
e	d	e	X	
a	d	e	a	
f	f	X	X	x
b	f	b	X	
e	f	b	e	
c	c	X	X	x

Total cache misses: 5

Несмотря на то, что **LFD** является оптимальным, **FWF** имеет меньше кэш-промахов. Но главная цель заключалась в том, чтобы свести к минимуму количество исключений, а для **FWF** пять промахов означают 15 исключений, что делает его самым плохим выбором для данного примера.

Онлайн подход

Теперь мы хотим подойти к онлайн пейджированию. Но сначала нам нужно понять, как это сделать. Очевидно, что онлайн алгоритм не может быть лучше оптимального автономного алгоритма. Но насколько он хуже? Для ответа на этот вопрос нужны формальные определения:

Определение 1.1: Задача оптимизации Π состоит из набора экземпляров $\sum \Pi$. Для каждого экземпляра $\sigma \in \sum \Pi$ имеется набор $Z\sigma$ решений и целевая функция $f\sigma : Z\sigma \rightarrow \mathcal{R} \geq 0$, которая присваивает каждому решению аппозитивное вещественное значение. Мы говорим, что $OPT(\sigma)$ - это значение оптимального решения, $A(\sigma)$ - это решение алгоритма A для задачи Π и $wA(\sigma) = f\sigma(A(\sigma))$ является его значением.

Определение 1.2: Онлайн алгоритм A для задачи минимизации Π имеет конкурентное соотношение $r \geq 1$, если имеется постоянная $\tau \in \mathcal{R}$ такая, что

$$wA(\sigma) = f\sigma(A(\sigma)) \leq r \cdot OPT(\sigma) + \tau$$

для всех экземпляров $\sigma \in \sum \Pi$. A называется **r-конкурентным** онлайн алгоритмом. Если даже

$$wA(\sigma) \leq r \cdot OPT(\sigma)$$

для всех экземпляров $\sigma \in \sum \Pi$, тогда A называется **строго r-конкурентным** онлайн алгоритмом.

Таким образом, вопрос в том, насколько **конкурентным** является наш онлайн-алгоритм по сравнению с оптимальным автономным алгоритмом. В знаменитой [книге](#) Аллана Бородина и Ран Эль-Янива использовался другой сценарий для описания ситуации с онлайн пейджированием:

Есть **злостный враг**, который знает ваш алгоритм и оптимальный автономный алгоритм. На каждом шаге он пытается запросить страницу, которая является самой худшей для вас и одновременно лучшей для автономного алгоритма. **Конкурентный фактор** вашего алгоритма - это фактор того, насколько плохо ваш алгоритм справился с оптимальным автономным алгоритмом противника. Если вы хотите попробовать быть противником, вы можете попробовать [Игру в противника](#) (попробуйте побить стратегии подкачки страниц).

Алгоритмы маркировки

Вместо того, чтобы анализировать каждый алгоритм по отдельности, давайте посмотрим на специальное семейство онлайн алгоритмов для проблемы пейджирования, называемое **алгоритмами маркировки**.

Пусть $\sigma = (\sigma_1, \dots, \sigma_p)$ - экземпляр для нашей проблемы и k - размер нашего кэша, тогда σ можно разделить на фазы:

- Фаза 1 - это максимальная подпоследовательность σ от начала до максимального k различных страниц
- Фаза $i \geq 2$ - это максимальная подпоследовательность σ с конца фазы $i-1$ до максимального k различных запрошенных страниц.

Например, для $k = 3$:

$$\sigma = \left(\overbrace{a, b, d}^{\text{фаза 1}}, \overbrace{a, e, a, f}^{\text{фаза 2}}, \overbrace{a, f, b, d}^{\text{фаза 3}}, \overbrace{a, c, c, d}^{\text{фаза 4}} \right)$$

Алгоритм маркировки (прямо или косвенно) устанавливает, маркирована страница или нет. В начале каждой фазы все страницы не помечены. Если запрашивается страница во время фазы, она помечается. Алгоритмы маркировки никогда не исключают помеченную страницу из кэша. Это означает, что страницы, которые используются во время фазы, не будут исключены.

Теорема 1.3: LRU и FWF являются алгоритмом маркировки.

Доказательство: В начале каждой фазы (за исключением первой) FWF кэш-промах и очищает кэш. Это означает, что у нас есть k пустых страниц. В каждой фазе максимум k различных

запрашиваемых страниц, так что теперь будет исключение во время фазы. Значит, **FWF** - это алгоритм маркировки.

Предположим, что **LRU** не является алгоритмом маркировки. Тогда есть экземпляр σ , где **LRU** помеченная страница x в фазе i исключена. Пусть σt сделает запрос в фазе i , где X исключается. Так как x отмечен, то должен быть более ранний запрос σt^* для x в той же фазе, так что $t^* < t$. После t^* x - самая новая страница в кэше, так что для того, чтобы исключить t , последовательность $\sigma t^* + 1, \dots, \sigma t$ должна запрашивать по крайней мере k из x разных страниц. Это означает, что фаза i запросила по крайней мере $k+1$ разных страниц, что противоречит определению фазы. Поэтому **LRU** должен быть алгоритмом маркировки.

Утверждение 1.4: Каждый алгоритм маркировки является строго **k-конкурентным**.

Доказательство: Пусть σ – это пример для задачи пейджинга, а l – число фаз для σ . Если $l = 1$, то каждый алгоритм маркировки является оптимальным и оптимальный оффлайн алгоритм не может быть лучше.

Предположим, что $l \geq 2$. Стоимость каждого алгоритма маркировки, например σ , ограничена сверху числом $l \cdot k$, потому что каждая фаза алгоритма маркировки не может вытеснить больше k страниц, не вытеснив одну помеченную страницу.

Сейчас попробуем показать, что оптимальный оффлайн алгоритм вытесняет как минимум $k + l - 2$ страниц для σ , k в первой фазе и как минимум одну для каждой последующей фазы кроме последней. Для доказательства определим $l - 2$ разобщенные подпоследовательности для σ . Подпоследовательность $i \in \{1, \dots, l - 2\}$ начинается со второй позиции фазы $i + 1$ и заканчивается на первой позиции фазы $i + 2$. Пусть x – первая страница фазы $i + 1$. В начале подпоследовательности i находится страница x и не более $k - 1$ различных страниц в кэше оптимального оффлайн алгоритма. В подпоследовательности i есть запросы к страницам, отличных от x , поэтому оптимальный оффлайн алгоритм должен вытеснить как минимум одну страницу для каждой подпоследовательности. Поскольку в начале первой фазы кэш все еще пустой, оптимальный оффлайн алгоритм вытесняет k страниц во время первой фазы. Это говорит о том, что

$$wA(\sigma) \leq l \cdot k \leq (k + l - 2)k \leq OPT(\sigma) \cdot k$$

Утверждение 1.5: **LRU** и **FWF** являются строго **k-конкурентными**.

Упражнение: Показать, что **FIFO** не является алгоритмом маркировки, но является строго **k-конкурентным**.

Если не существует константы r , для которой онлайн алгоритм А является r -конкурентными, то будем называть А **неконкурентным**

Утверждение 1.6: **LFU** и **LIFO** являются **неконкурентными**.

Доказательство: Пусть $l \geq 2$ – константа, $k \geq 2$ – размер кэша. Различные страницы кэша пронумерованы от 1 до $k + 1$. Рассмотрим следующую последовательность:

$$\sigma = (1^l, 2^l, \dots, (k-1)^l, (k, k+1)^{l-1})$$

Первая страница запрашивается в l раз чаще, чем страница 2 и так далее. В конце есть $(l-1)$ чередующихся запросов на страницы k и $k+1$.

LFU и **LIFO** заполняют свой кэш страницами $1 - k$. Когда запрашивается страница $k + 1$, страница k вытесняется, и наоборот. Это означает, что каждый запрос подпоследовательности $(k, k+1)l - 1$ вытесняет одну страницу. Кроме того, происходит промах кэша $k - 1$ при первом использовании страниц $1 - (k - 1)$. Таким образом, **LFU** и **LIFO** вытесняют ровно $k - 1 + 2(l - 1)$ страниц.

Теперь мы должны показать, что для любой константы $\tau \in \mathfrak{R}$ и для любой константы $r \leq 1$ существует такое число l , что

$$\omega_{LFU}(\sigma) = \omega_{LIFO}(\sigma) > r \cdot OPT(\sigma) + \tau$$

что эквивалентно

$$k - 1 + 2(l - 1) > r(k + 1) + \tau \iff l \geq 1 + \frac{r \cdot (k + 1) + \tau - k + 1}{2}$$

Чтобы это неравенство выполнялось, нужно выбрать достаточно большое l . Так что **LFU** и **LIFO** неконкурентные

Утверждение 1.7: Не существует неконкурентного детерминированного алгоритма пейджинга, для которого $r < k$.

Доказательство последнего утверждения довольно длинное и основано на утверждении, что **LFD** является оптимальным оффлайн алгоритмом. Заинтересованный читатель может найти его в книге Бородина и Эль-Янива (см. источники ниже).

Вопрос в том, можно ли сделать это еще лучше. Для этого мы должны оставить детерминистский подход позади и начать randomизировать наш алгоритм. Очевидно, что другим будет сложнее понять ваш алгоритм, если он будет randomизированным.

Randomизированный пейджинг будет рассмотрен в одном из следующих примеров...

Глава 28: Сортировка

Характеристика	Описание
Устойчивость (стабильность)	Алгоритм сортировки называется устойчивым (стабильным) , если он сохраняет относительный порядок равных элементов после сортировки.
Сортировка на месте	Алгоритм сортировки называется сортировкой на месте , если он используется только $O(1)$ дополнительной памяти (не считая массива, который нужно отсортировать).
Минимальная сложность	Алгоритм сортировки имеет минимальную временную сложность $O(T(n))$, если время его работы не менее $T(n)$ при любых входных данных.
Средняя сложность	Алгоритм сортировки имеет среднюю временную сложность $O(T(n))$, если среднее время его работы при любых входных данных равно $T(n)$.
Максимальная сложность	Алгоритм сортировки имеет максимальную временную сложность $O(T(n))$, если время его работы не превышает $T(n)$.

Раздел 28.1: Устойчивость сортировок

Устойчивость сортировки означает, что алгоритм сортировки поддерживает относительный порядок равных ключей входных данных на выходе.

Таким образом, алгоритм сортировки считается устойчивым, если два объекта с одинаковыми ключами появляются в отсортированном массиве в том же порядке, что и во входном неотсортированном.

Рассмотрим список пар:

(1, 2) (9, 7) (3, 4) (8, 5) (9, 3)

Теперь отсортируем список по первому элементу пары.

Устойчивая сортировка этого списка выдаст следующее:

(1, 2) (3, 4) (8, 6) (9, 7) (9, 3)

потому что (9, 3) появляется после (9, 7) в исходном списке.

Неустойчивая сортировка выдаст следующий список:

(1, 2) (3, 4) (8, 6) (9, 3) (9, 7)

Нестабильная сортировка может генерировать те же выходные данные, что и стабильная, но не всегда.

Известные устойчивые сортировки:

- Сортировка слиянием
- Сортировка вставками
- Поразрядная сортировка
- Tim sort
- Сортировка пузырьком

Известные неустойчивые сортировки:

- Пирамидальная сортировка (сортировка кучей)
- Быстрая сортировка

Глава 29: Сортировка пузырьком (Bubble Sort)

Характеристика	Описание
Устойчивая (стабильность)	Да
На месте	Да
Минимальная сложность	$O(n)$
Средняя сложность	$O(n^2)$
Максимальная сложность	$O(n^2)$
Пространственная сложность	$O(1)$

Раздел 29.1: Сортировка пузырьком (Bubble Sort)

Сортировка пузырьком сравнивает каждую последовательную пару элементов в неупорядоченном списке и меняет их местами, если они идут не по порядку.

Следующий пример иллюстрирует пузырковую сортировку списка $\{6, 5, 3, 1, 8, 7, 2, 4\}$ (пары, которые сравнивались на каждом шаге помечаются '**'):

```
{6, 5, 3, 1, 8, 7, 2, 4}
{**5, 6**, 3, 1, 8, 7, 2, 4} - 5 < 6 -> меняем местами
{5, **3, 6**, 1, 8, 7, 2, 4} - 3 < 6 -> меняем местами
{5, 3, **1, 6**, 8, 7, 2, 4} - 1 < 6 -> меняем местами
{5, 3, 1, **6, 8**, 7, 2, 4} - 8 > 6 -> не меняем местами
{5, 3, 1, 6, **7, 8**, 2, 4} - 7 < 8 -> меняем местами
{5, 3, 1, 6, 7, **2, 8**, 4} - 2 < 8 -> меняем местами
{5, 3, 1, 6, 7, 2, **4, 8**} - 4 < 8 -> меняем местами
```

После одной итерации имеем список $\{5, 3, 1, 6, 7, 2, 4, 8\}$. Обратите внимание, что наибольшее значение в массиве (в данном случае 8), всегда достигнет своей конечной позиции. Итак, чтобы убедиться, что список отсортирован, мы должны выполнить $n-1$ итераций для списка длины n .

Визуализация

Раздел 29.2: Реализация на С и С++

Пример реализации пузырьковой сортировки на C++:

```
void bubbleSort(vector<int>numbers) //на вход поступает массив с числами
{
    for(int i = numbers.size() - 1; i >= 0; i--) { /*повторять для всех
элементов массива, кроме крайнего*/
        for(int j = 1; j <= i; j++) { //начиная со второго элемента
            if(numbers[j-1] > numbers[j]) {/*если j-ый элемент больше
предыдущего*/
```

```
        swap(numbers[j-1],numbers(j)); /*поменять значения местами*/
    }
}
}
}
```

Реализация на С

```

void bubble_sort(long list[], long n) /*на вход поступает массив со значениями и
его длина*/
{
    long c, d, t; /*c - номер "проверки на перестановку", d - индекс чисел массива*/
    for (c = 0 ; c < (n - 1); c++)//
    {
        for (d = 0 ; d < n - c - 1; d++)/*начиная от самого первого элемента массива,
заканчивая самым крайним (меняется в зависимости от номера проверки)*/
        {
            if (list[d] > list[d+1])//если последующий элемент больше d-ого
            {
                /*Перестановка*/
                t = list[d]; //запоминаем значение d-ого элемента
                list[d] = list[d+1]; /*значение d-ого элемента присваиваем последующему*/
                list[d+1] = t; //последующему элементу присваиваем значение d-го
            }
        }
    }
}

```

Пузырьковая сортировка с указателем

```
void pointer_bubble_sort(long * list, long n) /*на вход поступает список типа
long и его длина*/
{
    long c, d, t;

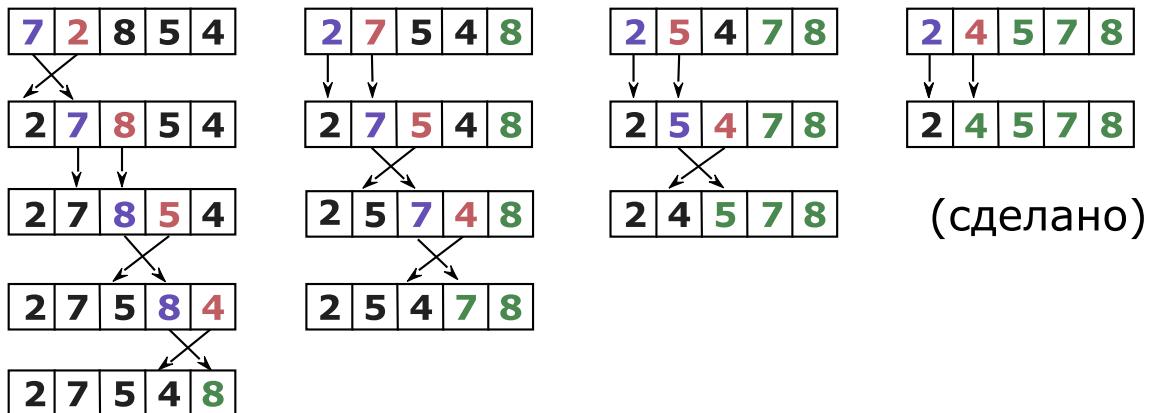
    for (c = 0 ; c < (n - 1); c++)
        for (d = 0 ; d < n - c - 1; d++)
    {
        if (* (list + d) > *(list+d+1)) //
        {
            /* Перестановка с помощью указателей*/

            t = * (list + d); //
            * (list + d) = * (list + d + 1);
            * (list + d + 1) = t;
        }
    }
}
```

Раздел 29.3: Реализация на C#

Пузырьковая сортировка также известна как **плавающая сортировка (Sinking Sort)**. Это простой алгоритм сортировки, который многократно проходит по сортируемому списку, сравнивает каждую пару соседних элементов и меняет их местами, если они находятся в неправильном порядке.

Реализация пузырьковой сортировки



Я использовал язык C# для реализации алгоритма сортировки пузырьком.

```
public class BubbleSort {
    public static void SortBubble(int[] input) //поступает массив
    {
        for (var i = input.Length - 1; i >= 0; i--) /*повторять на 1 раз меньше
размера массива*/
        {
            for (var j = input.Length - 1 - 1; j >= 0; j--) /*от первого элемента
до (размер массива-1)-го элемента*/
            {
                if (input[j] <= input[j + 1]) continue; /*если значение
следующего элемента больше*/
                var temp = input[j + 1]; /*запоминаем значение
следующего элемента*/
                //заменяем
                input[j + 1] = input[j];
                input[j] = temp;
            }
        }
    }

    public static int[] Main(int[] input)
    {
```

```

    //вызов функции
    SortBubble(input);
    return input;
}
}

```

Раздел 29.4: Реализация на Python

```

#!/usr/bin/python

input_list = [10,1,2,11] #создание массива

for i in range(len(input_list)):#повторять n раз, где n - кол-во элементов
    #в массиве
    for j in range(i):#для всех всех элементов
        if int(input_list[j]) > int(input_list[j+1]):#если значение последующего
            #элемента больше перестановка значений местами
            input_list[j],input_list[j+1] = input_list[j+1],input_list[j]

print input_list #вывод отсортированного массива

```

Раздел 29.5: Реализация на Java

```

public class MyBubbleSort {
    // функция сортировки пузырьком
    public static void bubble_srt(int array[]) {
        int n = array.length; // количество элементов
        int k;
        for (int m = n; m >= 0; m--) { // обратная итерация по массиву
            // прямая итерация, без последнего элемента
            for (int i = 0; i < n - 1; i++) {
                // перебор по индексу k + i
                k = i + 1;
                // если соседние элементы не по порядку
                if (array[i] > array[k]) {
                    // меняем местами
                    swapNumbers(i, k, array);
                }
            }
            // выводим
            printNumbers(array);
        }
    }
    // функция перестановки двух чисел в массиве
    private static void swapNumbers(int i, int j, int[] array) {
        int temp; // буфер для временного значения
        temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}

```

```

// функция вывода массива
private static void printNumbers(int[] input) {
    // итерация по всему массиву
    for (int i = 0; i < input.length; i++) {
        System.out.print(input[i] + ", ");
    }
    System.out.println("\n");
}

public static void main(String[] args) {
    // тестовые данные
    int[] input = { 4, 2, 9, 6, 23, 12, 34, 0, 1 };
    // заход в главную функцию
    bubble_srt(input);
}
}

```

Раздел 29.6: Реализация на Javascript

```

function bubbleSort(a) {
    var swapped; // флаг произошедшей сортировки
    do {
        swapped = false; // обнуление флага
        for (var i=0; i < a.length-1; i++) { // проход по всему массиву
            if (a[i] > a[i+1]) { // если соседние элементы не по порядку
                var temp = a[i]; // записываем элемент во временную переменную
                a[i] = a[i+1]; // меняем значения переменных местами
                a[i+1] = temp;
                swapped = true; // сортировка произошла
            }
        }
    } while (swapped); // пока все элементы не по порядку
}

var a = [3, 203, 34, 746, 200, 984, 198, 764, 9]; // тестовый набор
bubbleSort(a);
// результат
console.log(a); //logs [ 3, 9, 34, 198, 200, 203, 746, 764, 984 ]

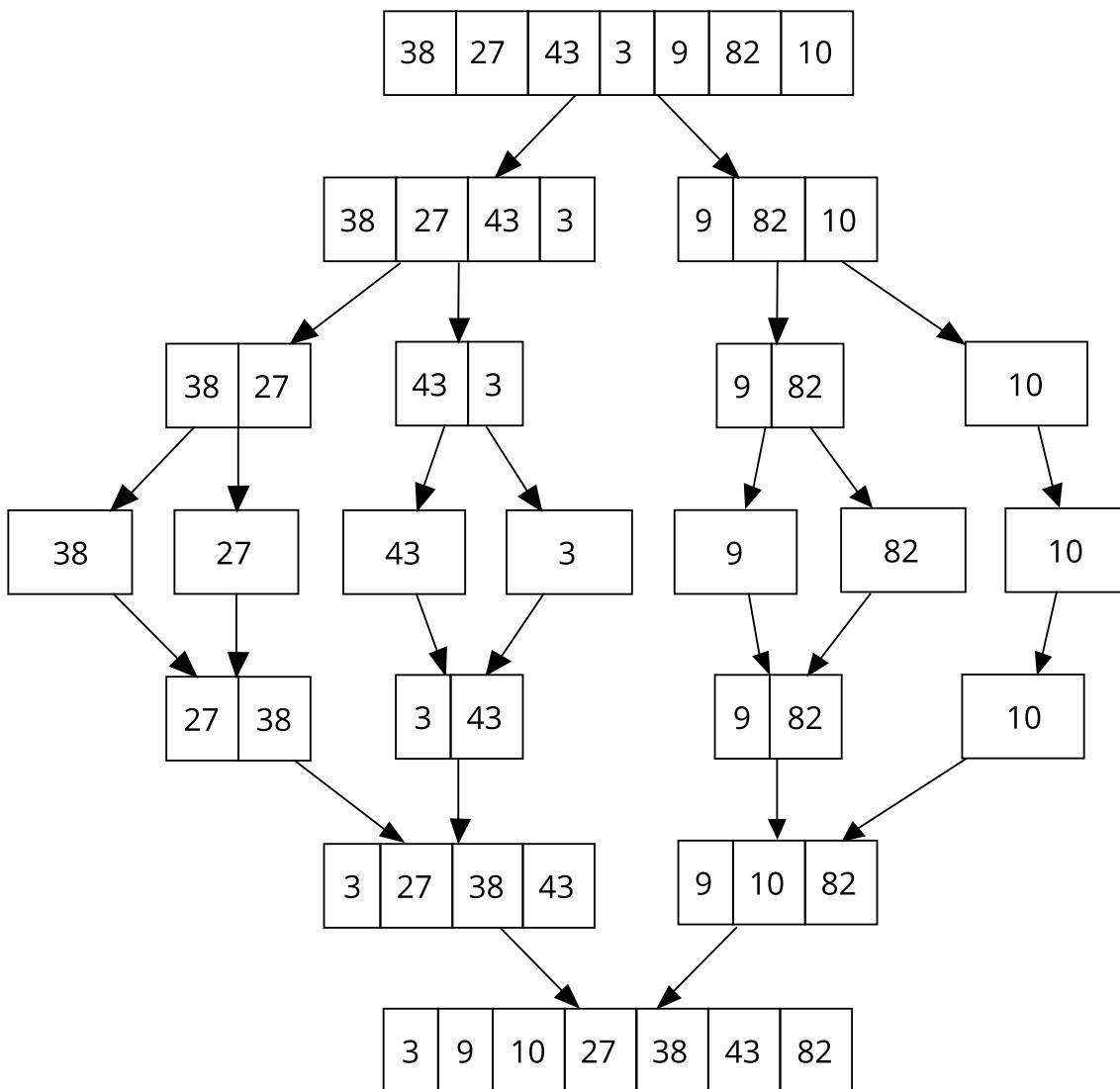
```

Глава 30: Сортировка слиянием

Раздел 30.1: Основы сортировки слиянием

Сортировка слиянием - это алгоритм типа «разделяй и властвуй». Он делит входной список длиной n пополам до тех пор, пока не будет n списков размером 1. Затем пары списков объединяются вместе с меньшим первым элементом из пары списков, добавленных на каждом шагу. Посредством последовательного слияния и сравнения первых элементов создается отсортированный список.

Пример:



Временная сложность: $T(n) = 2T(n/2)+\Theta(n)$

Вышеуказанная рекуррентность может быть решена с использованием метода дерева рекурсивных вызовов или метода рекуррентных соотношений. Это относится к случаю II метода рекуррентных соотношений, и решение получается за $\theta(n \log n)$. Временная сложность сортировки слиянием составляет $\theta(n \log n)$ во всех 3 случаях (наихудший, средний и лучший), поскольку сортировка слиянием всегда делит массив на две половины и требует линейного времени для объединения двух половин.

Вспомогательное пространство: $O(n)$

Алгоритмическая парадигма: Разделяй и властвуй

Сортировка на месте: Нетипичная реализация

Стабильно: Да

Раздел 30.2: Реализация сортировки слиянием на Go

```
package main
import "fmt"
func mergeSort(a []int) []int {
    // базовый случай - один элемент в массиве - конечный результат
    if len(a) < 2 {
        return a
    }
    // медианный индекс
    m := (len(a)) / 2
    // левая часть массива
    f := mergeSort(a[:m])
    // правая часть
    s := mergeSort(a[m:])
    // возвращаем через рекурсию более мелкие части массива до 1 элемента
    return merge(f, s)
}
func merge(f []int, s []int) []int {
    var i, j int
    // сумма сравниваемых частей массива
    size := len(f) + len(s)
    // выделяем память под результирующий массив
    a := make([]int, size, size)
    // итерация по всему массиву
    for z := 0; z < size; z++ {
        lenF := len(f) // длина первой части
        lenS := len(s) // длина второй части
        if i > lenF-1 && j <= lenS-1 { // элементов второго массива больше
            a[z] = s[j] // присваиваем недостающий элемент
            j++
        } else if j > lenS-1 && i <= lenF-1 { // первый массив больше
            a[z] = f[i] // присваиваем недостающий элемент
            i++
        } else if f[i] < s[j] { // элемент второго массива больше
            a[z] = f[i] // присваиваем элемент первого
            i++ // элемент первого массива больше
        } else {
            a[z] = s[j] // присваиваем элемент второго
            j++
        }
    }
    return a // возвращаем результирующий массив
}
func main() {
```

```

// тестовые
a := []int{75, 12, 34, 45, 0, 123, 32, 56, 32, 99, 123, 11, 86, 33}
// вывод исходных данных
fmt.Println(a)
// вывод результата
fmt.Println(mergeSort(a))
}

```

Раздел 30.3: Реализация сортировки слиянием на C & C#

Сортировка слиянием на C

```

int merge(int arr[],int l,int m,int h) {
    // Два временных массива для объединения
    int arr1[10],arr2[10];
    int n1,n2,i,j,k;
    n1=m-l+1; // длина левой части
    n2=h-m; // длина правой части
    for(i=0; i<n1; i++) // записываем в буфер левую часть
        arr1[i]=arr[l+i];
    for(j=0; j<n2; j++)
        arr2[j]=arr[m+j+1]; // записываем в буфер правую часть
    arr1[i]=9999; // отметить конец каждого временного массива
    arr2[j]=9999;
    i=0;pulseaudio -k && sudo alsa force-reload
    j=0;
    for(k=l; k<=h; k++) { // объединение двух отсортированных массивов
        if(arr1[i]<=arr2[j]) // меньший элемент присваиваем результату
            arr[k]=arr1[i++];
        else
            arr[k]=arr2[j++];
    }
    return 0;
}
int merge_sort(int arr[],int low,int high) {
    int mid;
    if(low<high) {
        mid=(low+high)/2; // медианный индекс
        // Разделение
        merge_sort(arr,low,mid); // на левую
        merge_sort(arr,mid+1,high); // на правую
        merge(arr,low,mid,high); // слияние
    }
    return 0;
}

```

Сортировка Слиянием на C#

```

public class MergeSort {
    static void Merge(int[] input, int l, int m, int r) {
        int i, j;

```

```

var n1 = m - l + 1; // длина левой части
var n2 = r - m; // длина правой части
var left = new int[n1]; // выделяем память под два массива
var right = new int[n2];
for (i = 0; i < n1; i++) {
    left[i] = input[l + i]; // заполняем левыми элементами
}
for (j = 0; j < n2; j++) {
    right[j] = input[m + j + 1]; // заполняем правыми элементами
}
i = 0;
j = 0;
var k = l;
while (i < n1 && j < n2) {
    if (left[i] <= right[j]) { // элемент левого массива меньше
        input[k] = left[i]; // заполняем им
        i++;
    }
    else {
        input[k] = right[j]; // иначе элемент правого массива
        j++;
    }
    k++;
}
// один из следующих циклов будет пропущен
while (i < n1) { // остается добавить либо левую часть
    input[k] = left[i];
    i++;
    k++;
}
while (j < n2) { // либо правую
    input[k] = right[j];
    j++;
    k++;
}
}
static void SortMerge(int[] input, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2; // медианный индекс
        SortMerge(input, l, m); // рекурсивно разбиваем правую часть
        SortMerge(input, m + 1, r); // рекурсивно разбиваем левую часть
        Merge(input, l, m, r); // сливаем части
    }
}
public static int[] Main(int[] input) {
    SortMerge(input, 0, input.Length - 1); // начало сортировки
    return input;
}
}

```

Раздел 30.4: Реализация сортировки слиянием на Java

Ниже приводится реализация на Java с использованием обобщенного программирования. Это тот же алгоритм, который представлен выше.

```
public interface InPlaceSort<T extends Comparable<T>> {
    void sort(final T[] elements); // шаблон элементов массива
}
public class MergeSort < T extends Comparable < T >> implements InPlaceSort < T > {

    @Override
    public void sort(T[] elements) {
        // объявляем сравниваемый массив
        T[] arr = (T[]) new Comparable[elements.length];
        // начинаем сортировку
        sort(elements, arr, 0, elements.length - 1);
    }
    // Мы проверяем обе наши стороны и затем объединяем их
    private void sort(T[] elements, T[] arr, int low, int high) {
        if (low >= high) return; // в случае пустого массива
        int mid = low + (high - low) / 2; // медианный элемент
        sort(elements, arr, low, mid); // рекурсия левой части
        sort(elements, arr, mid + 1, high); // рекурсия правой части
        merge(elements, arr, low, high, mid); // слияние
    }
    private void merge(T[] a, T[] b, int low, int high, int mid) {
        int i = low; // начало левого массива
        int j = mid + 1; // начало правого массива
        // выбираем самый меньший элемент из двух, помещаем в массив
        for (int k = low; k <= high; k++) {
            if (i <= mid && j <= high) {
                if (a[i].compareTo(a[j]) >= 0) { // вычитание правого из левого
                    b[k] = a[j++];
                } else { // иначе, левый элемент больше
                    b[k] = a[i++];
                }
            } else if (j > high && i <= mid) { // один из массивов закончился
                b[k] = a[i++];
            } else if (i > mid && j <= high) {
                b[k] = a[j++];
            }
        }
        for (int n = low; n <= high; n++) {
            a[n] = b[n]; // запись результата в первый массив
        }
    }
}
```

Раздел 30.5: Реализация сортировки слиянием на Python

```
def merge(X, Y):
    " слияние двух отсортированных массивов "
    p1 = p2 = 0
    out = [] # конечный массив
    while p1 < len(X) and p2 < len(Y):
        if X[p1] < Y[p2]:
            out.append(X[p1]) # записываем меньшие элементы из двух исходных
            p1 += 1 # инкремент индекса
        else:
            out.append(Y[p2])
            p2 += 1 # инкремент второго индекса
    out += X[p1:] + Y[p2:] # запись остатка
    return out
def mergeSort(A):
    if len(A) <= 1: # базовые случаи
        return A
    if len(A) == 2:
        return sorted(A)
    mid = len(A) / 2 # медианное значение
    return merge(mergeSort(A[:mid]), mergeSort(A[mid:])) # рекурсия
if __name__ == "__main__":
    # генерируем 20 элементов тестовых данных
    A = [randint(1, 100) for i in xrange(20)] print mergeSort(A)
```

Раздел 30.6: Реализация на Java метода восходящего слияния

```
public class MergeSortBU {
    // тестовые данные
    private static Integer[] array = {
        4, 3, 1, 8, 9, 15, 20, 2, 5, 6, 30, 70, 60, 80, 0, 9, 67, 54, 51, 52, 24, 54, 7      };
    public MergeSortBU() {
    }
    private static void merge(Comparable[] arrayToSort, Comparable[] aux, int lo,int mid,
        // делаем копию
        for (int index = 0; index < arrayToSort.length; index++) {
            aux[index] = arrayToSort[index];
        }
        int i = lo; // нижний индекс
        int j = mid + 1; // медиана
        for (int k = lo; k <= hi; k++) {
            if (i > mid) // когда дошли до середины идним из массивов
                arrayToSort[k] = aux[j++];
            else if (j > hi)
                arrayToSort[k] = aux[i++];
            // выполняется сначала нижняя часть составных условий
            else if (isLess(aux[i], aux[j])) {
                // присваиваем меньший элемент, увеличиваем счетчик
            }
        }
    }
}
```

```

        arrayToSort[k] = aux[i++];
    } else {
        arrayToSort[k] = aux[j++];
    }
}
}

public static void sort(Comparable[] arrayToSort, Comparable[] aux, int lo, int hi) {
    int N = arrayToSort.length; // количество элементов
    // цикл с шагом в два раза большим предыдущего
    for (int sz = 1; sz < N; sz = sz + sz) {
        // шаг больше на величину индекса, прошлого элемента
        for (int low = 0; low < N; low = low + sz + sz) {
            System.out.println("Size:" + sz); // вывод размера
            // слияние непересекающихся частей
            merge(arrayToSort, aux, low, low + sz - 1, Math.min(low + sz + sz - 1, N - 1));
            print(arrayToSort); // вывод результата
        }
    }
}

public static boolean isLess(Comparable a, Comparable b) {
    return a.compareTo(b) <= 0; // истина, если первый элемент меньше
}

private static void print(Comparable[] array) {
    StringBuffer buffer = new StringBuffer(); // объявляем для копии
    for (Comparable value : array) { // заполняем с пробелами
        buffer.append(value);
        buffer.append(' ');
    }
    System.out.println(buffer); // выводим копию
}

public static void main(String[] args) {
    // объявляем обрабатываемый массив
    Comparable[] aux = new Comparable[array.length];
    print(array);
    MergeSortBU.sort(array, aux, 0, array.length - 1);
}
}
}

```

Глава 31: Сортировка вставками

Раздел 31.1: Реализация на Haskell

```
-- делаем копию массива
insertSort :: Ord a => [a] -> [a]
insertSort [] = []
-- перебор по исходному с присваиванием
insertSort (x:xs) = insert x (insertSort xs)
-- начало условия включения
insert :: Ord a => a-> [a] -> [a]
-- включить остаток массива
insert n [] = [n]
-- который меньше чем текущий
insert n (x:xs) | n <= x = (n:x:xs)
                -- иначе обратная часть
                | otherwise = x:insert n xs
```

Глава 32: Блочная сортировка

Раздел 32.1: Реализация на C#

```
public class BucketSort {
    public static void SortBucket(ref int[] input) {
        int minValue = input[0]; // начальные пограничные значения
        int maxValue = input[0];
        int k = 0;
        // обратный цикл по массиву
        for (int i = input.Length - 1; i >= 1; i--) {
            // поиск максимального и минимального значения
            if (input[i] > maxValue)
                maxValue = input[i];
            if (input[i] < minValue)
                minValue = input[i];
        }
        // объявление двойного массива индексов значений
        List<int>[] bucket = new List<int>[maxValue - minValue + 1];
        for (int i = bucket.Length - 1; i >= 0; i--) {
            bucket[i] = new List<int>(); // объявление для индексов чисел
        }
        foreach (int i in input) {
            bucket[i - minValue].Add(i); // добавляем индексы ключа-значения
        }
        foreach (List<int> b in bucket) {
            if (b.Count > 0) {
                // если элемент присутствует в подмассиве индексов
                foreach (int t in b) {
                    input[k] = b[t]; // заполняем значениями в выходной массив
                    k++;
                }
            }
        }
    }
    public static int[] Main(int[] input) {
        SortBucket(ref input); // в функцию указатель на исходный массив
        return input; // значение изменилось на результат
    }
}
```

Глава 33: Быстрая сортировка

Раздел 33.1: Основы быстрой сортировки

Быстрая сортировка - это алгоритм сортировки, который выбирает элемент («опорный элемент») и переупорядочивает массив, образуя два разбиения, так что все элементы, меньшие опорного, идут перед ним, а все элементы с большим значением идут после него. Затем алгоритм рекурсивно применяется к разбиениям до тех пор, пока список не будет отсортирован.

1. Механизм схемы разбиения Ломуто:

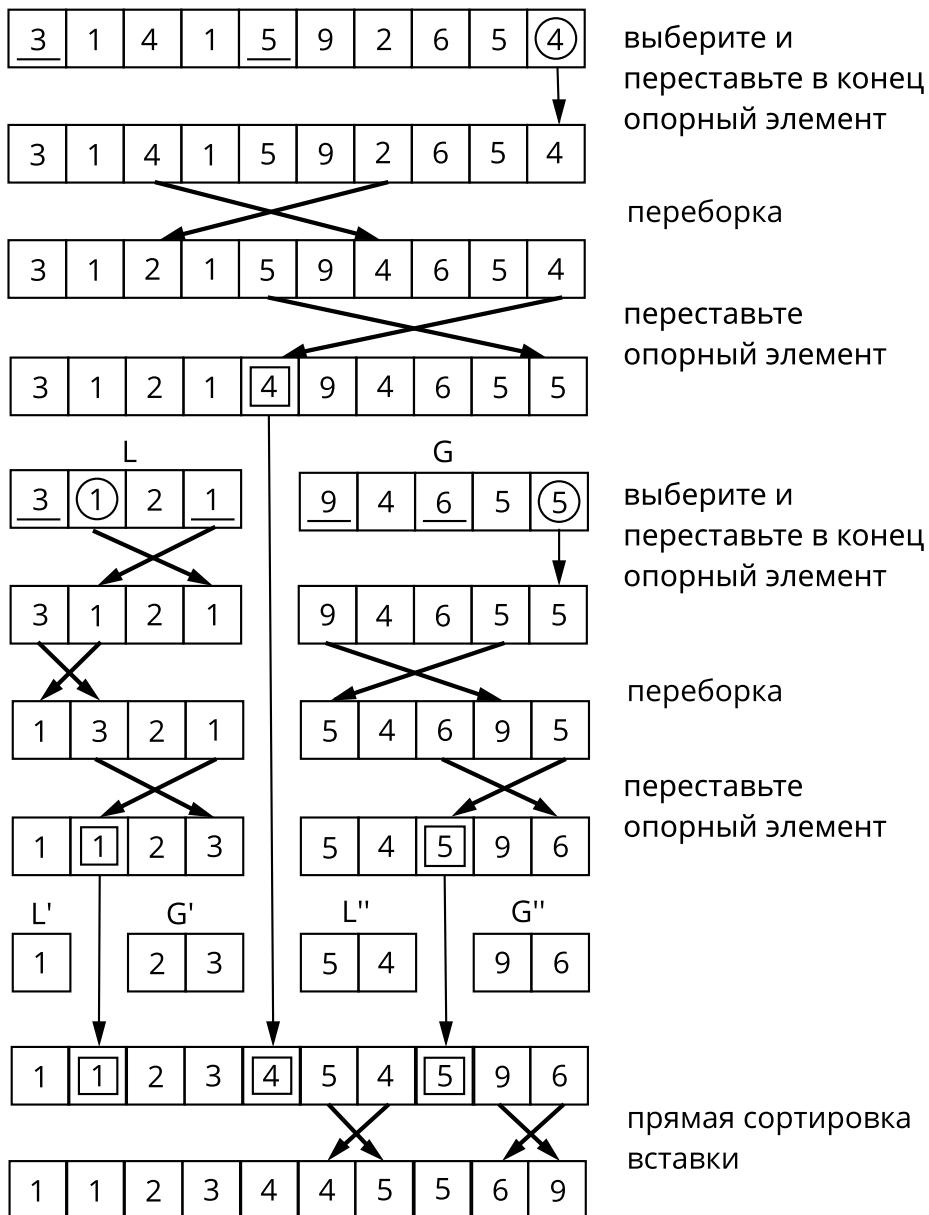
Эта схема выбирает опорный элемент, который обычно является последним элементом в массиве. Алгоритм сохраняет индекс, чтобы поместить опорный элемент в переменную i , и каждый раз, когда он находит элемент, меньший или равный опорному элементу, этот индекс увеличивается, и этот элемент будет помещен перед опорным.

```
partition(A, low, high) is
    pivot := A[high] // опорный элемент
    i := low // индекс начала
    for j := low to high - 1 do
        if A[j] <= pivot then // если текущий элемент меньше опорного
            swap A[i] with A[j] // меняем их местами
            i := i + 1 // смещаем индекс начала
    swap A[i] with A[high] // меняем последний элемент с обрабатываемым
    return i // возвращаем индекс обрабатываемого элемента
```

Механизм быстрой сортировки:

```
quicksort(A, low, high) is
    if low < high then // обработка в заданном диапазоне
        p := partition(A, low, high)
        quicksort(A, low, p - 1) // рекурсивный обход с левой частью
        quicksort(A, p + 1, high) // с правой частью
```

Пример быстрой сортировки:



2. Схема разбиения Хоара:

Она использует два индекса, которые начинаются на концах разбиваемого массива, а затем движутся навстречу друг другу, пока не обнаружат инверсию: пара элементов, один из которых больше или равен опорному, другой меньше или равен опорному, и эта пара неправильно упорядочена относительно друг друга. Затем инвертированные элементы меняются местами. Когда индексы встречаются, алгоритм останавливается и возвращает конечный индекс. Схема Хоара более эффективна, чем схема разбиения Ломуто, потому что она делает в среднем в три раза меньше перестановок и создает эффективные разбиения, даже если все значения равны.

```

quicksort(A, lo, hi) is
if lo < hi then // обработка в заданном диапазоне
    p := partition(A, lo, hi)
    quicksort(A, lo, p) // рекурсивный обход с левой частью
    quicksort(A, p + 1, hi) // с правой частью

```

Разбиение :

```

partition(A, lo, hi) is
pivot := A[lo] // опорный первый элемент
i := lo - 1
j := hi + 1
loop forever do:
    i := i + 1 // прямой перебор, сдвигаем левую границу вправо
    while A[i] < pivot do: // пока меньше опорного
        j := j - 1 // сдвигаем правую границу влево
    while A[j] > pivot do: // пока правый граничный больше опорного
        if i >= j then // если границы сомкнулись
            return j // возвращаем граничный индекс элемента
    swap A[i] with A[j] // меняем местами

```

33.2: Реализация быстрой сортировки на Python

```

def quicksort(arr):
    if len(arr) <= 1: # базовый случай
        return arr
    pivot = arr[len(arr) / 2] # опорный элемент посередине
    left = [x for x in arr if x < pivot] # все элементы, меньше опорного
    middle = [x for x in arr if x == pivot] # все равные опорному
    right = [x for x in arr if x > pivot] # все большие опорного
    # рекурсивно обрабатываются две части
    return quicksort(left) + middle + quicksort(right)

print quicksort([3,6,8,10,1,2,1])

```

Печатает "[1, 1, 2, 3, 6, 8, 10]"

Раздел 33.3: Реализация разбиения Ломуту на Java

```

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in); // вводим массив
    int n = sc.nextInt();
    int[] ar = new int[n]; // объявляем обрабатываемый массив
    for(int i=0; i<n; i++)
        ar[i] = sc.nextInt(); // заполняем
    quickSort(ar, 0, ar.length-1);
}

public static void quickSort(int[] ar, int low, int high) {
    if(low<high) { // если границы не сомкнулись
        int p = partition(ar, low, high); // разбиваем с опорным элементом
        quickSort(ar, 0 , p-1); // обработка левой части
        quickSort(ar, p+1, high); // правой части
    }
}

public static int partition(int[] ar, int l, int r) {
    int pivot = ar[r]; // опорный последний элемент
    int i = l;
    for(int j=l; j<r; j++) { // начиная слевой границы

```

```
if(ar[j] <= pivot) { // элемент меньше опорного
    int t = ar[j]; // меняем местами текущий с опорным
    ar[j] = ar[i];
    ar[i] = t;
    i++; // шаг для левой границы
}
}
int t = ar[i]; // смена местами
ar[i] = ar[r];
ar[r] = t;
return i;
}
}
```

Глава 34. Сортировка подсчетом

Раздел 34.1: Основная информация о сортировке подсчетом

Сортировка подсчетом — это алгоритм целочисленной сортировки для набора объектов, который сортируется в соответствии с ключами объектов.

Этапы

- Строим рабочий массив C , размер которого равен диапазону входного массива A .
- Перебираем массив A и присваиваем числу $C[x]$ количество вхождений числа x в массив A .
- Преобразовываем C в массив, где $C[x]$ - количество значений $\leq x$, и присваиваем каждому $C[x]$ сумму значений предыдущего и текущего элементов.
- Перебираем массив A с конца, помещаем каждое значение в новый отсортированный массив B по индексу, записанному в C . Делается это для текущего элемента $A[x]$ путем присваивания $A[x]$ в $B[C[A[x]]]$ и декрементирования $C[A[x]]$ в том случае, если было несколько чисел равных по значению в исходном неотсортированном массиве.

Пример сортировки подсчетом:

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3
	0	1	2	3	4	5		
C	2	0	2	3	0	1		

(a)

	0	1	2	3	4	5	6	7	8
C	2	2	4	7	7	8			

(b)

	1	2	3	4	5	6	7	8
B							3	
	0	1	2	3	4	5		

(c)

	1	2	3	4	5	6	7	8
B		0					3	
	0	1	2	3	4	5		
C	1	2	4	6	7	8		

(d)

	1	2	3	4	5	6	7	8
B		0					3	
	0	1	2	3	4	5		
C	1	2	4	5	7	8		

(e)

	1	2	3	4	5	6	7	8
B		0	0	2	2	3	3	3
	0	1	2	2	3	3	3	5

(f)

Дополнительная память: $O(n+k)$

Временная сложность: в худшем случае - $O(n+k)$, в лучшем - $O(n)$, в стандартном - $O(n+k)$

Раздел 34.2: Реализация на псевдокоде

Ограничения:

1. Входные данные (массив для сортировки)
2. Количество элементов на входе (n)
3. Ключи в диапазоне $0..k-1$ (k)
4. Счетчик (массив чисел)

Псевдокод:

```
for x in input:  
    count[key(x)] += 1  
total = 0  
for i in range(k):  
    oldCount = count[i]  
    count[i] = total  
    total += oldCount  
for x in input:  
    output[count[key(x)]] = x  
    count[key(x)] += 1  
return output
```

Глава 35: Пирамидальная сортировка

Раздел 35.1: Реализация на C#

```
public class HeapSort
{
    public static void Heapify(int[] input, int n, int i)
    {
        int largest = i;
        int l = i + 1;
        int r = i + 2;

        if (l < n && input[l] > input[largest])
            largest = l;

        if (r < n && input[r] > input[largest])
            largest = r;

        if (largest != i)
        {
            var temp = input[i];
            input[i] = input[largest];
            input[largest] = temp;
            Heapify(input, n, largest);
        }
    }

    public static void SortHeap(int[] input, int n)
    {
        for (var i = n - 1; i >= 0; i--)
        {
            Heapify(input, n, i);
        }
        for (int j = n - 1; j >= 0; j--)
        {
            var temp = input[0];
            input[0] = input[j];
            input[j] = temp;
            Heapify(input, j, 0);
        }
    }

    public static int[] Main(int[] input)
    {
        SortHeap(input, input.Length);
        return input;
    }
}
```

Раздел 35.2: Основная информация о пирамидальной сортировке

Пирамидальная сортировка — это метод сортировки сравнением, основанный на такой структуре данных как двоичная куча. Она похожа на сортировку выбором, где мы сначала ищем максимальный элемент и помещаем его в конец. Далее мы повторяем ту же операцию для оставшихся элементов.

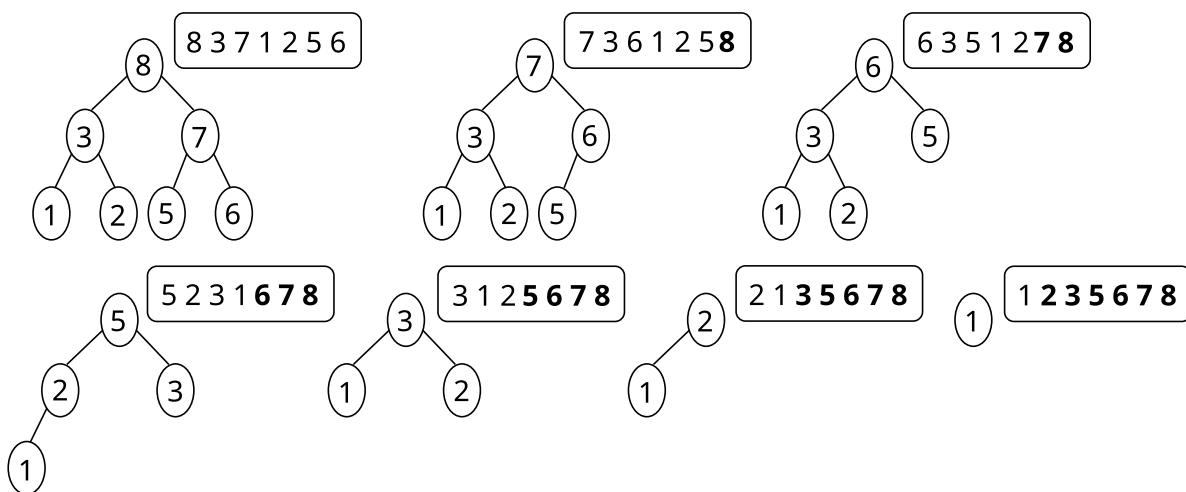
Псевдокод для пирамидальной сортировки:

```
function heapsort(input, count)
    heapify(a, count)
    end <- count - 1
    while end >= 0 do
        swap(a[end], a[0])
        end<-end-1
        restore(a, 0, end)

function heapify(a, count)
    start <- parent(count - 1)
    while start >= 0 do
        restore(a, start, count - 1)
        start <- start - 1
```

Пример пирамидальной сортировки:

Пример:-На рис. показаны шаги пирамидальной сортировки списка (2 3 7 1 8 5 6)



Дополнительная память: $O(1)$

Временная сложность: $O(n \log n)$

Глава 36: Сортировка циклом

Раздел 36.1: Реализация на псевдокоде

```
(input)
output = 0
for cycleStart from 0 to length(array) - 2
    item = array[cycleStart]
    pos = cycleStart
    for i from cycleStart + 1 to length(array) - 1
        if array[i] < item:
            pos += 1
    if pos == cycleStart:
        continue
    while item == array[pos]:
        pos += 1
    array[pos], item = item, array[pos]
    writes += 1
    while pos != cycleStart:
        pos = cycleStart
        for i from cycleStart + 1 to length(array) - 1
            if array[i] < item:
                pos += 1
        while item == array[pos]:
            pos += 1
        array[pos], item = item, array[pos]
        writes += 1
return outout
```

Глава 37: Четно-нечетная сортировка

Раздел 37.1: Основная информация о четно-нечетной сортировке

Четно-нечетная сортировка или сортировка по кирпичикам - это простой алгоритм сортировки, разработанный для использования на параллельных процессорах с локальным соединением. Работает он путем сравнения всех нечетных- четных проиндексированных пар соседних элементов в списке, и, если пара находится в неправильном порядке, то элементы переставляются. Следующий шаг повторяется для четных-нечетных проиндексированных пар. Затем нечетные-четные и четные-нечетные шаги чередуются до тех пор, пока список не будет отсортирован.

Псевдокод для четно-нечетной сортировки:

```
if n>2 then
    1. apply odd-even merge(n/2) recursively to the even subsequence a0, a2, ..., an-2 and
       odd subsequence a1, a3, , ..., an-1
    2. comparison [i : i+1] for all i element {1, 3, 5, 7, ..., n-3}
else
    comparison [0 : 1]
```

На Википедии есть лучшая иллюстрация четно-нечетной сортировки:

[Четно-нечетная сортировка](#)

Пример четно-нечетной сортировки:

Неотсортировано

3	2	3	8	5	6	4	1		Этап 1 (чет.)
2	3	3	8	5	6	1	4		Этап 2(нечет.)
2	3	3	5	8	1	6	4		Этап 3 (чет.)
2	3	3	5	1	8	4	6		Этап 4(нечет.)
2	3	3	1	5	4	8	6		Этап 5 (чет.)
2	3	1	3	4	5	6	8		Этап 6(нечет.)
2	1	3	3	4	5	6	8		Этап 7 (чет.)
1	2	3	3	4	5	6	8		Этап 8(нечет.)
1	2	3	3	4	5	6	8		

Отсортировано

Реализация:

Используется C# для реализации четно-нечетного алгоритма сортировки:

```
public class OddEvenSort
{
    private static void SortOddEven(int[] input, int n)
    {
        var sort = false;

        while (!sort)
        {
            sort = true;
            for (var i = 1; i < n - 1; i += 2)
            {
                if (input[i] <= input[i + 1]) continue;
```

```

        var temp = input[i];
        input[i] = input[i + 1];
        input[i + 1] = temp;
        sort = false;
    }
    for (var i = 0; i < n - 1; i += 2)
    {
        if (input[i] <= input[i + 1]) continue;
        var temp = input[i];
        input[i] = input[i + 1];
        input[i + 1] = temp;
        sort = false;
    }
}
public static int[] Main(int[] input)
{
    SortOddEven(input, input.Length);
    return input;
}
}

```

Дополнительная память: $O(n)$

Временная сложность: $O(n)$

Глава 38: Сортировка выбором

Раздел 38.1: Реализация на Elixir

```
defmodule Selection do

  def sort(list) when is_list(list) do
    do_selection(list, [])
  end

  def do_selection([head| []], acc) do
    acc ++ [head]
  end

  def do_selection(list, acc) do
    min = min(list)
    do_selection(:lists.delete(min, list), acc ++ [min])
  end

  defp min([first| [second| []]]) do
    smaller(first, second)
  end

  defp min([first| [second| tail]]) do
    min([smaller(first, second)| tail])
  end

  defp smaller(e1, e2) do
    if e1 <= e2 do
      e1
    else
      e2
    end
  end
end
Selection.sort([100,4,10,6,9,3])
|> IO.inspect
```

Раздел 38.2: Основная информация о сортировке выбором

Сортировка выбором - это алгоритм сортировки, в частности сортировка сравнения на месте. Сортировка имеет временную сложность $O(n^2)$, что делает ее неэффективной в больших списках и обычно работает хуже, чем сортировка вставкой. Сортировка выбором отличается простотой и имеет преимущества в производительности по сравнению с более сложными алгоритмами в определенных ситуациях, в частности там, где дополнительная память ограничена.

Алгоритм разделяет входной список на две части: на подсписок уже отсортированных элементов, который строится в начале списка слева направо, и на подсписок оставшихся элементов,

которые занимают оставшуюся часть списка.Изначально отсортированный подсписок пуст, а несортированный подсписок - это весь входной список. Алгоритм работает следующим образом: находится наименьший(или наибольший, в зависимости от порядка сортировки) элемент в неотсортированном подсписке, обменивается с крайним левым элементом из неотсортированного подсписка, затем граница отсортированного подсписка перемещается направо.

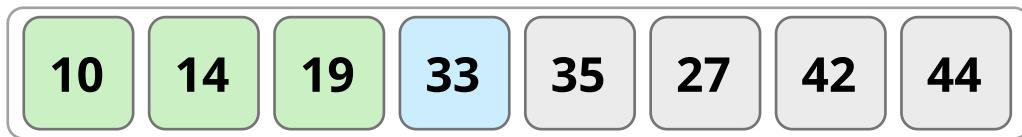
Псевдокод для сортировки выбором:

```
function select(list[1..n], k)
    for i from 1 to k
        minIndex = i
        minValue = list[i]
        for j from i+1 to n
            if list[j] < minValue
                minIndex = j
                minValue = list[j]
        swap list[i] and list[minIndex]
    return list[k]
```

Визуализация сортировки выбором:

[Сортировка выбором](#)

Пример Сортировки Выбором:



Дополнительная память: $O(n)$

Временная сложность: $O(n^2)$

Раздел 38.3: Реализация Сортировки выбором на C#

Используется C# для реализации алгоритма сортировки выбора:

```
public class SelectionSort
{
    private static void SortSelection(int[] input, int n)
    {
        for (int i = 0; i < n - 1; i++)
        {
            var minId = i;
            int j;
            for (j = i + 1; j < n; j++)
            {
                if (input[j] < input[minId]) minId = j;
            }
            var temp = input[minId];
            input[minId] = input[i];
            input[i] = temp;
        }
    }
    public static int[] Main(int[] input)
    {
        SortSelection(input, input.Length);
        return input;
    }
}
```

Глава 39: Поиск

Раздел 39.1: Бинарный поиск

Введение

Бинарный поиск - алгоритм поиска “разделяй и властвуй”. Он использует время $O(\log n)$ для нахождения элемента в поисковом пространстве, где n — размер поискового пространства.

Бинарный поиск работает за счет сокращения вдвое пространства поиска вдвое на каждой итерации после сравнения целевого значения со средним значением поискового пространства.

Чтобы использовать бинарный поиск, поисковое пространство должно быть упорядочено (отсортировано) каким-либо образом. Дублирующие записи (сравниваемые по функции сравнения) нельзя различить, хотя они не нарушают свойство бинарного поиска.

Обычно мы используем меньше ($<$) в качестве функции сравнения. Если $a < b$, она вернёт истину. Если a не меньше b и b не меньше a , то a и b равны.

Пример вопроса

Вы экономист, но довольно плохой. Перед вами поставлена задача поиска равновесной цены (то есть цены, где предложение равно спросу) на рис.

Помните, что чем выше цена, тем больше предложение и тем меньше спрос...

Поскольку ваша компания очень эффективна в вычислении рыночных сил, вы можете мгновенно получить предложение и спрос в единицах измерения риса, когда цена на рис установлена по определенной цене p .

Ваш босс хочет получить равновесную цену как можно скорее, но говорит, что равновесная цена может быть положительным целым числом, не превосходящим 10^{17} и гарантированно будет 1 положительное целое решение в диапазоне. Так что приступайте к работе, прежде чем ее потерять!

Вы можете вызвать функции `getSupply(k)` и `getDemand(k)`, которые будут делать именно то, что указано в задаче.

Объяснение примера

Здесь наше поисковое пространство составляет от 1 до 10^{17} . Таким образом, линейный поиск неосуществим.

Обратите внимание, что, когда k растёт, `getSupply(k)` увеличивается, а `getDemand(k)` уменьшается. Таким образом, для любого $x > y$, `getSupply(x) - getDemand(x) > getSupply(y) - getDemand(y)`. Таким образом, это поисковое пространство монотонно и мы можем использовать двоичный поиск.

Следующий псевдокод демонстрирует использование бинарного поиска.

```
high = 10000000000000000000 // Верхняя грань поиска
low = 1                      // Нижняя грань поиска
```

```
while high - low > 1           // Пока есть в диапазоне более 1 элемента
mid = (high + low) / 2          // Берем середину
supply = getSupply(mid)
demand = getDemand(mid)
if supply > demand           // Если текущее число больше искомого
high = mid                     // Решение находится в меньшей половине
else if demand > supply
low = mid                      // Решение находится в большей половине
else                           // Условие supply==demand
return mid                     // Решение найдено!
```

Алгоритм работает за $O(\log 10^7)$. Можно обобщить до $O(\log S)$ времени, где S — размер пространства поиска, поскольку на каждой итерации цикла while мы вдвое сокращаем пространство поиска (от [низкого:высокого] до [низкого:среднего] или [среднего:высокого])

Реализация бинарного поиска с помощью рекурсии на С

```
//Функция возвращает индекс искомого элемента
// Если элемент не был найден, то функция вернет -1
int binsearch(int a[], int x, int low, int high) {
    int mid;
    // low больше high, значит, элемент не был найден
    if (low > high)
        return -1;
    // Берем середину
    mid = (low + high) / 2;
    // Элемент найден
    if (x == a[mid])
        return (mid);
    } else
    // Если середина больше, то
    if (x < a[mid])
        // Ищем в меньшей половине
        binsearch(a, x, low, mid - 1);
    } else {
        // Ищем в большей половине
        binsearch(a, x, mid + 1, high);
    }
}
```

Раздел 39.2: Рабин Карп

Алгоритм Рабина – Карпа или алгоритм Карпа – Рабина — это алгоритм поиска строк, который использует хеширование для нахождения любого набора шаблонов строк в тексте. Его среднее и лучшее время работы составляет $O(n+m)$ в пространстве $O(p)$, но его худшее время $O(nm)$, где n — длина текста, m — длина шаблона.

Реализация алгоритма в java для сопоставления строк

```
void RabinfindPattern(String text, String pattern){
/*
q - Простое число
p - хеш-значение для pattern
t - хеш-значение для text
d - мощность алфавита
*/
int d=128;
int q=100;
int n=text.length();
```

```

int m=pattern.length();
int t=0,p=0;
int h=1;
int i,j;
// Вычисляем хеш-значение для h,
for (i=0;i<m-1;i++)
    // h - коэффициент хеша для первой буквы подстроки
    h = (h*d)%q;
// Вычисляем хеш для подстроки начала text'a (размером m) и pattern
for (i=0;i<m;i++){
p = (d*p + pattern.charAt(i))%q;
t = (d*t + text.charAt(i))%q;
}
// Поиск шаблона в строке
for(i=0;i<n-m;i++){
    // Если хеши p и t совпадают
    // проверим на совпадение символов посимвольно
    if(p==t){
        for(j=0;j<m;j++)
            // Если нашли несовпадение
            if(text.charAt(j+i)!=pattern.charAt(j))
                break;
            // Если мы дошли до конца цикла и i - адекватное число
            // Мы нашли шаблон в тексте по индексу i
            if(j==m && i>=start)
                System.out.println("Pattern match found at index "+i);
    }
    // Если ещё не конец текста
    if(i<n-m){
        // Перевычислим хеш для следующей подстроки
        // Т.к мы текущую подстроку передвигаем на символ вперед
        // Вычитаем из t хеш первой буквы и прибавляем хеш последней
        t =(d*(t - text.charAt(i)*h) + text.charAt(i+m))%q;
        // Если Хеш оказался отрицательным, прибавим q
        if(t<0) t=t+q;
    }
}
}

```

При расчете хеш-значения мы делим его на простое число, чтобы избежать столкновения. После деления на простое число шансы столкновения будут меньше, но тем не менее есть шанс, что значение хеша может быть одинаковым для двух строк, так что, когда мы получим совпадение, мы должны проверить его символ за символом, чтобы убедиться, что мы получили правильное совпадение.

$$t = (d * (t - \text{text.charAt}(i) * h) + \text{text.charAt}(i+m))$$

Это делается для того, чтобы пересчитать хеш-значение для шаблона, сначала удалив левый символ, а затем добавив новый символ из текста.

Раздел 39.3: Анализ линейного поиска (худший, средний и лучший случаи)

У нас может быть три случая для анализа алгоритма:

1.Худший случай

2.Средний случай

3.Лучший случай

```
#include <stdio.h>
// Линейный поиск x в arr[]. Если x находится в arr[], вернем его индекс,
// Иначе вернем -1
int search(int arr[], int n, int x) {
    int i;
    // Пробегаем по массиву arr[] от 0 до n
    for (i=0; i<n; i++) {
        // Если x найден, вернем его индекс
        if (arr[i] == x)
            return i;
    }
    return -1;
}

/* Программа-тестер*/
int main() {
    int arr[] = {1, 10, 30, 15};
    int x = 30;
    // n - количество элементов в arr[]
    // Делим на размер arr[0], так как
    // каждый элемент может занимать несколько байт
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("%d is present at index %d", x, search(arr, n, x));
    // Задержка вывода программы
    // Нажатием любой клавиши закончим программу
    getchar();
    return 0;
}
```

Анализ худшего случая (обычно проводится)

В худшем случае мы вычисляем верхнюю границу времени работы алгоритма. Мы должны знать о том, что является причиной проведения максимального числа операций. Для линейного поиска худший случай возникает, когда элемент, подлежащий поиску (x в указанном выше коде), не присутствует в массиве. Когда x не присутствует, функции поиска() сравнивают его со всеми элементами $\text{arr}[]$ один за другим. Следовательно, в худшем случае времененная сложность линейного поиска будет $O(n)$.

Анализ среднего случая (иногда проводится)

В среднем случае мы берём все возможные входные данные и рассчитываем время вычисления для них. Суммируем все рассчитанные значения и делим сумму на общее число вводимых ресурсов. Мы должны знать (или предугадать) распределение случаев. Для задачи линейного поиска давайте предположим, что все случаи равномерно распределены (включая случай, когда x не присутствует в массиве). Таким образом, мы суммируем все случаи и делим сумму на $(n+1)$. Ниже приведено значение средней временной сложности случая.

$$\text{Время Среднего случая} = \frac{\sum_{i=1}^{n+1} O(i)}{n+1} = \frac{O((n+1)*(n+2)/2)}{n+1} = O(n)$$

Анализ наилучшего случая (подделка)

В лучшем случае мы вычисляем нижнюю границу времени работы алгоритма. Мы должны знать, что вызывает минимальное число операций, подлежащих выполнению. В задаче линейного поиска лучший случай возникает тогда, когда x присутствует на первой позиции. Число операций в лучшем случае является постоянным (не зависящим от n). Таким образом, временная сложность в лучшем случае будет $O(1)$. Чаще всего мы проводим анализ худшего случая для анализа алгоритмов. В худшем случае мы гарантируем верхний предел времени работы алгоритма, который является хорошей информацией. Средний анализ не является простым в большинстве случаев и проводится редко. В среднем мы должны знать (или предсказывать) математическое распределение всех возможных входных данных. Анализ в лучшем случае — подделка. Гарантия нижней границы алгоритма не дает никакой информации, так как в худшем случае алгоритм может работать годами.

Для некоторых алгоритмов все случаи асимптотически одинаковы, т.е. нет худших и лучших случаев. Например, сортировка слиянием выполняет операции $O(n\log n)$ во всех случаях. Большинство других алгоритмов сортировки имеют худшие и лучшие случаи. Например, в типичной реализации быстрой сортировки (где поворот выбран в качестве углового элемента) худший случай, когда входной массив уже отсортирован, и лучший случай, когда узловые элементы всегда делят массив на две половины. Для сортировки вставками худший случай возникает при обратной сортировке массива и лучший случай возникает тогда, когда массив сортируется в том же порядке, что и вывод.

Раздел 39.4: Бинарный поиск: на отсортированных числах

Очень легко показать бинарный поиск на числах , используя псевдокод

```
int array[1000] = { /*отсортированный массив данных*/ };
// Количество обрабатываемых чисел
int N = 100;
// Верхняя, нижняя грани и середина
int high, low, mid;
// Искомое значение
int x;
low = 0;
high = N -1;
while(low < high)
{
    // Берем середину
    mid = (low + high)/2;
    //Если искомое число больше
    if(array[mid] < x)
        // Смещаем нижнюю грань на середину
        low = mid + 1;
    else
        // Смещаем верхнюю грань на середину
        high = mid;
    }
if(array[low] == x)
// Значение найдено
else
// Значение не найдено
```

Не пытайтесь вернуть значение раньше времени, сравнивая `array[mid]` с `x` для равенства. Дополнительное сравнение может только замедлить код. Обратите внимание, что нужно добавить ещё одно сравнение, дабы избежать попадания в ловушку целочисленного деления, которое всегда округляет вниз.

Интересно отметить, что вышеупомянутая версия бинарного поиска позволяет найти наименьшее число `X` в массиве. Если массив содержит дубликаты `x`, алгоритм может быть слегка изменен, чтобы он вернул наибольшую частоту `x`, просто добавив условие:

```
while(low < high)
{
    // Ищем середину
    // Мы не складываем high+low, чтобы избежать переполнения
    mid = low + ((high - low) / 2);
    // Если x больше ИЛИ есть дубликат с большим индексом
    if(array[mid] < x || (array[mid] == x && array[mid + 1] == x))
        // Смещаем нижнюю грань
        low = mid + 1;
    else
```

```
// Смещаем верхнюю грань
high = mid;
}
```

Заметим, что вместо того, чтобы делать $mid = (\text{low}+\text{high}) / 2$, можно также попробовать $mid = \text{low} + (\text{high} - \text{low}) / 2$ для реализаций, таких как реализация Java, чтобы снизить риск получения переполнения для действительно больших входных данных.

Раздел 39.5: Линейный поиск

Линейный поиск — простой алгоритм. Он обходит элементы до тех пор, пока не будет найден запрос, линейным алгоритмом его делает сложность $O(n)$, где n - число элементов, через которые нужно пройти.

Почему $O(n)$? В худшем случае вам придется просмотреть все n пунктов.

Это можно сравнить с поиском книги в стопке книг: просмотрите их все, пока не найдете ту, которая вам нужна.

Ниже приводится реализация на Python:

```
# searchable_list - поисковый список,
# query - искомый элемент
# Возвращает True, если элемент в списке есть, иначе False
def linear_search(searchable_list, query):
    for x in searchable_list:
        if query == x:
            return True
    return False
linear_search(['apple', 'banana', 'carrot', 'fig', 'garlic'], 'fig') #возвращает True
```

Глава 40: Поиск подстроки

Раздел 40.1: Введение в алгоритм Кнута-Морриса-Пратта (КМП)

Предположим, что у нас есть *текст* и *шаблон*. Нам нужно определить, существует ли шаблон в тексте или нет. Например:

+-----+-----+-----+-----+-----+-----+-----+
Index 0 1 2 3 4 5 6 7
+-----+-----+-----+-----+-----+-----+-----+
Text a b c b c g l x
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+
Index 0 1 2 3
+-----+-----+-----+-----+
Pattern b c g l
+-----+-----+-----+-----+

Этот шаблон *существует* в *тексте*. Таким образом, наш поиск по подстроке должен возвращать **3**, индекс позиции, с которой начинается этот *шаблон*. Так как же работает наша процедура поиска подстрок грубой силы?

Что мы обычно делаем: начинаем с **0-го** индекса текста и **0-го** индекса нашего *шаблона и сравниваем **Text[0]** (текст) с **Pattern[0]** (шаблон). Поскольку они не совпадают, мы переходим к следующему индексу нашего *текста* и сравниваем **Text[1]** с **Pattern[0]**. Поскольку это совпадение, мы увеличиваем индекс нашего *шаблона* и индекс *текста*. Мы сравниваем **Text[2]** с **Pattern[1]**. Они также совпадают. Следуя процедуре, описанной ранее, теперь мы сравним **Text[3]** с **Pattern[2]**. Поскольку они не совпадают, мы начинаем со следующей позиции, где начали искать совпадение. Это *индекс 2* текста. Мы сравниваем **Text[2]** с **Pattern[0]**. Они не совпадают. Затем, увеличивая индекс *текста*, мы сравниваем **Text[3]** с **Pattern[0]**. Они совпадают. Снова **Text[4]** и **Pattern[1]** совпадают, **Text[5]** и **Pattern[2]** совпадают, **Text[6]** и **Pattern[3]** совпадают. Поскольку мы достигли конца нашего *шаблона*, мы возвращаем индекс, с которого началось наше совпадение, то есть **3**. Если наш *шаблон* был: **bcll**, это означает, что если *шаблон* не существует в нашем *тексте*, наш поиск должен вернуть исключение или **-1** или любое другое предопределенное значение. Мы можем ясно видеть, что в худшем случае этот алгоритм будет занимать $O(mn)$ времени, где **m** - длина *текста*, а **n** - длина *шаблона*. Как уменьшить это время? Вот тут-то и появляется алгоритм поиска подстроки КМП.

Алгоритм поиска строк Кнута-Морриса-Пратта или алгоритм КМП ищет вхождения «шаблона» в основном «тексте», используя наблюдение о том, что при возникновении несоответствия само слово содержит достаточно информации, чтобы определить, где может начаться следующее совпадение, таким образом, минуя пересмотр ранее согласованных символов.

Алгоритм был задуман в 1970 году Дональдом Кнутом и Воном Праттом и независимо Джеймсом Х. Моррисом. Трио опубликовало его совместно в 1977 году.

Давайте расширим наш пример *Text*(текста) и *Pattern*(шаблона) для лучшего понимания:

```

+-----+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10|11|12|13|14|15|16|17|18|19|20|21|22|
+-----+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Text | a | b | c | x | a | b | c | d | a | b | x | a | b | c | d | a | b | c | d | a | b | c | y |
+-----+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+-----+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-----+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | y |
+-----+---+---+---+---+---+---+---+

```

Сначала наши *Text* и *Pattern* совпадают до индекса **2**. **Text[3]** и **Pattern[3]** не совпадают. Поэтому наша цель - не возвращаться назад в этом *тексте*, то есть в случае несоответствия мы не хотим, чтобы наше

2 сопоставление начиналось снова с той позиции, с которой мы начали сравнение. Чтобы достичь этого, мы будем искать **суффикс** в нашем *шаблоне* прямо перед тем, как произойдет наше несовпадение (подстрока **abc**), которая также является **префиксом** подстроки нашего *шаблона*. В нашем примере, поскольку все символы уникальны, суффикса нет, это префикс нашей совпавшей подстроки. Так что это означает, что наше следующее сравнение начнется с индекса **0**. Подожди немного, скоро ты поймешь, почему мы это сделали. Далее мы сравниваем **Text[3]** с **Pattern[0]**, и он не совпадает. После этого для *Text* с индексом **4** по индекс **9** и для *Pattern* с индексом **0** по индекс **5** мы находим совпадение. Мы находим несоответствие в **Text[10]** и **Pattern[6]**. Таким образом, мы берем подстроку из *Pattern* прямо перед местом, где происходит несовпадение (подстрока **abcdabc**), мы проверяем суффикс, который также является префиксом этой подстроки. Здесь мы видим, что **ab** является как суффиксом, так и префиксом этой подстроки. Это означает, что, поскольку мы сопоставили до **Text[10]**, символами, стоящими прямо перед несоответствием, являются **ab**. Из этого мы можем сделать вывод, что, поскольку **ab** также является префиксом подстроки, которую мы взяли, нам не нужно снова проверять **ab**, и следующая проверка может начинаться с **Text[10]** и **Pattern[2]**. Нам не нужно было оглядываться назад на весь *Text*, мы можем начать непосредственно с того места, где произошло наше несоответствие. Теперь мы проверяем **Text[10]** и **Pattern[2]**, так как это несоответствие, а подстрока перед несоответствием (**abc**) не содержит суффикса, который также является префиксом, мы проверяем **Text[10]** и **Pattern[0]**, они не совпадают. После этого для *Text* начиная с индекса **11** и до индекса **17** и для *Pattern* с индексом **0** и до индекс **6**. Мы находим несоответствие в **Text[18]** и **Pattern[7]**. Итак, еще раз мы проверяем подстроку перед несовпадением (substring **abcdabc**) и находим **abc** как суффикс, так и префикс. Таким образом, поскольку у нас совпадало до **Pattern[7]**, **abc** должен быть перед **Text[18]**. Это означает, что нам не нужно сравнивать до **Text[17]**, и наше сравнение начнется с **Text[18]** и **Pattern[3]**. Таким образом, мы найдем совпадение и вернем **15**, которое является нашим начальным индексом совпадения. Вот как работает наш поиск по подстроке КМП с использованием информации о суффиксах и префиксах.

Теперь поговорим о том, как нам эффективно вычислить, если суффикс совпадает с префиксом, и в какой момент начать проверку, если есть несоответствие символа между *текстом* и *шаблоном*. Давайте посмотрим на пример:

```

+-----+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+-----+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Pattern | a | b | c | d | a | b | c | a |
+-----+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

Мы сгенерируем массив, содержащий необходимую информацию. Назовем этот массив **S**. Размер массива будет таким же, как длина шаблона. Поскольку первая буква *шаблона (Pattern)* не может быть суффиксом какого-либо префикса, мы положим **S[0] = 0**. Сначала мы берем **i = 1** и **j = 0**. На каждом шаге мы сравниваем **Pattern[i]** и **Pattern[j]** и увеличиваем **i**. Если есть совпадение, мы помещаем в **S[i] = j + 1** и увеличиваем **j**, если есть несоответствие, то мы проверяем предыдущую позицию значения **j** (если доступно) и устанавливаем **j = S[j-1]** (если **j** не равно **0**), мы продолжаем делать это до тех пор, пока **S[j]** не совпадет с **S[i]** или **j** не станет равно **0**. Для более позднего мы положим **S[i] = 0**. Для нашего примера:

	j	i
+-----+-----+-----+-----+-----+-----+-----+		
Index 0 1 2 3 4 5 6 7		
+-----+-----+-----+-----+-----+-----+-----+		
Pattern a b c d a b c a		
+-----+-----+-----+-----+-----+-----+-----+		

Pattern[j] и **Pattern[i]** не совпадают, поэтому мы увеличиваем **i** и, поскольку **j** равно **0**, мы не проверяем предыдущее значение и присваиваем **Pattern [i] = 0**. Если мы продолжаем увеличивать **i**, для **i = 4** мы получим совпадение, поэтому мы присваиваем **S[i] = S[4] = j + 1 = 0 + 1 = 1** и увеличиваем **j** и **i**. Наш массив будет выглядеть так:

	j	i
+-----+-----+-----+-----+-----+-----+-----+		
Index 0 1 2 3 4 5 6 7		
+-----+-----+-----+-----+-----+-----+-----+		
Pattern a b c d a b c a		
+-----+-----+-----+-----+-----+-----+-----+		
S 0 0 0 0 1		
+-----+-----+-----+-----+-----+-----+-----+		

Поскольку **Pattern[1]** и **Pattern[5]** совпадают, мы присваиваем **S[i] = S[5] = j + 1 = 1 + 1 = 2**. Если мы продолжим, мы найдем несоответствие для **j = 3** и **i = 7**. Поскольку **j** не равно **0**, присваиваем **j = S[j-1]**. И мы сравним символы в **i** и **j** одинаковые они или нет, так как они одинаковые, мы присваиваем **S[i] = j + 1**. Наш законченный массив будет выглядеть так:

+-----+-----+-----+-----+-----+-----+-----+		
S 0 0 0 0 1 2 3 1		
+-----+-----+-----+-----+-----+-----+-----+		

Это наш обязательный массив. В нем ненулевое значение **S[i]** означает, что существует суффикс длины **S[i]**, такой же, как и префикс в этой подстроке (подстрока от **0** до **i**), и следующее сравнение начнется с **S[i] + 1** позиции *Pattern*. Наш алгоритм генерации массива будет выглядеть так:

```
Procedure GenerateSuffixArray(Pattern):
    i := 1
    j := 0
    n := Pattern.length
```

```

while i is less than n
    if Pattern[i] is equal to Pattern[j]
        S[i]:= j +1
        j := j +1
        i := i +1
    else
        if j is not equal to 0
            j := S[j-1]else
            S[i]:=0
            i := i +1
    end if
end if
end while

```

Сложность по времени для построения этого массива составляет $O(n)$, а сложность пространства также равна $O(n)$. Чтобы убедиться, что вы полностью поняли алгоритм, попробуйте сгенерировать массив для шаблона **aabaabaa** и проверить, совпадает ли результат с [этим](#).

Теперь давайте выполним поиск по подстроке, используя следующий пример:

	Index	0	1	2	3	4	5	6	7	8	9	10	11
Text	a b x a b c a b c a b y												
Index	0 1 2 3 4 5												
Pattern	a b c a b y												
S	0 0 0 1 2 0												

У нас есть *Text*, *Pattern* и предварительно рассчитанный массив **S** с использованием нашей логики, представленной ранее. Мы сравниваем **Text[0]** и **Pattern[0]**, и они одинаковы. **Text[1]** и **Pattern[1]** одинаковы. **Text[2]** и **Pattern[2]** не совпадают. Мы проверяем значение в позиции прямо перед несоответствием. Поскольку **S[1]** равно **0**, нет суффикса, совпадающего с префиксом в нашей подстроке, и наше сравнение начинается с позиции **S[1]**, которая равна **0**. Таким образом, **Pattern[0]** не совпадает с **Text[2]**, поэтому мы идем дальше. Значение **Text[3]** такой же, как и значение **Pattern[0]**, а значит есть совпадение до **Text[8]** и **Pattern[5]** соответственно. Мы делаем шаг назад в массив **S** и находим **2**. Таким образом, это означает, что существует префикс длины **2**, который также является суффиксом этой подстроки (**abcab**), который является **ab**. Это также означает, что перед **Text[8]** стоит **ab**. Таким образом, мы можем спокойно игнорировать **Pattern[0]** и **Pattern[1]** и начать наше следующее сравнение с **Pattern[2]** и **Text[8]**. Если продолжать дальше, мы найдем образец в тексте. Наша процедура будет выглядеть так:

```

Procedure KMP(Text, Pattern)
GenerateSuffixArray(Pattern)

```

```

m := Text.Length
n := Pattern.Length
i := 0
j := 0
while i is less than m
    if Pattern[j] is equal to Text[i]
        j := j + 1
        i := i + 1
        if j is equal to n
            Return(j-i)
        elseif i < m and Pattern[j] is not equal to Text[i]
            if j is not equal to 0
                j = S[j-1]
            else
                i := i + 1
        end if
    end if
end while
Return -1

```

Временная сложность этого алгоритма, за исключением вычисления суффиксного массива, составляет $O(m)$. Поскольку $\text{GenerateSuffixArray}$ принимает $O(n)$, общая времененная сложность алгоритма КМП составляет: $O(m + n)$.

PS: Если вы хотите найти несколько вхождений Pattern (шаблона) в Text (текст), вместо того, чтобы возвращать значение, выведите его / сохраните и установите $j := S[j-1]$. Также сохраняйте флагок, чтобы отслеживать, нашли ли вы какое-либо вхождение или нет, и обрабатывайте его соответствующим образом.

Раздел 40.2: Введение в алгоритм Рабина-Карпа

Алгоритм Рабина-Карпа - это алгоритм поиска строк, созданный Ричардом М. Карпом и Майклом О. Рабином, который использует хеширование для поиска любого из набора шаблонных строк в тексте.

Подстрока - это другая строка, которая встречается в строке. Например, *ver* является подстрокой *stackoverflow*. Не путать с подпоследовательностью, потому что *cover* - это подпоследовательность той же строки. Другими словами, любое подмножество последовательных букв в строке является подстрокой данной строки.

В алгоритме Рабина-Карпа мы сгенерируем хеш нашего шаблона, который мы ищем, и проверим, совпадает ли хеш нашего текста с шаблоном или нет. Если он не совпадает, мы можем гарантировать, что шаблон не существует в тексте. Однако, если он совпадает, *pattern* **может** присутствовать в тексте. Давайте посмотрим на пример:

Допустим, у нас есть текст: **yeminsajid**, и мы хотим выяснить, существует ли в тексте шаблон **nsa**. Чтобы вычислить хеш и скользящий хеш, нам нужно использовать простое число. Это может быть любое простое число. Давайте возьмем **простое число = 11** для этого примера. Мы определим значение хеша, используя эту формулу:

$$(1\text{st letter}) X (\text{prime}) + (2\text{nd letter}) X (\text{prime})^1 + (3\text{rd letter}) X (\text{prime})^2 X + \dots$$

Мы будем обозначать:

a -> 1	g -> 7	m -> 13	s -> 19	y -> 25
b -> 2	h -> 8	n -> 14	t -> 20	z -> 26
c -> 3	i -> 9	o -> 15	u -> 21	
d -> 4	j -> 10	p -> 16	v -> 22	
e -> 5	k -> 11	q -> 17	w -> 23	
f -> 6	l -> 12	r -> 18	x -> 24	

Значение хеша **nsa** будет:

$$14 \times 11^0 + 19 \times 11^1 + 1 \times 11^2 = 344$$

Теперь мы находим скользящий хэш нашего текста. Если скользящий хэш совпадает со значением хэша нашего шаблона, мы проверим соответствуют строки или нет. Поскольку наш шаблон состоит из **3** букв, мы возьмем первые **3** буквы **yem** из нашего текста и вычислим значение хеша. Мы получили:

$$25 \times 11^0 + 5 \times 11^1 + 13 \times 11^2 = 1653$$

Это значение не совпадает со значением хэша нашего шаблона. Так что строка здесь не существует. Теперь нам нужно рассмотреть следующий шаг для вычисления значения хеша нашей следующей строки **emi**. Мы можем рассчитать это по нашей формуле. Но это было бы довольно тривиально и стоило бы нам дороже. Вместо этого мы используем другую технику.

- Мы вычитаем значение **первой буквы предыдущей строки** из нашего текущего значения хеша. В этом случае **y**. Мы получаем, $1653 - 25 = 1628$.
- Мы делим разницу с нашим **простым числом**, которое в данном примере равно **11**. Получаем, $1628 / 11 = 148$.
- Мы добавляем **новую букву X (простое число) -1**, где **m** - длина шаблона, с частным, которое равно **i = 9**. Получаем, $148 + 9 \times 11^2 = 1237$.

Новое хеш-значение не равно нашему хеш-значению шаблона. Двигаясь дальше, для **n** мы получаем:

```
Previous String: emi
First Letter of Previous String: e(5)
New Letter: n(14)
NewString: "min"
1237 - 5 = 1232
1232 / 11 = 112
112 + 14 X 11^2 = 1806
```

Это не соответствует. После этого для **s** получаем:

```

Previous String: min
First Letter of Previous String: m(13)
New Letter: s(19)
NewString:"ins"
1806 - 13 = 1793
1793 / 11 = 163
163 + 19 X 11^2 = 2462

```

Это не соответствует. Далее, для **a**, мы получаем:

```

Previous String: ins
First Letter of Previous String: i(9)
New Letter: a(1)
NewString:"nsa"
2462 - 9 = 2453
2453 / 11 = 223
223 + 1 X 11^2 = 344

```

Совпало! Теперь мы сравним наш шаблон с текущей строкой. Поскольку обе строки совпадают, то подстрока существует в этой строке. И мы возвращаем стартовую позицию нашей подстроки.

Псевдокод:

Расчет хеша:

```

Procedure Calculate-Hash(String, Prime, x):
hash :=0
for m from 1 to x           hash value
    hash := hash +(Value of String[m])^(-1)
end for
Return hash

```

Пересчет хеша:

```

Procedure Recalculate-Hash(String, Curr, Prime, Hash):
Hash := Hash - Value of String[Curr]      First Letter of Previous String
Hash := Hash / Prime
m :=String.length
New:= Curr + m -1
Hash := Hash +(Value of String[New])^(-1)
Return Hash

```

Строка соответствия:

```

Procedure String-Match(Text, Pattern, m):
for i from m to Pattern-length + m -1

```

```

if Text[i] is not equal to Pattern[i]
    Return false
end if
end for
Return true

```

Rabin-Karp:

```

Procedure Rabin-Karp(Text, Pattern, Prime):
m := Pattern.Length
HashValue := Calculate-Hash(Pattern, Prime, m)
CurrValue := Calculate-Hash(Pattern, Prime, m)
for i from 1 to Text.length- m
    if HashValue == CurrValue and String-Match(Text, Pattern, i) is true
        Return i
    end if
    CurrValue := Recalculate-Hash(String, i+1, Prime, CurrValue)
end for

```

Если алгоритм не находит соответствия, он просто возвращает **-1**.

Этот алгоритм используется при обнаружении плагиата. Исходя из исходного материала, алгоритм может быстро искать в документе примеры предложений из исходного материала, игнорируя такие детали, как регистр и знаки препинания. Из-за обилия искомых строк алгоритмы поиска в одной строке здесь нецелесообразны. Опять же, **алгоритм Кнута-Морриса-Пратта** или **алгоритм поиска строк Бойера-Мура** являются более быстрыми алгоритмами поиска по одной строке шаблона, чем **алгоритм Рабина-Карпа**. Тем не менее, это алгоритм выбора для поиска по нескольким шаблонам. Если мы хотим найти любое из большого числа, скажем, k , шаблонов фиксированной длины в тексте, мы можем создать простой вариант алгоритма Рабина-Карпа.

Для текста шаблонов длины n и p общей длины m его среднее и наилучшее время выполнения равно $O(n+m)$ в пространстве $O(p)$, но его наихудшее время равно $O(nm)$.

Раздел 40.3: Реализация алгоритма КМП на Python

Стог сена: Стока, в которой необходимо найти данный шаблон.

Игла: Шаблон для поиска.

Временная сложность: Часть поиска (метод strstr) имеет сложность $O(n)$, где n - длина стога сена, но для построения таблицы префиксов также требуется анализ иглы, поэтому сложность $O(m)$ требуется для построения таблицы префиксов, где m - длина иглы.

Следовательно, общая времененная сложность для КМП составляет $O(n+m)$.

Пространственная сложность: $O(m)$ из-за таблицы префиксов на игле.

Примечание: Следующая реализация возвращает начальную позицию совпадения в стоге сена (если есть совпадение), в противном случае возвращается -1, для крайних случаев, например, если игла или стог сена является пустой строкой или игла не найдена в стоге сена.

```

def get_prefix_table(needle):
    prefix_set = set()
    n = len(needle)
    prefix_table = [0]*n
    delimiter = 1
    while(delimiter < n):
        prefix_set.add(needle[:delimiter])
        j = 1
        while(j < delimiter+1):
            if needle[j:delimiter+1] in prefix_set:
                prefix_table[delimiter] = delimiter - j + 1
                break
            j += 1
        delimiter += 1
    return prefix_table

def strstr(haystack, needle):
    haystack_len = len(haystack)
    needle_len = len(needle)
    if (needle_len > haystack_len) or (not haystack_len) or (not needle_len):
        return -1
    prefix_table = get_prefix_table(needle)
    m = i = 0
    while((i < needle_len) and (m < haystack_len)):
        if haystack[m] == needle[i]:
            i += 1
            m += 1
        else:
            if i != 0:
                i = prefix_table[i-1]
            else:
                m += 1
        if i == needle_len and haystack[m-1] == needle[i-1]:
            return m - needle_len
    else:
        return -1

if __name__ == '__main__':
    needle = 'abcaby'
    haystack = 'abxabcabcaby'
    print strstr(haystack, needle)

```

Раздел 40.4: Алгоритм КМП на Си

С учетом текста *txt* и шаблона *pat*, целью этой программы будет распечатать все вхождения *pat* в *txt*.

Примеры:

Входные данные:

```
txt[] = "THIS IS A TEST TEXT"  
pat[] = "TEST"
```

Выходные данные:

```
Pattern found at index 10
```

Входные данные:

```
txt[] = "AABAACAAADAAABAAABAA"  
pat[] = "AABA"
```

Выходные данные:

```
Pattern found at index 0  
Pattern found at index 9  
Pattern found at index 13
```

Реализация на языке Си:

```
// Программа на языке Си для реализации алгоритма КМП поиска  
// подстроки  
#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
void computeLPSArray(char *pat, int M, int *lps);  
  
void KMPSearch(char *pat, char *txt)  
{  
    int M = strlen(pat);  
    int N = strlen(txt);  
  
    int *lps = (int *)malloc(sizeof(int)*M);  
    int j = 0;  
  
    computeLPSArray(pat, M, lps);  
  
    int i = 0;  
    while (i < N)  
    {  
        if (pat[j] == txt[i])  
        {  
            j++;  
            i++;  
        }  
    }  
}
```

```

    if (j == M)
    {
        printf("Found pattern at index %d \n", i-j);
        j = lps[j-1];
    }

    else if (i < N && pat[j] != txt[i])
    {
        if (j != 0)
            j = lps[j-1];
        else
            i = i+1;
    }
}

free(lps);
}

void computeLPSArray(char *pat, int M, int *lps)
{
    int len = 0;
    int i;

    lps[0] = 0
    i = 1;

    while (i < M)
    {
        if (pat[i] == pat[len])
        {
            len++;
            lps[i] = len;
            i++;
        }
        else
        {
            if (len != 0)
            {

                len = lps[len-1];

            }
            else
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}
}

```

```
int main()
{
    char *txt = "ABABDABACDABABCABAB";
    char *pat = "ABABCABAB";
    KMPSearch(pat, txt);
    return 0;
}
```

Выходные данные:

Found pattern at index 10

Ссылка:

<https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/>

Глава 41: Поиск в ширину

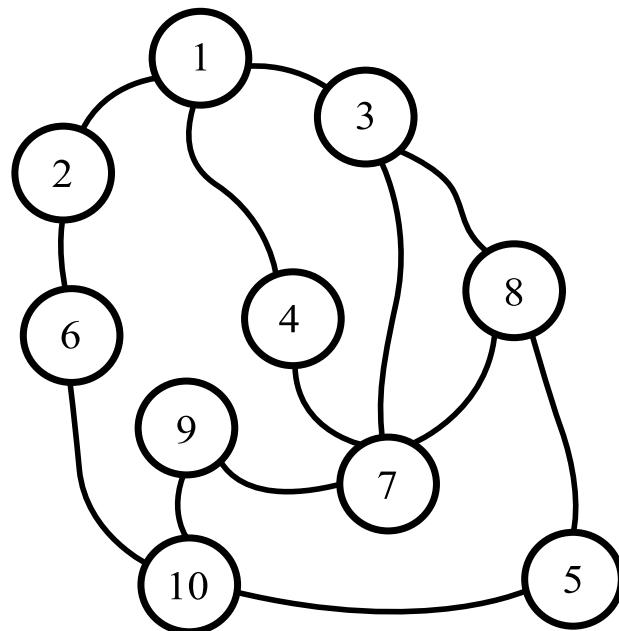
Раздел 41.1. Нахождение кратчайшего пути от источника к другим узлам

Поиск в ширину(BFS) - это алгоритм обхода или поиска таких структур данных, как дерево или граф. Он начинается с корня дерева (или некоторого произвольного узла графа, иногда называемого «ключом поиска») и в первую очередь исследует соседние узлы в первую очередь, прежде чем перейти к соседям следующего уровня. BFS был изобретен в конце 1950-х годов Эдвардом Форрестом Муром, который использовал его, чтобы найти кратчайший путь из лабиринта и, независимо с К. Ю. Ли, обнаружил, что его можно использовать в качестве алгоритма проводной маршрутизации в 1961 году.

Процесс работы алгоритма BFS описан в этих предложениях:

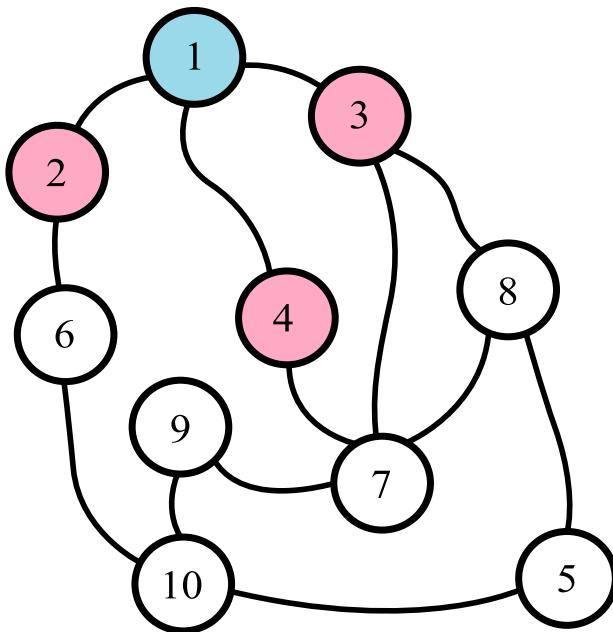
1. Мы не пройдем ни один узел более одного раза.
2. Исходный узел или узел, с которого мы начинаем, расположен на уровне 0.
3. Узлы, к которым мы можем напрямую обратиться из узла-источника, являются узлами уровня 1, узлы, к которым мы можем напрямую обратиться из узлов уровня 1, являются узлами уровня 2 и так далее.
4. Уровень обозначает расстояние по кратчайшему пути от источника.

Давайте посмотрим на пример:

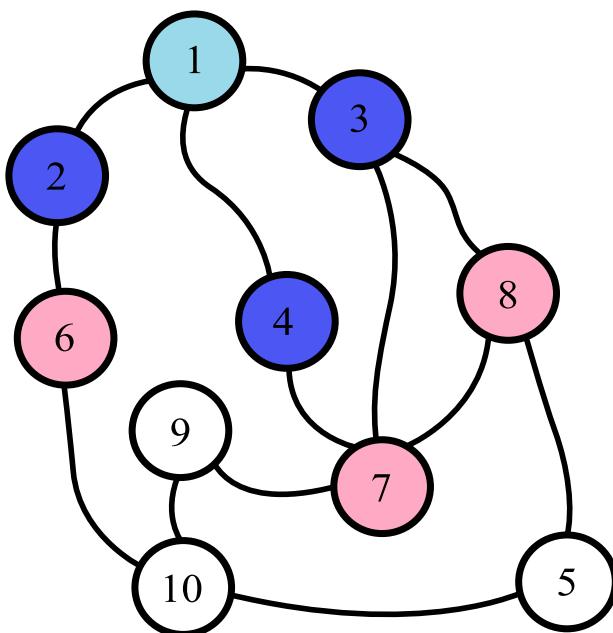


Давайте предположим, что этот граф представляет связь между некоторыми городами, где каждый узел означает город, а ребро между двумя узлами означает, что существует дорога, связывающая их. Мы хотим перейти от **узла 1** к **узлу 10**. Таким образом, **узел 1** является нашим **источником**, который является **уровнем 0**. Мы помечаем **узел 1** как посещенный.

Отсюда мы можем перейти к узлу 2, 3 и 4. Таким образом, они будут **уровнями** $(0 + 1) =$ **уровнями 1**. Теперь мы отметим их как посещенные и будем работать с ними.

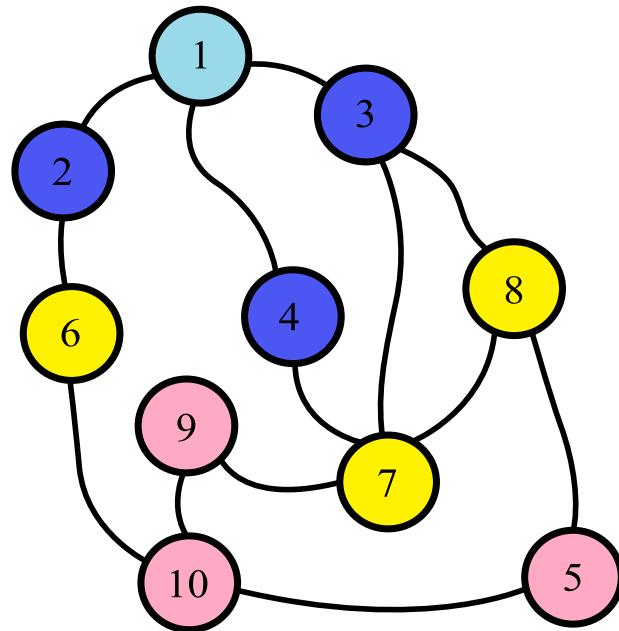


Окрашенные узлы посещаются. Узлы, с которыми мы сейчас работаем, отмечаются розовым цветом. Мы не будем посещать один и тот же узел дважды. Из **узлов 2, 3 и 4**, мы можем пойти к **узлам 6, 7 и 8**. Давайте отметим их как посещенные. Уровень этих узлов будет **уровень** $(1 + 1) =$ **уровень 2**.

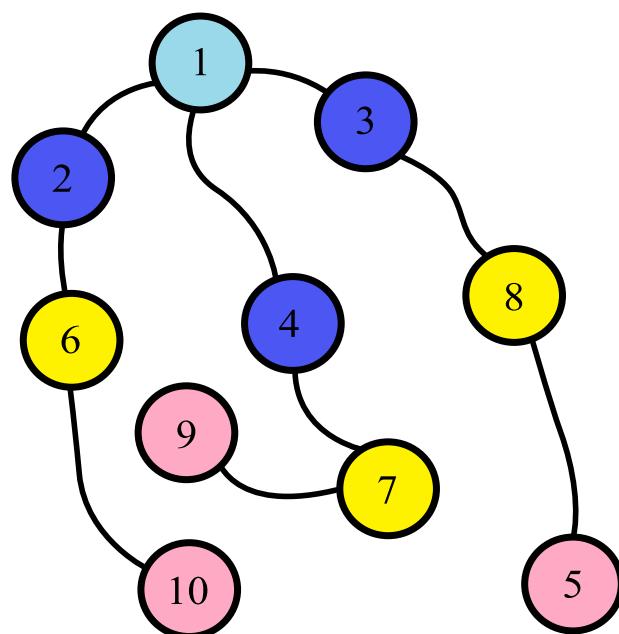


Если вы не заметили, уровень узлов просто обозначает кратчайшее расстояние от **источника**. Например: мы нашли **узел 8** на **уровне 2**. Таким образом, расстояние от **источника** до **узла 8** равно **2**.

Мы еще не достигли нашего целевого узла, то есть **узла 10**. Итак, давайте посетим следующие узлы. Мы можем напрямую перейти от **узла 6**, **7** и **узла 8**.



Мы видим, что мы нашли **узел 10** на **уровне 3**. Таким образом, кратчайший путь из **источника** к **узлу 10** равен **3**. Мы исследовали граф уровень за уровнем и нашли кратчайший путь. Теперь давайте удалим рёбра, которые мы не использовали:



После удаления ребер, которые мы не использовали, мы получаем дерево, называемое BFS деревом. Это дерево показывает кратчайший путь от **источника** ко всем остальным узлам.

Таким образом, наша задача состоит в том, чтобы перейти от узла **источника** к узлам **уровня 1**. Затем с **уровня 1** до **уровня 2** и так далее, пока мы не достигнем пункта назначения. Мы

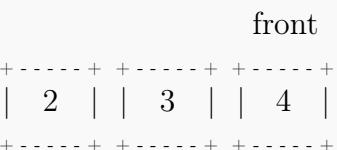
можем использовать *очередь* для хранения узлов, которые мы собираемся обработать. То есть для каждого узла, с которым мы собираемся работать, мы будем добавлять в очередь все другие узлы, которые могут быть напрямую пройдены или которые еще не пройдены в очереди.

Моделирование нашего примера:

Сначала мы помещаем источник в очередь. Наша очередь будет выглядеть так:



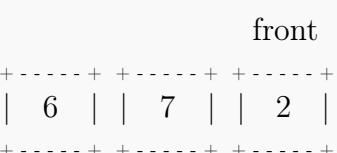
Уровень **узла 1** будет равен 0. $\text{level}[1] = 0$. Теперь мы начинаем наш BFS. Сначала мы извлекаем узел из нашей очереди. Мы получаем **узел 1**. Мы можем перейти к **узлу 4, 3 и 2** от него. Мы достигли этих узлов из **узла 1**. Таким образом, $\text{level}[4] = \text{level}[3] = \text{level}[2] = \text{level}[1] + 1 = 1$. Теперь мы помечаем их как посещенных и помещаем их в очередь.



Теперь мы извлекаем **узел 4** и работаем с ним. Мы можем перейти к **узлу 7** из **узла 4**. $\text{level}[7] = \text{level}[4] + 1 = 2$. Мы помечаем **узел 7** как посещенный и помещаем его в очередь.



Из **узла 3** мы можем перейти к **узлу 7** и **узлу 8**. Так как мы уже пометили **узел 7** как посещенный, мы пометим теперь **узел 8**. Мы меняем $\text{level}[8] = \text{level}[3] + 1 = 2$ и добавляем **узел 8** в очередь.



Этот процесс будет продолжаться до тех пор, пока мы не достигнем пункта назначения или очередь не станет пустой. Массив **уровней** предоставит нам расстояние по кратчайшему пути от **источника**. Мы можем инициализировать массив **уровней** значением **бесконечность**, которое отметит, что узлы еще не посещены. Наш псевдокод будет выглядеть так:

```
Procedure BFS(Graph, source):  
    Q = queue();  
    level[] = infinity  
    level[source] := 0
```

```

Q.push(source)
while Q is not empty
    u -> Q.pop()
    for all edges from u to v in Adjacency list
        if level[v] == infinity
            level[v] := level[u] + 1
            Q.push(v)
    end if
end for
end while
Return level

```

Перебирая массив **уровней**, мы можем узнать расстояние от источника до каждого узла. Например: расстояние до **узла 10** от **источника** будет сохранено в **level[10]**.

Иногда нам может потребоваться распечатать не только кратчайшее расстояние, но и путь, по которому мы можем перейти к нашему конечному узлу из **источника**. Для этого нам нужно сохранить массив **parent**. **parent[source]** будет равен NULL. Для каждого обновления в массиве **level** мы просто добавим **parent[v] := u** в наш псевдокод внутри цикла for. После завершения BFS, чтобы найти путь, мы будем перемещаться назад по массиву **parent**, пока не достигнем **источника**, который будет обозначен значением NULL.

Псевдокод будет выглядеть так:

//рекурсивный Procedure PrintPath(u) : if parent[u] is not equal to null PrintPath(parent[u]) end if print -> u	//итеративный Procedure PrintPath(u) : S = Stack() while parent[u] is not equal to null S.push(u) u := parent[u] end while while S is not empty print -> S.pop end while
---------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Сложность:

Мы посетили каждый узел и каждое ребро один раз. Таким образом, сложность будет **O(V+E)**, где **V** - количество узлов, а **E** - количество ребер.

Раздел 41.2. Поиск кратчайшего пути от источника в двумерном графе

Большую часть времени нам нужно будет искать кратчайший путь от одного источника ко всем остальным узлам или к конкретному узлу в 2D-графе. Скажем, например, что мы хотим выяснить, сколько ходов требуется, чтобы конь достиг определенного квадрата на шахматной доске, или у нас есть массив, в котором некоторые ячейки заблокированы, и мы должны найти кратчайший путь от одной ячейки к другой. Мы можем двигаться только горизонтально и вертикально. Однако диагональные ходы могут быть тоже возможны. Для этих случаев мы можем преобразовать квадраты или ячейки в узлы и легко решить эту проблему, используя BFS. Теперь наши **visited**, **parents** и **level** будут двумерными массивами. Для каждого узла мы рассмотрим все возможные ходы. Чтобы найти расстояние до определенного узла, мы также проверим, достигли ли мы пункта назначения.

Будет еще один массив, называемый массивом направлений. Он просто сохранит все возможные комбинации направлений, по которым мы можем идти. Допустим, для горизонтальных и вертикальных перемещений наши массивы направлений будут:

+-----+	-----+	-----+	-----+	-----+
dx	1	-1	0	0
+-----+	-----+	-----+	-----+	-----+
dy	0	0	1	-1
+-----+	-----+	-----+	-----+	-----+

Здесь dx представляет собой движение по оси x , а dy - движение по оси y . Опять же эта часть не обязательна. Вы также можете написать все возможные комбинации отдельно, но легче справиться с этим, используя массив направлений. Комбинации для диагональных ходов и ходов коня могут быть разнообразней, а их количество гораздо больше.

Дополнительная часть, на которую мы должны обратить внимание:

- Если какая-либо из ячеек заблокирована, то для каждого возможного хода мы будем проверять, заблокирована ли ячейка или нет.
- Мы также проверим, вышли ли мы за пределы, то есть пересекли ли мы границы массива.
- Количество строк и столбцов будет дано.

Наш псевдокод будет выглядеть так:

```
Procedure BFS2D(Graph, blocksign, row, column):
    for i from 1 to row
        for j from 1 to column
            visited[i][j] := false
        end for
    end for
    visited[source.x][source.y] := true
    level[source.x][source.y] := 0
    Q = queue()
    Q.push(source)
    m := dx.size
    while Q is not empty
        top := Q.pop
        for i from 1 to m
            temp.x := top.x + dx[i]
            temp.y := top.y + dy[i]
            if temp is inside the row and column and top
                does not equal to blocksign
                visited[temp.x][temp.y] := true
                level[temp.x][temp.y] := level[top.x][top.y] + 1
                Q.push(temp)
            end if
        end for
    end while
    Return level
```

Как мы уже обсуждали ранее, BFS работает только для невзвешенных графов. Для взвешенных графов нам понадобится алгоритм Дейкстры. Для циклов с отрицательным ребром нам нужен алгоритм Беллмана-Форда. Опять же, этот алгоритм является алгоритмом кратчайшего пути

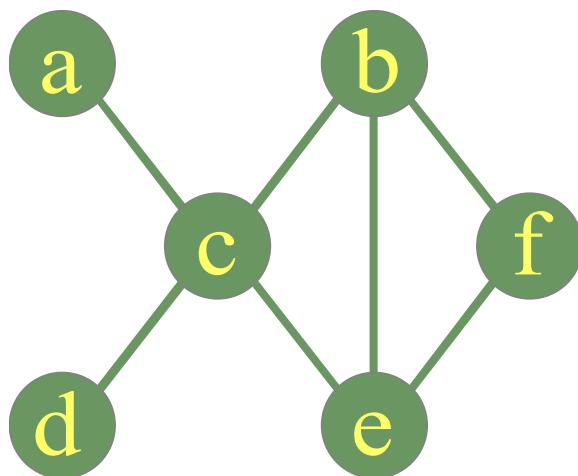
из одного источника. Если нам нужно узнать расстояние от каждого узла до всех других узлов, нам понадобится алгоритм Флойда-Уоршелла.

Раздел 41.3: Связные компоненты неориентированного графа с использованием BFS

BFS может использоваться для поиска связных компонент неориентированного графа. Мы также можем определить, связан ли данный граф или нет. Наше последующее обсуждение предполагает, что мы имеем дело с неориентированными графами. Определение связного графа:

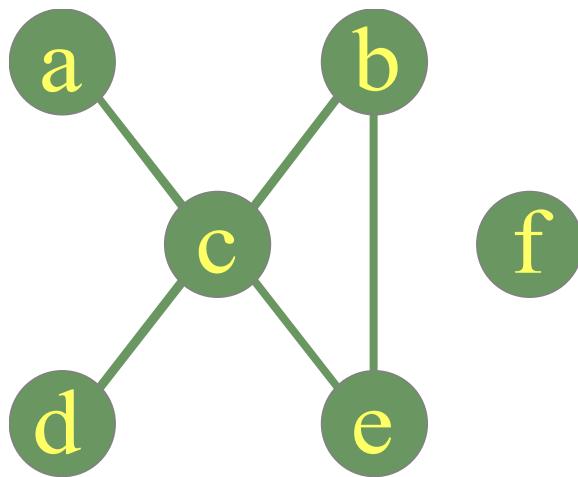
Граф связен, если между каждой парой вершин есть путь.

Ниже приведен **связный** граф.



Следующий граф не связан и имеет 2 компоненты связности:

1. Компонента связности 1: a, b, c, d, e
2. Компонента связности 2: f



BFS - это алгоритм обхода графа. Таким образом, начиная со случайного исходного узла, если по завершении алгоритма все узлы посещены, то граф связан, в противном случае он не связан.

Псевдокод для алгоритма.

```

boolean isConnected(Graph g)
{
    BFS(v)
    if(allVisited(g))
    {
        return true;
    }
    else return false;
}

```

Реализация на С для определения того, связан ли ненаправленный граф или нет:

```

#include <stdio.h>
#include <stdlib.h>
#define MAXVERTICES 100

void enqueue(int);
int deque();
int isConnected(char **graph, int noOfVertices);
void BFS(char **graph, int vertex, int noOfVertices);
int count = 0;

struct node
{
    int v;
    struct node *next;
};

typedef struct node Node;
typedef struct node *Nodeptr;

Nodeptr Qfront = NULL;
Nodeptr Qrear = NULL;
char *visited;

int main()
{
    int n, e;
    int i, j;
    char **graph;

    printf("Enter number of vertices:");
    scanf("%d", &n);

    if (n < 0 || n > MAXVERTICES)
    {
        fprintf(stderr, "Please enter a valid positive integer
                  from 1 to %d", MAXVERTICES);
        return -1;
    }

    graph = malloc(n * sizeof(char *));

```

```

visited = malloc(n * sizeof(char));

for (i = 0; i < n; ++i)
{
    graph[i] = malloc(n * sizeof(int));
    visited[i] = 'N';
    for (j = 0; j < n; ++j)
        graph[i][j] = 0;
}

printf("enter number of edges and then enter them in pairs:");
scanf("%d", &e);

for (i = 0; i < e; ++i)
{
    int u, v;
    scanf("%d%d", &u, &v);
    graph[u - 1][v - 1] = 1;
    graph[v - 1][u - 1] = 1;
}

if (isConnected(graph, n))
    printf("The graph is connected");
else printf("The graph is NOT connected\n");
}

void enqueue(int vertex)
{
    if (Qfront == NULL)
    {
        Qfront = malloc(sizeof(Node));
        Qfront->v = vertex;
        Qfront->next = NULL;
        Qrear = Qfront;
    }
    else
    {
        Nodeptr newNode = malloc(sizeof(Node));
        newNode->v = vertex;
        newNode->next = NULL;
        Qrear->next = newNode;
        Qrear = newNode;
    }
}

int dequeue()
{
    if (Qfront == NULL)
    {
        printf("Q is empty, returning -1\n");
        return -1;
    }
}

```

```

    }
else
{
    int v = Qfront->v;
    Nodeptr temp = Qfront;
    if (Qfront == Qrear)
    {
        Qfront = Qfront->next;
        Qrear = NULL;
    }
    else
        Qfront = Qfront->next;
    free(temp);
    return v;
}
}

int isConnected(char **graph, int noOfVertices)
{
    int i;

    BFS(graph, 0, noOfVertices);

    for (i = 0; i < noOfVertices; ++i)
        if (visited[i] == 'N')
            return 0;

    return 1;
}

void BFS(char **graph, int v, int noOfVertices)
{
    int i, vertex;
    visited[v] = 'Y';
    enqueue(v);
    while ((vertex = deque()) != -1)
    {
        for (i = 0; i < noOfVertices; ++i)
            if (graph[vertex][i] == 1 && visited[i] == 'N')
            {
                enqueue(i);
                visited[i] = 'Y';
            }
    }
}

```

Чтобы найти все связные компоненты неориентированного графа, нам нужно всего лишь добавить 2 строки кода в функцию BFS. Идея состоит в том, чтобы вызывать функцию BFS, пока все вершины не будут посещены.

Строки, которые будут добавлены:

```
printf("\nConnected component %d\n", ++count);
```

А также

```
printf("%d ", vertex + 1);
add this as first line of while loop in BFS
```

и еще мы определяем следующую функцию:

```
void listConnectedComponents(char **graph, int noOfVertices)
{
    int i;
    for (i = 0; i < noOfVertices; ++i)
    {
        if (visited[i] == 'N')
            BFS(graph, i, noOfVertices);
    }
}
```

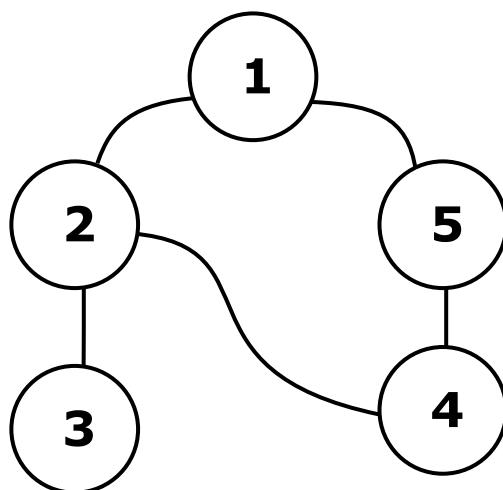
Глава 42: Поиск в глубину

Раздел 42.1: Введение в “поиск в глубину”

Поиск в глубину - это алгоритм обхода или поиска структур данных дерева или графа. Каждый обход начинается с корня и исследует как можно дальше вдоль каждой ветви перед возвратом. Алгоритм поиска в глубину был исследован французским математиком 19-го века Шарлем Пьером Тремо как стратегия решения лабиринтов.

Поиск в глубину - это систематический способ найти все вершины, достижимые из исходной вершины. Как и поиск в ширину, DFS (DFS - Поиск в глубину) пересекает связную компоненту данного графа и определяет оствовное дерево. Основная идея поиска в глубину заключается в методическом исследовании каждого ребра. Мы начинаем с разных вершин по мере необходимости. Как только мы обнаруживаем вершину, DFS начинает исследовать ее (в отличие от BFS, которая помещает вершину в очередь, чтобы потом ее исследовать).

Давайте посмотрим на пример. Мы пройдемся по этому графу:

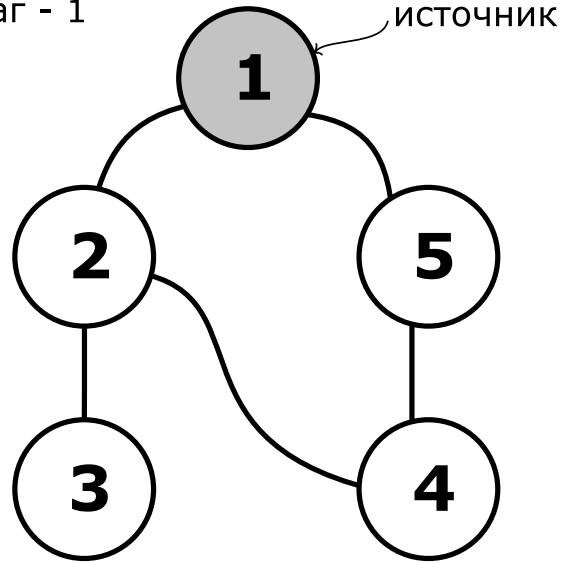


Мы пройдемся по графу, следуя этим правилам:

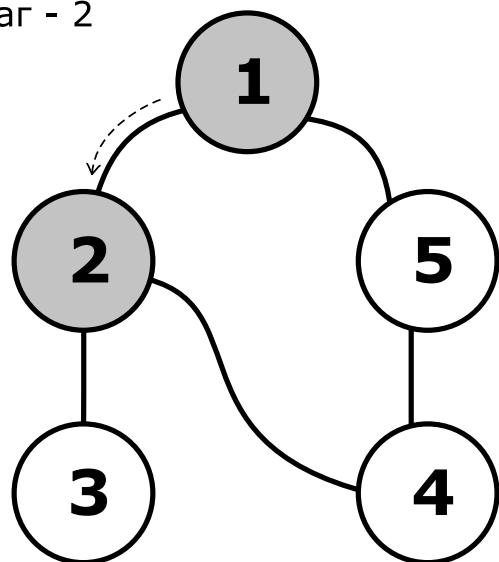
- Начнем с исходной вершины.
- Ни один узел не будет посещен дважды.
- Узлы, которые мы еще не посетили, будут окрашены в белый цвет.
- Узел, который мы посетили, но не посетили все его дочерние узлы, будет окрашен в серый цвет.
- Полностью пройденные узлы будут окрашены в черный цвет.

Давайте посмотрим на это шаг за шагом:

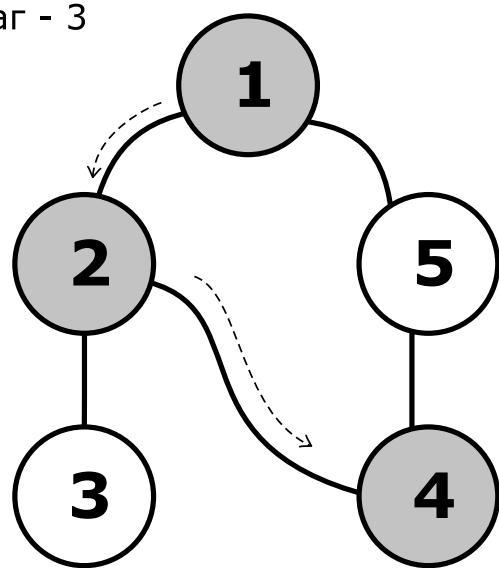
Шаг - 1



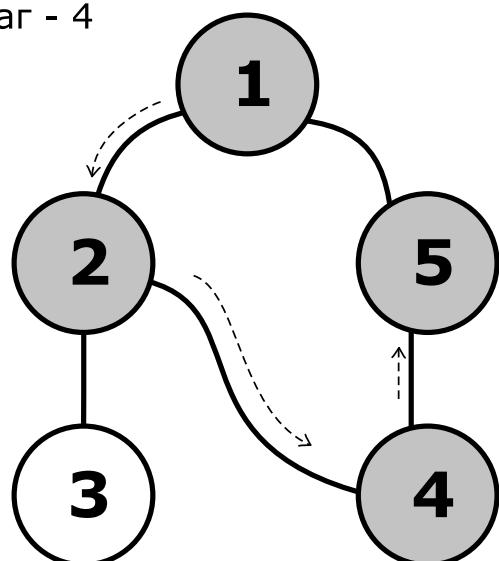
Шаг - 2



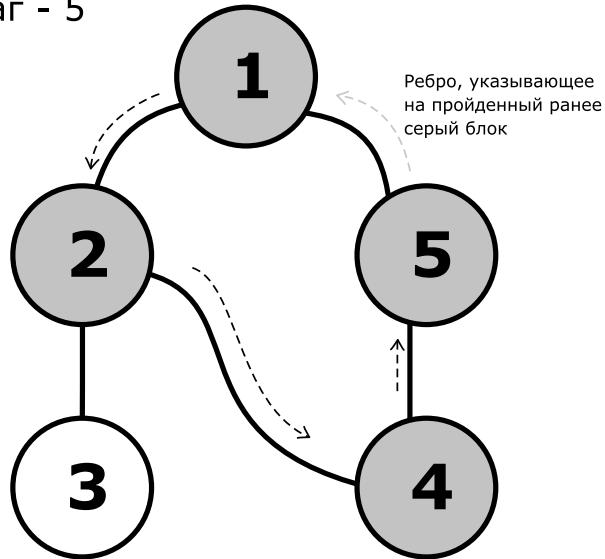
Шаг - 3



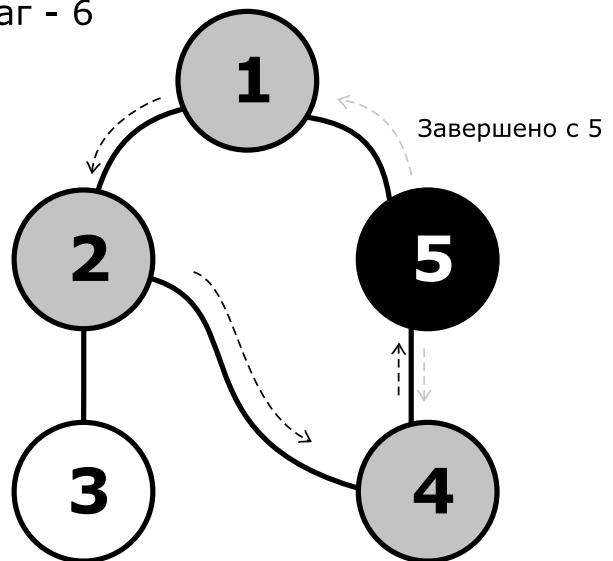
Шаг - 4



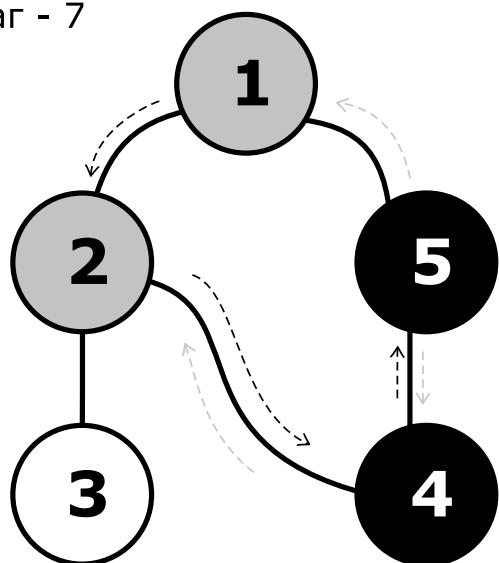
Шаг - 5



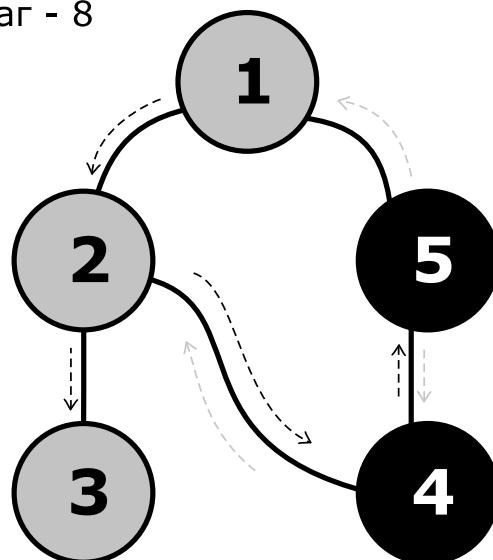
Шаг - 6



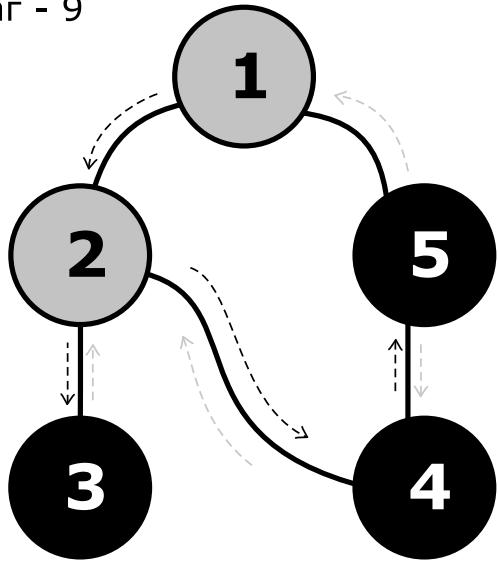
Шаг - 7



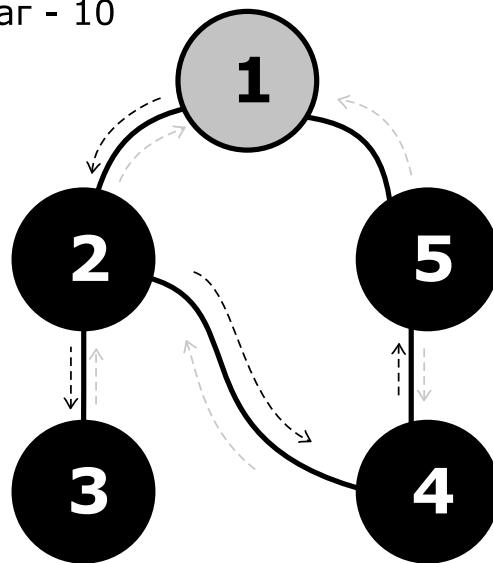
Шаг - 8



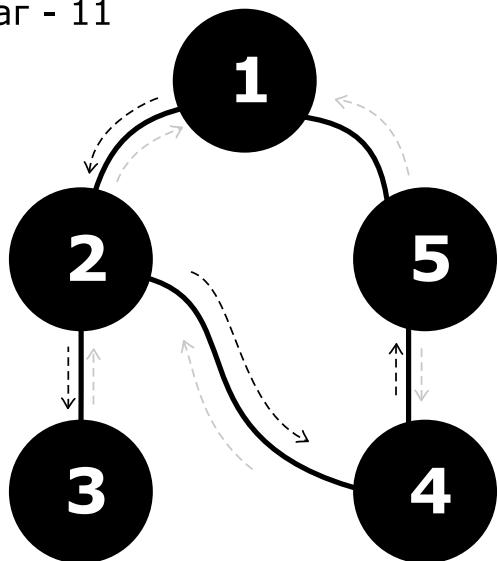
Шаг - 9



Шаг - 10



Шаг - 11



Мы видим одно важное и ключевое понятие – **обратное ребро**. Вы можете увидеть это. **5-1** называется обратным ребром. Это потому, что мы еще не закончили с **узлом-1**, поэтому переход от другого узла к **узлу-1** означает, что в графе есть цикл. В DFS, если мы можем перейти от одного серого узла к другому, мы можем быть уверены, что у графа есть цикл. Это один из способов обнаружения цикла в графе. В зависимости от **исходного** узла и порядка посещаемых нами узлов, мы можем найти все те ребра, которые являются **обратными**. Например: если бы мы сначала пошли в **5** из **1**, мы бы увидели, что **2-1** это обратное ребро.

Ребро, которое мы берем, чтобы перейти от серого узла к белому узлу, называется **ребром дерева**. Если мы оставим только **ребра дерева** и удалим остальные, мы получим **DFS дерево**.

В неориентированном графе, если мы можем посетить уже посещенный узел, это должно быть **обратное ребро**. Но для ориентированных графов мы должны проверить цвета. *Если и только если мы можем перейти от одного серого узла к другому серому узлу, то это ребро называется обратным ребром.*

В DFS мы также можем хранить временные метки для каждого узла, которые могут использоваться разными способами (например топологическая сортировка).

1. Когда узел v меняется с белого на серый, время записывается в $d[v]$.
2. Когда узел v изменяется с серого на черный, время записывается в $f[v]$.

Здесь $d[]$ означает время обнаружения, а $f[]$ означает время окончания. Наш псевдокод будет выглядеть так:

```
Procedure DFS(G):
    for each node u in V[G]
        color[u] := white
        parent[u] := NULL
    end for
    time := 0
    for each node u in V[G]
        if color[u] == white
            DFS-Visit(u)
        end if
    end for

Procedure DFS-Visit(u):
    color[u] := gray
    time := time + 1
    d[u] := time
    for each node v adjacent to u
        if color[v] == white
            parent[v] := u
            DFS-Visit(v)
        end if
    end for
    color[u] := black
    time := time + 1
    f[u] := time
```

Сложность:

Каждый узел и ребро посещаются один раз. Таким образом, сложность DFS составляет $O(V + E)$, где V обозначает количество узлов, а E обозначает количество ребер.

Применение поиска в глубину:

- Нахождение кратчайших путей между всеми вершинами неориентированного графа.
- Обнаружение цикла в графе.
- Нахождение пути.
- Топологическая сортировка.
- Проверка, является ли граф двудольным.
- Нахождение компоненты сильной связности.
- Решение головоломок с одним решением.

Глава 43: Хеш-функции

Раздел 43.1: Хеш-коды для общих типов в C#

Хеш-коды, созданные методом `GetHashCode()` для [встроенных](#) и общих типов C# из пространства имен `System`, показаны ниже.

Boolean

1, если значение истинно, в противном случае 0.

Byte, UInt16, Int32, UInt32, Single

Значение (при необходимости приведено к `Int32`).

SByte

```
((int)m_value ^ (int)m_value << 8);
```

Char

```
((int)m_value ^ (int)m_value << 16);
```

Int16

```
((int)((ushort)m_value) ^ ((int)m_value << 16));
```

Int64, Double

Сложение по модулю 2 между нижними и верхними 32 битами 64-битного числа

```
(unchecked((int)((long)m_value)) ^ (int)(m_value >> 32));
```

UInt64, DateTime, TimeSpan

```
((int)m_value ^ (int)(m_value >> 32));
```

Decimal

```
((((int *)&dbl)[0]) & 0xFFFFFFFF0) ^ ((int *)&dbl)[1];
```

Object

```
RuntimeHelpers.GetHashCode(this);
```

В реализации по умолчанию используется [индекс блока синхронизации](#).

String

Вычисление хеш-кода зависит от типа платформы (Win32 или Win64), возможности использования рандомизированного хеширования строк, режима отладки/выпуска. В случае платформы Win64:

```
int hash1 = 5381;
int hash2 = hash1;
int c;
```

```

char *s = src;
while ((c = s[0]) != 0) {
    hash1 = ((hash << 5) + hash1) ^ c;
    c = s[1];
    if (c == 0)
        break;
    hash2 = ((hash2 << 5) + hash2) ^ c;
    s += 2;
}
return hash1 + (hash2 * 1566083941);

```

ValueType

Первое нестатическое поле ищет и получает хеш-код. Если тип не имеет нестатических полей, возвращается хеш-код типа. Хеш-код статического члена не может быть взят, потому что, если этот член имеет тот же тип, что и исходный тип, вычисление заканчивается бесконечным циклом.

Nullable<T>

```
return hasValue ? value.GetHashCode() : 0;
```

Array

```

int ret = 0;
for (int i = (Length >= 8 ? Length - 8 : 0); i < Length; i++)
{
    ret = ((ret << 5) + ret) ^ comparer.GetHashCode(GetValue(i));
}

```

Ссылки:

- [GitHub .Net Core CLR](#)

Раздел 43.2: Введение в хеш-функции

Хэш-функция $h()$ - это произвольная функция, которая отображает данные $x \in X$ произвольного размера в значения $y \in Y$ фиксированного размера: $y = h(x)$. Хорошие хэш-функции имеют следующие ограничения:

- хэш-функции ведут себя как равномерное распределение
- хэш-функции являются детерминированными. $h(x)$ всегда должен возвращать одно и то же значение для данного x .
- быстрый расчет (время выполнения $O(1)$)

В общем случае размер хэш-функции меньше размера входных данных: $|y| < |x|$. Хэш-функции необратимы или, другими словами, они могут привести к коллизии: $\exists x_1, x_2 \in X, x_1 \neq x_2 : h(x_1) = h(x_2)$. X может быть конечным или бесконечным множеством, а Y - конечным множеством.

Хэш-функции используются во многих областях информатики, например, в разработке программного обеспечения, криптографии, базах данных, сетях, машинном обучении и так далее.

Существует много различных типов хеш-функций с различными специфическими для домена свойствами.

Часто хеш является целочисленным значением. В языках программирования существуют специальные методы для вычисления хеша. Например, в C# метод `GetHashCode()` для всех типов возвращает значение `Int32` (32-разрядное целое число). В Java каждый класс предоставляет метод `hashCode()`, который возвращает `int`. Каждый тип данных имеет собственные или определенные пользователем реализации.

Методы хеширования

Существует несколько подходов для определения хеш-функции. Без потери общности, пусть $x \in X = \{z \in Z : z \geq 0\}$ - положительные целые числа. Часто m - простое число (не слишком близкое к точной степени 2).

Метод	Хеш-функция
Метод деления	$h(x) = x \bmod m$
Метод умножения	$h(x) = \lfloor m (xA \bmod 1) \rfloor, A \in \{z \in R: 0 < z < 1\}$

Хеш-таблица

Хеш-функции, используемые в хеш-таблицах, используются для вычисления индекса в массиве слотов. Хеш-таблица - это структура данных для реализации словарей (структура ключ-значение). Хорошо реализованные хеш-таблицы имеют время $O(1)$ для следующих операций: вставка, поиск и удаление данных по ключу. Несколько ключей могут хешироваться в одном и том же слоте. Есть два способа разрешения коллизии:

1. Цепочка хешей: связанный список используется для хранения элементов с одинаковым значением хеша в слоте
2. Открытая адресация: в каждом слоте хранится ноль элементов или один элемент

Следующие методы используются для вычисления последовательностей, необходимых для открытой адресации

Метод	Формула
Линейное зондирование	$h(x, i) = (h'(x) + i) \bmod m$
Квадратичное зондирование	$h(x, i) = (h'(x) + c1 * i + c2 * i^2) \bmod m$
Двойное хэширование	$h(x, i) = (h1(x) + i * h2(x)) \bmod m$

Где $i \in \{0, 1, \dots, m - 1\}$, $h'(x), h1(x), h2(x)$ - вспомогательные хеш-функции, $c1, c2$ - положительные вспомогательные постоянные.

Примеры

Пусть $x \in U\{1, 1000\}$, $h = x \bmod m$. В следующей таблице показаны значения хэша в составном и простом случаях. Текст, выделенный жирным шрифтом, указывает на те же значения хеша.

x	$m = 100$ (составное)	$m = 101$ (простое)
723	23	16
103	3	2
738	38	31
292	92	90
61	61	61
87	87	87
995	95	86
549	49	44
991	91	82
757	57	50
920	20	11
626	26	20
557	57	52
831	31	23
619	19	13

Ссылки:

- Томас Х. Кормен, Чарльз Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. Введение в алгоритмы
- [Обзор хэш-таблиц](#)
- [Wolfram MathWorld - Хеш-функция](#)

Глава 44: Задача коммивояжера

Раздел 44.1: Алгоритм полного перебора

Путь, в котором мы проходим через каждую вершину ровно один раз, равносителен упорядочиванию вершин в некотором порядке. Следовательно, чтобы рассчитать минимальную стоимость за прохождение через каждую вершину ровно один раз, мы можем сделать полный перебор каждой из $N!$ перестановок чисел от 1 до N .

Псевдокод

```
minimum = INF # Присваиваем минимуму бесконечно большое значение
for all permutations P # Перебираем все перестановки

    current = 0 # Вводим счетчик для всех стоимостей путей

    for i from 0 to N-2
        current = current + cost[P[i]][P[i+1]] # Добавить стоимость пути
                                                # из вершины i в следующую

    current = current + cost[P[N-1]][P[0]] # Добавить стоимость пути из
                                            # последней вершины в первую
    if current < minimum # Обновить минимум, если необходимо
        minimum = current
output minimum # Выводим минимальную стоимость
```

Временная сложность

Есть $N!$ перестановок пути, которые нужно пройти. Цена за каждое прохождение вычисляется за $O(N)$, следовательно, этот алгоритм занимает $O(N * N!)$ времени для вывода точного ответа.

Раздел 44.2: Алгоритм динамического программирования

Обратите внимание, что если мы рассмотрим путь (по порядку):

(1,2,3,4,6,0,5,7)

и путь

(1,2,3,5,0,6,7,4)

то стоимость за переход из вершины 1 в вершину 2, а затем в вершину 3 остается одинаковой, так зачем же ее пересчитывать? Этот результат может быть сохранен для дальнейшего

использования.

Пусть dp [битовая маска] [вершина] представляет собой минимальные затраты на перемещения через все вершины, у которых соответствующий бит в битовой маске установлен на 1 окончание в вершине. Например:

```
dp[12][2]
12 = 1 1 0 0
    ^ // соответствие вершин и значения 1 в битовой маске
vertices: 3 2 1 0 // вершины
```

Поскольку 12 соответствует **1100** в двоичной системе счисления, значение $dp[12][2]$ - прохождение через вершины 2 и 3 в графе с путем, заканчивающимся в вершине 2.

Таким образом, мы получаем следующий алгоритм (реализация на языке C++):

```
int cost[N][N]; // Установить значение N, если необходимо
int memo[1 << N][N]; // Установить здесь все на -1
// Задача коммивояжера
int TSP(int bitmask, int pos){
    int cost = INF; // Присваиваем бесконечно большое значение стоимости
    if (bitmask == ((1 << N) - 1)){ // Если все вершины были посещены
        return cost[pos][0]; // Стоимость возвращения
    }
    if (memo[bitmask][pos] != -1){ // Если данное значение уже было вычислено
        return memo[bitmask][pos]; // Просто возвратить значение,
                                    // не нужно пересчитывать
    }
    for (int i = 0; i < N; ++i){ // Для каждой вершины
        if ((bitmask & (1 << i)) == 0){ // Если вершина не была посещена
            cost = min(cost,TSP(bitmask |
                (1 << i) , i) + cost[pos][i]); // Посетить вершину
        }
    }
    memo[bitmask][pos] = cost; // Сохранить результат
    return cost;
}
//Вызывать TSP(1,0)
```

Эта строка может быть немного запутанной, поэтому давайте пройдемся по ней детально:

```
cost = min(cost,TSP(bitmask | (1 << i) , i) + cost[pos][i]);
```

Здесь $bitmask | (1 \ll i)$ устанавливает i -й бит битовой маски на 1, что означает, что была проверена i -я вершина. i после запятой означает новую pos (позицию) в вызове функции, которая является новой «последней» вершиной. $cost[pos][i]$ нужна, чтобы добавлять к общим

затратам стоимость перемещения из последней вершины (`pos`) в вершину `i`.

Таким образом, эта строка нужна для нахождения минимально возможного значения за прохождение через все те вершины, которые еще не были проверены, и дальнейшего присвоения этого значения переменной `cost`.

Временная сложность

TSP(bitmask, pos) Функция `TSP(bitmask, pos)` имеет 2^N значений для битовой маски и N значений для `pos`. Каждая функция занимает $O(N)$ времени для выполнения (цикла `for`). Таким образом, эта реализация требует $O(N^2 * 2^N)$ времени, чтобы вывести точный ответ.

Глава 45: Задача о рюкзаке

Раздел 45.1: Основы задачи о рюкзаке

Задача: Дан набор предметов, в котором у каждого предмета есть два параметра: вес и ценность, определите сколько предметов нужно добавить в рюкзак, чтобы общий вес был меньше или равен заданному ограничению, а суммарная стоимость предметов была как можно большее.

Псевдокод к задаче о рюкзаке

Дано:

1. Стоимости предметов (массив v)
2. Веса (массив w)
3. Количество предметов (n)
4. Вместимость (W)

```
for j from 0 to W do: # Прямой ход
    m[0, j] := 0 # Все предметы не лежат в рюкзаке
for i from 1 to n do:
    for j from 0 to W do:
        if w[i] > j then: # Если вес предмета больше ограничения по весу
            m[i, j] := m[i-1, j] # Решение задачи с i предметом
            # сводится к решению задачи с i-1 предметом
        else:
            m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
            # Стоимости выборки присваивается максимальное значение
```

Простая реализация вышеуказанного псевдокода на Python:

```
# Задача о рюкзаке
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)] # Максимальная стоимость
    # предметов, которые можно уложить в рюкзак вместимости W, если можно
    # использовать только первые n предметов
    for i in range(n+1):
        for w in range(W+1):
            if i==0 or w==0:
                K[i][w] = 0 # Предмет не лежит в рюкзаке
            elif wt[i-1] <= w: # Вес i-1 предмета меньше вместимости рюкзака
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]) # Выбираем
                # предмет наибольшей стоимости
            else: # Не берем предмет с номером i
```

```

        K[i][w] = K[i-1][w]
    return K[n][W] # Выводим максимальную стоимость предметов, которые мы кладем
                # в рюкзак
val = [60, 100, 120] # Входные значения стоимостей
wt = [10, 20, 30] # Входные значения весов
W = 50 # Вместимость рюкзака
n = len(val) # Количество предметов
print(knapSack(W, wt, val, n)) # Вывод ответа к задаче

```

Запуск кода: сохраните это в файле с именем knapSack.py

```
$ python knapSack.py
220
```

Временная сложность приведенного выше кода: $O(nW)$, где n - количество предметов, а W - вместимость рюкзака.

Раздел 45.2: Реализация решения на C#

```

// Задача о рюкзаке
public class KnapsackProblem
{
    private static int Knapsack(int w, int[] weight, int[] value, int n)
    {
        int i;
        int[,] k = new int[n + 1, w + 1];
        for (i = 0; i <= n; i++)
        {
            int b;
            for (b = 0; b <= w; b++)
            {
                if (i == 0 || b == 0)
                {
                    k[i, b] = 0; // Все предметы не лежат в рюкзаке
                }
                else if (weight[i - 1] <= b) // Вес i-1 предмета меньше вместимости
                                            // рюкзака
                {
                    k[i, b] = Math.Max(value[i - 1] + k[i - 1, b - weight[i - 1]], 
                        k[i - 1, b]); // Выбираем предмет наибольшей стоимости
                }
                else // Не берем предмет с номером i
                {
                    k[i, b] = k[i - 1, b];
                }
            }
        }
    }
}
```

```
    return k[n, w]; // Выводим максимальную стоимость предметов,
                    // которые мы кладем в рюкзак
}
public static int Main(int nItems, int[] weights, int[] values)
{
    int n = values.Length; // Количество предметов
    return Knapsack(nItems, weights, values, n); // Выход ответа к задаче
}
}
```

Глава 46: Решение уравнений

Раздел 46.1: Линейные уравнения

Существует два класса методов решения линейных уравнений:

1. **Прямые методы:** Им свойственно преобразование исходного уравнения в эквивалентные уравнения, которые проще решаются, то есть мы получаем решение непосредственно из уравнения.
2. **Итерационный метод (Непрямой метод):** Начинается с предположения о решении уравнения, а затем повторно уточняется решение до тех пор, пока не будет достигнут определенный критерий сходимости. Итерационные методы обычно менее эффективны, чем прямые методы, поскольку требуется большое количество операций. Примером могут послужить итерационный метод Якоби или итерационный метод Гаусса-Зейделя.

Реализация на C-

```
// Реализация метода Якоби
void JacobisMethod(int n, double x[n], double b[n], double a[n][n]){
// n-размерность матрицы, x[n]-решение уравнения, b[n]-начальное приближение,
// a[n][n]-матрица коэффициентов
    double Nx[n]; // модифицированная форма переменных
    int rootFound=0; // флаг

    int i, j;
    while(!rootFound){
        for(i=0; i<n; i++){ // вычисление
            Nx[i]=b[i]; // присваиваем начальное приближение

            for(j=0; j<n; j++){
                if(i!=j) Nx[i] = Nx[i]-a[i][j]*x[j]; // приближение для
                // недиагональных эл-в матрицы
            }
            Nx[i] = Nx[i] / a[i][i]; // приближение для диагональных эл-в
        }

        rootFound=1; // проверка
        for(i=0; i<n; i++){
            if(!( (Nx[i]-x[i])/x[i] > -0.000001 && (Nx[i]-x[i])/x[i] < 0.000001 )){
                // уточнение погрешности
                rootFound=0;
                break;
            }
        }

        for(i=0; i<n; i++){ // оценка
```

```

        x[i]=Nx[i];
    }
}

return ;
}

// Реализация метода Гаусса-Зейделя
void GaussSeidalMethod(int n, double x[n], double b[n], double a[n][n]){
    double Nx[n]; // модифицированная форма переменных
    int rootFound=0; //флаг

    int i, j;
    for(i=0; i<n; i++){ // инициализация
        Nx[i]=x[i];
    }

    while(!rootFound){
        for(i=0; i<n; i++){
            Nx[i]=b[i];

            for(j=0; j<n; j++){
                if(i!=j) Nx[i]= Nx[i]-a[i][j]*Nx[j];
            }
            x[i]= Nx[i]/ a[i][i];
        }

        rootFound=1;
        for(i=0; i<n; i++){
            if(((Nx[i]-x[i])/x[i])>-0.000001&&(Nx[i]-x[i])/x[i]<0.000001)){
                rootFound=0;
                break;
            }
        }
    }

    for(i=0; i<n; i++){
        x[i]=Nx[i];
    }
}

return ;
}

void print(int n, double x[n]){
    int i;
    for(i=0; i<n; i++){
        rintf("%lf, ", x[i]);
    }
    printf("\n\n");
}

```

```

    return ;
}

int main(){

    int n=3;
    double x[n];
    double b[n],
           [n] [n];

    a[0][0]=8; a[0][1]=2; a[0][2]=-2; b[0]=8;      //8x_1+2x_2-2x_3+8=0
    a[1][0]=1; a[1][1]=-8; a[1][2]=3; b[1]=-4;     //x_1-8x_2+3x_3-4=0
    a[2][0]=2; a[2][1]=1; a[2][2]=9; b[2]=12;      //2x_1+x_2+9x_3+12=0

    int i;

    for(i=0; i<n; i++){
        x[i]=0;
    }
    cobisMethod(n, x, b, a);
    print(n, x);

    for(i=0; i<n; i++){
        x[i]=0;
    }
    GaussSeidalMethod(n, x, b, a);
    print(n, x);

    return 0;
}

```

Раздел 46.2: Нелинейное уравнение

Уравнение типа $f(x) = 0$ является алгебраическим или трансцендентным. Эти виды уравнений могут быть решены с использованием двух типов метода:

- Прямой метод:** этот метод дает точное значение всех корней непосредственно за конечное число шагов.
- Косвенный или итерационный метод:** итерационный метод лучше всего подходит для компьютерных программ для решения уравнения. Он основан на концепции последовательного приближения. В итеративном методе есть два способа решить уравнение:
 - Метод скобок:** мы берем две начальные точки такие, что корень лежит между ними. Например- биективный метод, метод ложного позиционирования.
 - Открытый метод:** мы берем одно или два начальных значения, что корень может быть где угодно. Например - метода Ньютона-Рафсона, метод последовательных приближений, метод секущих.

Реализация в С:

```

#define f(x) ( ((x)*(x)*(x)) - (x) - 2 )
#define f2(x) ( (3*(x)*(x)) - 1 )
#define g(x) ( cbrt( (x) + 2 ) )

double BisectionMethod(){
    double root=0;

    double a=1, b=2;
    double c=0;

    int loopCounter=0;
    if(f(a)*f(b)<0){
        while(1){
            loopCounter++;
            c=(a+b)/2;

            if(f(c)<0.00001 && f(c)>-0.00001){
                root=c;
                break;
            }

            if((f(a))*(f(c))<0){
                b=c;
            }else{
                a=c;
            }
        }
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}

double FalsePosition(){
    double root=0;

    double a=1, b=2;
    double c=0;

    int loopCounter=0;
    if(f(a)*f(b)<0){
        while(1){
            loopCounter++;

            c=(a*f(b)- b*f(a))/(f(b)- f(a));

            /*printf("%lf\t%lf \n", c, f(c));/**//test
            if(f(c)<0.00001 && f(c)>-0.00001){
                root=c;
                break;
            }
    }
}

```

```

        if((f(a))*(f(c))<0){
            b=c;
        }else{
            a=c;
        }
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}

double NewtonRaphson(){
    double root=0;

    double x1=1;
    double x2=0;

    int loopCounter=0;
    while(1){
        loopCounter++;

        x2 = x1 -(f(x1)/f2(x1));
        /*printf("%lf \t %lf \n", x2, f(x2));/**//test

        if(f(x2)<0.00001 && f(x2)>-0.00001){
            root=x2;
            break;
        }

        x1=x2;
    }
    printf("It took %d loops.\n", loopCounter);

    return root;
}

double FixedPoint(){
    double root=0;
    double x=1;

    int loopCounter=0;
    while(1){
        loopCounter++;

        if((x-g(x))<0.00001 && (x-g(x))>-0.00001){
            root = x;
            break;
        }

        /*printf("%lf \t %lf \n", g(x), x-(g(x)));/**//test
    }
}

```

```

x=g(x);

}

printf("It took %d loops.\n", loopCounter);

return root;
}

double Secant(){
double root=0;

double x0=1;
double x1=2;
double x2=0;

int loopCounter=0;
while(1){
    loopCounter++;

    /*printf("%lf \t %lf \t %lf \n", x0, x1, f(x1));/**//test

    if(f(x1)<0.00001 && f(x1)>-0.00001){
        root=x1;
        break;
    }

    x2 =((x0*f(x1))-(x1*f(x0)))/(f(x1)-f(x0));

    x0=x1;
    x1=x2;
}
printf("It took %d loops.\n", loopCounter);

return root;
}

int main(){
double root;

root = BisectionMethod();
printf("Using Bisection Method the root is: %lf \n\n", root);

root = FalsePosition();
printf("Using False Position Method the root is: %lf \n\n", root);

root = NewtonRaphson();
printf("Using Newton-Raphson Method the root is: %lf \n\n", root);

root = FixedPoint();
printf("Using Fixed Point Method the root is: %lf \n\n", root);

```

```
root = Secant();
printf("Using Secant Method the root is: %lf \n\n", root);
return 0;
}
```

Глава 47: Самая длинная общая подпоследовательность

Раздел 47.1: Объяснение самой длинной общей подпоследовательности

Одной из наиболее важных реализаций динамического программирования является поиск самой длинной общей подпоследовательности. Давайте сначала определим некоторые основные термины.

Подпоследовательности:

Подпоследовательность - это последовательность, которая может быть получена из другой последовательности путем удаления некоторых символов без изменения порядка оставшихся символов. Допустим, у нас есть строка **ABC**. Если мы сотрем ноль, один или более чем один символ из этой строки, мы получаем подпоследовательность этой строки. Таким образом, подпоследовательностями строки **ABC** будут {"**A**", "**B**", "**C**", "**AB**", "**AC**", "**BC**", "**ABC**", ""}. Даже если мы удалим все символы, пустая строка также будет подпоследовательностью. Чтобы обнаружить подпоследовательность, для каждого символа в строке у нас есть два варианта - либо мы берем символ, либо нет. Поэтому, если длина строки равна **n**, есть **2n** подпоследовательностей этой строки.

Самая длинная общая подпоследовательность:

Как следует из названия, из всех общих подпоследовательностей между двумя строками самая длинная общая подпоследовательность (LCS) - та, которая имеет максимальную длину. Например: общие подпоследовательности между "**HELLOM**" и "**HMLD**" – это "**H**", "**HL**", "**HM**" и т. д. Здесь "**HLL**" - самая длинная общая подпоследовательность, которая имеет длину **3**.

Метод полного перебора:

Мы можем сгенерировать все подпоследовательности двух строк, используя поиск с возвратом. Тогда мы можем сравнить их, чтобы узнать общие подпоследовательности. После мы должны найти ту, что имеет максимальную длину. Мы уже сталкивались с этим ранее, здесь **2n** подпоследовательности строки длины **n**. Потребовалось бы годы, чтобы решить проблему, если бы наши **n** пересекали **20-25**.

Метод динамического программирования:

Давайте изучим данный метод на примере. Предположим, что у нас есть две строки **abcdef** и **acbcf**. Обозначим их **s1** и **s2**. Таким образом, самая длинная общая подпоследовательность этих двух строк будет "**abcf**", которая имеет длину **4**. Напомню, что подпоследовательности не должны быть непрерывными в строке. Чтобы построить "**abcf**", мы проигнорировали "**da**" в **s1** и "**c**" в **s2**. Как мы узнаем это с помощью динамического программирования?

Мы начнем с таблицы (двумерный массив), содержащей все символы **s1** в строке и все символы **s2** в столбце. Здесь таблица имеет индекс **0**, и мы помещаем символы от **1** и далее. Мы пройдем по таблице слева направо для каждого ряда. Наша таблица будет выглядеть так:

		0	1	2	3	4	5	6	
	ch	a	b	c	d	a	f		
0									
1	a								
2	c								
3	b								
4	c								
5	f								

Здесь каждая строка и столбец представляют длину самой длинной общей подпоследовательности между двумя строками, если мы возьмем символы этой строки и столбца и добавим к префиксу перед ним. Например: **Таблица[2][3]** представляет длину самой длинной общей подпоследовательности между "ac" и "abc". 0-й столбец представляет пустую подпоследовательность **s1**. Аналогично, 0-я строка представляет пустую подпоследовательность **s2**. Если мы возьмем пустую подпоследовательность строки и попытаемся сопоставить ее с другой строкой, неважно, какова длина второй подстроки, общая подпоследовательность будет иметь длину 0. Таким образом, мы можем заполнить 0-е строки и 0-е столбцы с 0. Мы получили:

		0	1	2	3	4	5	6	
	ch	a	b	c	d	a	f		
0	0	0	0	0	0	0	0	0	
1	a	0							
2	c	0							
3	b	0							
4	c	0							
5	f	0							

Давайте начнем. Когда мы заполняем **таблицу[1][1]**, мы спрашиваем себя, если бы у нас была только строка a и другая строка a, что будет самой длинной общей подпоследовательностью? Длина LCS здесь будет 1. Теперь давайте посмотрим на **таблицу[1][2]**. У нас есть строка ab и строка a. Длина LCS будет равна 1. Как видите, остальные значения также будут равны 1 для первой строки, так как она рассматривает только строку a с abcd, abcda, abcdaf. Так будет выглядеть наша таблица:

		0	1	2	3	4	5	6	
0	ch	a	b	c	d	a	f		
1	a	0	1	1	1	1	1	1	
2	c	0							
3	b	0							
4	c	0							
5	f	0							

Для строки 2, которая теперь будет включать с. Для Таблицы [2] [1] у нас есть ас с одной стороны и а с другой стороны. Так что длина LCS - 1. Откуда мы взяли эту 1? Сверху, которая обозначает что LCS а между двумя подстроками.

И что мы говорим, что если $s1[2]$ и $s2[1]$ не совпадают, то длина LCS будет максимальной из длины LCS сверху или слева. Взятие длины LCS сверху означает, что мы не берем текущий символ из $s2$. Точно также, принимая длину LCS слева, мы не берем текущий символ из $s1$ для создания LCS. Мы получили:

		0	1	2	3	4	5	6	
0	ch	a	b	c	d	a	f		
1	a	0	1	1	1	1	1	1	
2	c	0	1						
3	b	0							
4	c	0							
5	f	0							

Итак, наша первая формула будет:

```
if s2[i] is not equal to s1[j]
    Table[i][j] = max(Table[i-1][j], Table[i][j-1])
endif
```

Двигаясь дальше, для Таблицы[2][2] мы имеем строку ab и ac. Поскольку c и b не совпадают,

мы ставим максимум вершины или оставляем здесь. В данном случае это снова **1**. После этого для **Таблицы[2][3]** у нас есть строки **abc** и **ac**. На этот раз текущие значения и строки и столбца одинаковы. Теперь длина LCS будет равна максимальной длине $LCS + 1$. Как мы можем получить максимальную длину LCS до сих пор? Мы проверяем диагональное значение, которое представляет лучшее соответствие между **ab** и **a**. Из этого состояния для текущих значений мы добавляем еще один символ к **s1** и **s2**, с которыми случилось то же самое. Так что длина LCS, конечно, увеличится. Мы положим $1 + 1 = 2$ в **таблице[2][3]**. Мы получили:

			0	1	2	3	4	5	6	
0	ch	a	b	c	d	a	f			
1	a	0	1	1	1	1	1	1	1	
2	c	0	1	1	2					
3	b	0								
4	c	0								
5	f	0								

Итак, наша вторая формула будет:

```
if s2[i] equals to s1[j]
    Table[i][j] = Table[i-1][j-1]+1
endif
```

Мы определили оба случая. Используя эти две формулы, мы можем заполнить всю таблицу. После заполнения таблица будет выглядеть так:

			0	1	2	3	4	5	6	
0	ch	a	b	c	d	a	f			
1	a	0	1	1	1	1	1	1	1	
2	c	0	1	1	2	2	2	2	2	
3	b	0	1	2	2	2	2	2	2	
4	c	0	1	2	3	3	3	3	3	
5	f	0	1	2	3	3	3	3	4	

Длина LCS между s_1 и s_2 будет Таблицей $[5][6] = 4$. Здесь 5 и 6 - длина s_2 и s_1 соответственно. Наш псевдокод будет выглядеть так:

```
Procedure LCSlength(s1, s2):
Table[0][0]=0
for i from 1 to s1.length
    Table[0][i]=0
endfor
for i from 1 to s2.length
    Table[i][0]=0
endfor
for i from 1 to s2.length
    for j from 1 to s1.length
        if s2[i] equals to s1[j]
            Table[i][j]= Table[i-1][j-1]+1
        else
            Table[i][j]= max(Table[i-1][j], Table[i][j-1])
        endif
    endfor
endfor
Return Table[s2.length][s1.length]
```

Временная сложность для этого алгоритма: $O(mn)$, где m и n обозначают длину каждой строки.

Как мы узнаем самую длинную общую подпоследовательность? Начнем с правого нижнего угла. Мы будем проверять, откуда приходит значение. Если значение приходит из диагонали, то есть если **Таблица[i-1][j-1]** равна **Таблице[i][j] - 1**, мы вставляем либо $s_2[i]$, либо $s_1[j]$ (оба одинаковые) и движемся по диагонали. Если значение приходит сверху, это означает, что если **Таблица[i-1][j]** равна **Таблице[i][j]**, мы переходим к вершине. Если значение идет слева, значит, что если **Таблица [i][j-1]** равна **Таблице[i][j]**, мы двигаемся влево. Когда мы достигаем крайнего левого или верхнего столбца, наш поиск заканчивается. Затем мы извлекаем значения из стека и печатаем их. Псевдокод:

```
Procedure PrintLCS(LCSlength, s1, s2)
temp := LCSlength
S = stack()
i := s2.length
j := s1.length
while i is not equal to 0 and j is not equal to 0
    if Table[i-1][j-1]== Table[i][j]-1 and s1[j]==s2[i]
        S.push(s1[j])
        i := i -1
        j := j -1
    elseif Table[i-1][j]== Table[i][j]
        i := i-1
    else
        j := j-1
    endif
endwhile
```

```
while S is not empty
    print(S.pop)
endwhile
```

Следует отметить: если **Таблица[i-1][j]** и **Таблица[i][j-1]** равны **Таблице[i][j]** и **Таблица[i-1][j-1]** не равна **Таблице [i][j] - 1**, то может быть две LCS. Этот псевдокод не учитывает такую ситуацию. Вы должны будете решить эту проблему рекурсивно, чтобы найти несколько LCS .

Временная сложность для этого алгоритма: **O (max (m, n))**.

Глава 48: Самая длинная возрастающая подпоследовательность

Раздел 48.1: Самая длинная возрастающая подпоследовательность.

Проблема самой длинной возрастающей подпоследовательности состоит в том, чтобы найти подпоследовательность из заданной входной последовательности, в которой элементы подпоследовательности сортируются в порядке убывания. Все подпоследовательности не являются смежными или уникальными.

Применение самой длинной возрастающей подпоследовательности:

Такие алгоритмы как: самая длинная возрастающая подпоследовательность, самая длинная общая подпоследовательность используются в управлении версиями таких систем как Git и т. д.

Простая форма алгоритма:

1. Найти уникальные строки, которые являются общими для обоих документов.
2. Взять все такие строки из первого документа и упорядочить их в соответствии с их появлением во втором документе.
3. Вычислить LIS полученной последовательности (выполнив Терпеливую сортировку), получив самую длинную подходящую последовательность строк - соответствие между строками двух документов.
4. Повторяйте алгоритм на каждом диапазоне линий между уже подобранными.

Теперь давайте рассмотрим более простой пример проблемы LCS. Здесь вход представляет собой только одну последовательность различных целых чисел a_1, a_2, \dots, a_n , и мы хотим найти самую длинную возрастающую подпоследовательность в ней. Например, если входным значением будут **7,3,8,4,2,6** тогда самая длинная возрастающая подпоследовательность составляет **3,4,6**.

Самый простой подход заключается в сортировке входных элементов в порядке возрастания и применении алгоритма LCS к исходной и отсортированной последовательности. Однако, если вы посмотрите на результирующий массив, вы заметите, что многие значения одинаковы, и массив выглядит повторяющимся. Это говорит о том, что проблема LIS (самая длинная возрастающая подпоследовательность) может быть решена с применением алгоритма динамического программирования, используя только одномерный массив.

Псевдокод:

1. Описать массив значений, которые мы хотим вычислить.

Для $1 \leq i \leq n$ пусть $A(i)$ будет длиной самой длинной увеличивающейся последовательности ввода. Обратите внимание, что длина, которая в +конечном итоге нас интересует - $\max \{A(i) \mid 1 \leq i \leq n\}$.

2. Задать рекуррентность.

Для $1 \leq i \leq n$, $A(i) = 1 + \max\{A(j) | 1 \leq j < i \text{ and } \text{input}(j) < \text{input}(i)\}$.

3. Вычислить значения A .

4. Найти оптимальное решение.

Следующая программа использует A для вычисления оптимального решения. Первая часть вычисляет значение m так, что $A(m)$ - длина оптимальной возрастающей подпоследовательности ввода. Вторая часть вычисляет оптимальное увеличение подпоследовательности, для удобства выводим её в обратном порядке. Эта программа выполняется за время $O(n)$, поэтому весь алгоритм работает за время $O(n^2)$.

Часть 1:

```
m <- 1
for i :2..n
    if A(i) > A(m) then
        m <- i
    end if
end for
```

Часть 2:

```
put a
while A(m) > 1 do
    i <- m - 1
    while not(ai < am and A(i) = A(m) - 1) do
        i <- i - 1
    end while
    m <- i
    put a
end while
```

Рекурсивное решение:

Подход 1:

```
LIS(A[1..n]):
    if(n = 0) then return 0
    m = LIS(A[1..(n - 1)])
    B is subsequence of A[1..(n - 1)] with only elements less than a[n]
    (* let h be size of B, h <= n-1*)
    m = max(m, 1+ LIS(B[1..h]))
    Output m
```

Временная сложность в подходе 1: $O(n^2)$

Подход 2:

```

LIS(A[1..n], x):
    if(n = 0) then return 0
    m = LIS(A[1..(n - 1)], x)
    if(A[n] < x) then
        m = max(m, 1+ LIS(A[1..(n - 1)], A[n]))
    Output m

MAIN(A[1..n]):
    return LIS(A[1..n], infinity)

```

Временная сложность в подходе 2: $O(n^2)$

Подход 3:

```

LIS(A[1..n]):
    if(n = 0) return 0
    m = 1
    for i = 1 to n-1 do
        if(A[i] < A[n]) then
            m = max(m, 1+ LIS(A[1..i]))
    return m

MAIN(A[1..n]):
    return LIS(A[1..i])

```

Временная сложность в подходе 3: $O(n^2)$

Итерационный алгоритм:

Вычисляет значения итеративно снизу вверх.

```

LIS(A[1..n]):
    Array L[1..n]
    (* L[i]= value of LIS ending(A[1..i])*)
    for i =1 to n do
        L[i]=1
        for j =1 to i-1 do
            if(A[j] < A[i]) do
                L[i]= max(L[i], 1+ L[j])
    return L

MAIN(A[1..n]):
    L = LIS(A[1..n])
    return the maximum value in L

```

Временная сложность в итерационном подходе: $O(n^2)$

Вспомогательное пространство: $O(n)$

Давайте возьмем $\{0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15\}$ в качестве входных данных. Итак, самая длинная возрастающая подпоследовательность для данного входа - $\{0, 2, 6, 9, 11, 15\}$.

Глава 49: Проверка двух строк на анаграмму

Две строки с одинаковым набором символов называются анаграммой. Здесь я использовал javascript.

Мы создадим хэш str1 и увеличим количество + 1. Мы сделаем цикл на 2-й строке и проверим, все ли символы есть в хэше, и уменьшим значения хэш-ключа. Проверьте, если все значения хэш-ключа равны нулю, тогда будет анаграмма.

Раздел 49.1: Ввод и вывод образцов

Пример 1:

```
let str1 ='stackoverflow';
let str2 ='flowerovstack';
```

Эти строки - анаграммы.

// Создайте хэш из str1 и увеличьте один счетчик.

```
hashMap ={  
    s :1,  
    t :1,  
    a :1,  
    c :1,  
    k :1,  
    o :2,  
    v :1,  
    e :1,  
    r :1,  
    f :1,  
    l :1,  
    w :1  
}
```

Вы можете видеть, что хэш-ключ 'o' содержит значение 2, потому что о встречается 2 раза в строке.

Теперь зациклите str2 и проверьте, чтобы каждый символ присутствовал в hashMap, если да, уменьшите значение ключа hashMap, иначе верните Ложь(false) (что указывает на то, что это не анаграмма).

```
hashMap ={  
    s :0,  
    t :0,
```

```

a :0,
c :0,
k :0,
o :0,
v :0,
e :0,
r :0,
f :0,
l :0,
w :0
}

```

Теперь зациклите объект hashMap и проверьте, что все значения равны нулю в ключе hashMap.

В нашем случае все значения равны нулю, так что это анаграмма.

Раздел 49.2: Общий код для анаграмм

```

(function(){

var hashMap ={};

function isAnagram (str1, str2){

    if(str1.length!== str2.length){
        return false;
    }

    value one (+1).
    createStr1HashMap(str1);

    var valueExist = createStr2HashMap(str2);

    return isStringsAnagram(valueExist);
}

function createStr1HashMap (str1){
    [].map.call(str1, function(value, index, array){
        hashMap[value]= value in hashMap ?(hashMap[value]+1):1;
        return value;
    });
}

function createStr2HashMap (str2){
    var valueExist =[].every.call(str2, function(value, index, array){
        if(value in hashMap){
            hashMap[value]= hashMap[value]-1;
        }
        return value in hashMap;
    });
}
}

```

```
});

return valueExist;
}

function isStringsAnagram (valueExist){
    if(!valueExist){
        return valueExist;
    }else{
        var isAnagram;
        for(var i in hashMap){
            if(hashMap[i] !==0){
                isAnagram =false;
                break;
            }else{
                isAnagram =true;
            }
        }
        return isAnagram;
    }
}

isAnagram("stackoverflow", "flowerovstack");
isAnagram("stackoverflow", "flowervvstack");
```

Временная сложность: $3n$, т.е. $O(n)$.

Глава 50 : Треугольник Паскаля

Раздел 50.1: Треугольник Паскаля на С

```
int i, space, rows, k=0, count = 0, count1 = 0; /*rows - кол-во строк,
//space - кол-во отступов, count - кол-во чисел до "середины строки",
//k (count1) - число, на которое числа увеличиваются(уменьшаются)*/
rows=5;

//вывод треугольника Паскаля
for(i=1; i<=rows; ++i)//подсчет выведенных строк
{
    for(space=1; space <= rows-i; ++space)/*создание отступов для выравнивания
    //треугольника. количество отступов соответствует (число строк - номер
    //строки)*/
    {
        printf("  ");
        ++count;
    }
    while(k != 2*i-1)//количество чисел в строке
    {
        if (count <= rows-1)/*вывод первой половины строки. первое число -
        //номер строки,
        //последующие - увеличиваются на единицу.
        //максимальное число - середина строки - с номером rows по счету*/
        {
            printf("%d ", i+k);
            ++count;
        }
        else/*вывод второй половины строки. каждое последующее число уменьшается
        //на единицу*/
        {
            ++count1;
            printf("%d ", (i+k-2*count1));
        }
        ++k;
    }
    count1 = count = k = 0;
    printf("\n");//переход к новой строке
}
```

Выход

```
1
2 3 2
3 4 5 4 3
4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
```

Глава 51: Алгоритм:- Вывод матрицы m*n по спирали

Примеры ввода и вывода приведены ниже.

Раздел 51.1: Пример

Input:

```
14 15 16 17 18 21  
19 10 20 11 54 36  
64 55 44 23 80 39  
91 92 93 94 95 42
```

Output:

```
print value in index  
14 15 16 17 18 21 36 39 42 95 94 93 92 91 64 19 10 20 11 54 80 23 44 55
```

or print index

```
00 01 02 03 04 05 15 25 35 34 33 32 31 30 20 10 11 12 13 14 24 23 22 21
```

Раздел 51.2: Напишем общий код

```
#число "уровней" спирали  
function noOfLooping(m,n) {  
    # ищем минимальную сторону  
    if(m > n) {  
        smallestValue = n;  
    } else {  
        smallestValue = m;  
    }  
    # делим ее пополам с округлением вверх  
    if(smallestValue % 2 == 0) {  
        return smallestValue/2;  
    } else {  
        return (smallestValue+1)/2;  
    }  
}  
#вывод матрицы по спирали  
function squarePrint(m,n) {  
    var looping = noOfLooping(m,n); #количество повторений  
  
    #вывод одного "уровня" спирали  
    for(var i = 0; i < looping; i++) {  
        for(var j = i; j < m - 1 - i; j++) { #верхняя горизонталь  
            console.log(i+' '+j); #вывести индексы i, j  
        }  
        for(var k = i; k < n - 1 - i; k++) { #правая вертикаль  
            console.log(k+' '+j); #вывести индексы k, j  
        }  
    }  
}
```

```
        console.log(k+' '+j);
    }
    for(var l = j; l > i; l--) {#нижняя горизонталь
        console.log(k+' '+l);
    }
    for(var x = k; x > i; x--) {#левая вертикаль
        console.log(x+' '+l);
    }
}
}

squarePrint(6,4); #применить к матрице размером 4x6
```

Глава 52: Возвведение матрицы в степень

Раздел 52.1: Возвведение матрицы в степень для решения типовых задач

Найти $f(n)$: n -е число Фибоначчи. Задача довольно проста, когда n относительно маленькое. Мы можем использовать простую рекурсию, $f(n) = f(n-1) + f(n-2)$, или динамическое программирование, чтобы избежать вычисления одной и той же функции снова и снова. Но что вы будете делать, если в задаче сказано: **при $0 < n < 10^9$ найти $f(n) \bmod 999983$?** Динамическое программирование не подойдет, так как же мы решим эту задачу?

Сначала давайте посмотрим, как возведение матрицы в степень может помочь представить рекурсивное отношение.

Необходимые умения:

- Имея две матрицы, необходимо знать, как найти их произведение. Далее, имея матрицу, полученную путем перемножения этих двух матриц, и одну из этих матриц, необходимо знать, как найти другую матрицу.
- Имея матрицу размера $d \times d$, необходимо знать, как найти ее n -ю степень за $O(d^3 \log(n))$.

Шаблоны:

Сначала нам нужно рекурсивное отношение, а после мы должны найти матрицу M , которая может привести нас к желаемому состоянию из множества уже известных состояний. Предположим, что мы знаем k состояний данного рекуррентного отношения и хотим найти состояние $(k+1)$. Пусть M - матрица размера $k \times k$, и из известных состояний рекуррентного отношения мы строим матрицу $A: [k \times 1]$, после мы строим матрицу $B: [k \times 1]$, которая будет представлять множество следующих состояний, т. е. $M \times A = B$, как показано ниже:

$$M \times \begin{vmatrix} f(n) \\ f(n-1) \\ f(n-2) \\ \dots \\ f(n-k) \end{vmatrix} = \begin{vmatrix} f(n+1) \\ f(n) \\ f(n-1) \\ \dots \\ f(n-k+1) \end{vmatrix}$$

Так что если мы сможем построить M в соответствии с вышесказанным, то наша задача будет выполнена! Затем матрица будет использоваться для представления рекуррентного отношения.

Тип 1:

Начнем с самого простого, $f(n) = f(n-1) + f(n-2)$.

Мы получаем, $f(n+1) = f(n) + f(n-1)$.

Предположим, что мы знаем $f(n)$ и $f(n-1)$; Мы хотим найти $f(n+1)$

Из ситуации, описанной выше, матрица A и матрица B могут быть построены так, как показано ниже:

$$\begin{array}{c|c} \text{Matrix A} & \text{Matrix B} \\ \hline f(n) & f(n+1) \\ f(n-1) & f(n) \end{array}$$

[Примечание: Матрица **A** всегда будет построена таким образом, что каждое состояние, от которого зависит $f(n+1)$, будет присутствовать] Теперь нам нужно построить матрицу **M** размера **2 × 2** таким образом, чтобы она удовлетворяла $\mathbf{M} \times \mathbf{A} = \mathbf{B}$, как указано выше.

Первый элемент **B** - это $f(n+1)$, что на самом деле является $f(n) + f(n-1)$. Чтобы получить это из матрицы **A**, нам нужно $1 \times f(n)$ и $1 \times f(n-1)$. Таким образом, первая строка **M** будет равна **[1 1]**.

$$\begin{vmatrix} 1 & 1 \\ \cdots & \end{vmatrix} \times \begin{vmatrix} f(n) \\ f(n-1) \end{vmatrix} = \begin{vmatrix} f(n+1) \\ \cdots \end{vmatrix}$$

[Примечание: \cdots означает, что нам не важно это значение.]

Аналогично, 2-й элемент **B** это $f(n)$, который можно получить просто взяв $1 \times f(n)$ из **A**, поэтому 2-я строка **M** равна **[1 0]**.

$$\begin{vmatrix} \cdots & \end{vmatrix} \times \begin{vmatrix} f(n) \\ f(n-1) \end{vmatrix} = \begin{vmatrix} \cdots \\ f(n) \end{vmatrix}$$

Тогда мы получим нашу желаемую матрицу **M** размера **2 × 2**.

$$\begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix} \times \begin{vmatrix} f(n) \\ f(n-1) \end{vmatrix} = \begin{vmatrix} f(n+1) \\ f(n) \end{vmatrix}$$

Эти матрицы просто выводятся с помощью матричного умножения.

Тип 2:

Давайте немного усложним задачу: найдем $f(n) = a \times f(n-1) + b \times f(n-2)$, где **a** и **b** - константы. Из этого следует, что $f(n+1) = a \times f(n) + b \times f(n-1)$.

К этому моменту должно быть ясно, что размерность матриц будет равна числу зависимостей, т. е. в данном примере снова 2. Таким образом, для **A** и **B** мы можем построить две матрицы размера **2 × 1**:

Matrix A	Matrix B
$f(n)$	$f(n+1)$
$f(n-1)$	$f(n)$

Теперь для $f(n+1) = a \times f(n) + b \times f(n-1)$ нам нужно иметь $[a, b]$ в первой строке искомой матрицы **M**. Что касается 2-го элемента в **B**, т. е. $f(n)$, мы уже знаем его из матрицы **A**, поэтому мы просто возьмем значение оттуда, из чего следует, что 2-я строка матрицы **M** равна **[1 0]**. На этот раз мы получим:

$$\begin{vmatrix} a & b \\ 1 & 0 \end{vmatrix} \times \begin{vmatrix} f(n) \\ f(n-1) \end{vmatrix} = \begin{vmatrix} f(n+1) \\ f(n) \end{vmatrix}$$

Довольно просто, да?

Тип 3:

Если вы дошли до этого этапа, вы явно повзрослели и готовы столкнуться с немногим сложным соотношением: найти $f(n) = a \times f(n-1) + c \times f(n-3)$?

Упс! Несколько минут назад все, что мы видели, имело непрерывные состояния, но здесь состояние $f(n-2)$ отсутствует. Что теперь?

На самом деле это уже не проблема, мы можем преобразовать соотношение следующим образом: $f(n) = a \times f(n-1) + 0 \times f(n-2) + c \times f(n-3)$, получая $f(n+1) = a \times f(n) + 0 \times f(n-1) + c \times f(n-2)$. Теперь мы видим, что это на самом деле форма, описанная в типе 2. Таким образом, здесь искомая матрица M будет иметь размер **3 X 3**, а ее элементы равны:

$$\begin{vmatrix} a & 0 & c \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix} \times \begin{vmatrix} f(n) \\ f(n-1) \\ f(n-2) \end{vmatrix} = \begin{vmatrix} f(n+1) \\ f(n) \\ f(n-1) \end{vmatrix}$$

Они рассчитываются так же, как и в типе 2. Если для вас это сложно, попробуйте вычислить их на бумаге.

Тип 4:

Жизнь становится адски сложной, и госпожа Проблема теперь просит вас найти $f(n) = f(n-1) + f(n-2) + c$, где **c** - любая константа.

На этот раз это что-то новое, ведь в прошлом мы видели лишь то, что после умножения каждое состояние в **A** переходит в свое следующее состояние в **B**.

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) + c \\ f(n+1) &= f(n) + f(n-1) + c \\ f(n+2) &= f(n+1) + f(n) + c \\ &\dots \text{ so on} \end{aligned}$$

Итак, мы не сможем продолжить в той же манере, что и раньше, но что будет, если добавить **c** в качестве состояния:

$$M \times \begin{vmatrix} f(n) \\ f(n-1) \\ c \end{vmatrix} = \begin{vmatrix} f(n+1) \\ f(n) \\ c \end{vmatrix}$$

Теперь не так уж трудно составить **M**. Вот как это делается, но не забудьте проверить самостоятельно:

$$\begin{vmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} f(n) \\ f(n-1) \\ c \end{vmatrix} = \begin{vmatrix} f(n+1) \\ f(n) \\ c \end{vmatrix}$$

Тип 5:

Давайте соберем все это: требуется найти $f(n) = a \times f(n-1) + c \times f(n-3) + d \times f(n-4) + e$. Давайте оставим это в качестве упражнения для вас. Сначала попробуйте найти состояния и матрицу M . И проверить, соответствует ли она вашему решению. Также найдите матрицы A и B .

$$\begin{vmatrix} a & 0 & c & d & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{vmatrix}$$

Тип 6:

Иногда рекурсия дана в таком виде:

$$\begin{aligned} f(n) &= f(n-1) & -> \text{if } n \text{ is odd} \\ f(n) &= f(n-2) & -> \text{if } n \text{ is even} \end{aligned}$$

В сокращенной форме:

$$f(n) = (n \& 1) \times f(n-1) + (!n \& 1) \times f(n-2)$$

Здесь мы можем разделить функции на основе четности (четные и нечетные) и хранить 2 разные матрицы для обеих из них, а затем вычислить их отдельно.

Тип 7:

Чувствуете себя слишком уверенно? Повезло вам. Иногда нам может потребоваться обработать более одной рекурсии там, где это необходимо. Например, пусть рекурсия задана следующим образом:

$$g(n) = 2g(n-1) + 2g(n-2) + f(n)$$

Здесь рекурсия $g(n)$ зависит от $f(n)$, и может быть вычислена в той же матрице, но с увеличенными размерами. Давайте сначала построим из этого выражения матрицы A и B .

Matrix A	Matrix B
$g(n)$	$g(n+1)$
$g(n-1)$	$g(n)$
$f(n+1)$	$f(n+2)$
$f(n)$	$f(n+1)$

Здесь $g(n+1) = 2g(n-1) + f(n+1)$ и $f(n+2) = 2f(n+1) + 2f(n)$. Теперь, используя процессы, описанные выше, мы можем найти искомую матрицу M , которая будет выглядеть следующим образом:

2	2	1	0
1	0	0	0
0	0	2	2
0	0	1	0

Итак, это основные категории рекуррентных отношений, которые для решения используют эту простую методику.

Глава 53: Алгоритм минимального вершинного покрытия

Переменная	Значение
G	Исходный связный неориентированный граф
X	Множество вершин
C	Окончательное множество вершин

Это полиномиальный алгоритм для нахождения минимального вершинного покрытия связного неориентированного графа. Временная сложность этого алгоритма равна $O(n^2)$

Псевдокод алгоритма

Алгоритм PMinVertexCover (граф G)

Входной подключенный граф G

Выходная минимальная вершина C

```
Set C <- new Set<Vertex>()

Set X <- new Set<Vertex>()

X <- G.getAllVerticiesArrangedDescendinglyByDegree()

for v in X do
    List<Vertex> adjacentVertices1 <- G.getAdjacent(v)

    if !C contains any of adjacentVertices1 then
        C.add(v)

for vertex in C do

    List<vertex> adjacentVertices2 <- G.adjacentVertecies(vertex)

    if C contains any of adjacentVertices2 then
        C.remove(vertex)

return C
```

C – это минимальное вершинное покрытие графа G

мы можем использовать блочную сортировку вершин по их степени, потому что максимальное значение степеней равно $(n-1)$, где n – число вершин. Временная сложность алгоритма сортировки будет равна $O(n)$

Глава 54: Динамическая трансформация временной шкалы (DTW)

Раздел 54.1: Введение в динамическую трансформацию временной шкалы (DTW)

Алгоритм динамической трансформации временной шкалы (DTW) - это алгоритм измерения сходства между двумя временными последовательностями, которые могут различаться по скорости. Например, сходство в ходьбе может быть обнаружено с помощью DTW, даже если один человек шел быстрее другого, или если человек ускорялся и замедлялся. Алгоритм может использоваться для сопоставления образца голосовой команды с другими командами, даже если человек произнес ее быстрее или медленнее заранее записанного образца. DTW может быть применен к временным последовательностям видео-, аудио- и графических данных. На самом деле любые данные, которые можно преобразовать в линейную последовательность, могут быть обработаны с помощью DTW.

В целом, DTW – это метод, который вычисляет оптимальное соответствие между двумя заданными последовательностями с определенными ограничениями. Но давайте остановимся на более простых моментах. Скажем, у нас есть две голосовые последовательности: **Sample** и **Test**, и мы хотим проверить, совпадают ли эти последовательности или нет. Здесь голосовая последовательность строится по преобразованному цифровому сигналу вашего голоса. Это может быть амплитуда или частота вашего голоса, обозначающие слова, которые вы произнесли. Давайте предположим:

```
Sample = {1, 2, 3, 5, 5, 5, 6}  
Test = {1, 1, 2, 2, 3, 5}
```

Мы хотим найти оптимальное соответствие между этими двумя последовательностями. Сначала мы определим расстояние между двумя точками $d(x, y)$, где **x** и **y** - точки. Пусть

```
d(x, y) = |x - y| // абсолютная разница
```

Давайте создадим двумерную матрицу **Table**, используя эти две последовательности. Мы рассчитаем расстояния между каждой точкой из **Sample** с каждой точкой из **Test** и найдем оптимальное соответствие между ними.

	0	1	1	2	2	3	5
0							
1							
2							
3							
5							
5							
5							
6							

Здесь в **Table[i][j]** представлено минимальное расстояние между двумя последовательностями, если мы рассматриваем последовательность до **Sample[i]** и **Text[j]**, учитывая все минимальные расстояния, найденные ранее.

В первой строке мы не берем никакие значения из **Sample**, значит расстояние между **Sample** и **Test** будет бесконечным. Поэтому мы поместим бесконечность(*inf*) в первую строку. Аналогично и для первого столбца. Если мы не возьмем никакие значения из **Test**, расстояние между **Test** и **Sample** будет также бесконечным. А расстояние между **0** и **0** будет равно **0**. Получим

	0	1	1	2	2	3	5
0	0	inf	inf	inf	inf	inf	inf
1	inf						
2	inf						
3	inf						
5	inf						
5	inf						
5	inf						
6	inf						

Теперь для каждого шага мы рассмотрим расстояние между двумя точками и добавим его к минимальному найденному до этого расстоянию. Это даст нам минимальное расстояние между двумя последовательностями до определенной позиции. Итак, наша формула:

```
Table[i][j] := d(i, j) + min(Table[i-1][j], Table[i-1][j-1], Table[i][j-1])
```

Для первого шага $d(1, 1) = 0$, в **Table[0][0]** представлен минимум. Поэтому значение **Table[1][1]** будет $0 + 0 = 0$. Для второго шага $d(1, 2) = 0$. В **Table[1][1]** представлен минимум. Значение **Table[1][2]** будет $0 + 0 = 0$. Продолжая заполнение таблицы таким же образом, мы получим следующий результат:

	0	1	1	2	2	3	5
0	0	inf	inf	inf	inf	inf	inf
1	inf	0	0	1	2	4	8
2	inf	1	1	0	0	1	4
3	inf	3	3	1	1	0	2
5	inf	7	7	4	4	2	0
5	inf	11	11	7	7	4	0
5	inf	15	15	10	10	6	0
6	inf	20	20	14	14	9	1

В **Table[7][6]** представлено максимальное расстояние между двумя заданными последовательностями. Здесь **1** означает, что максимальное расстояние между **Sample** и **Test** равно **1**.

Теперь, если мы вернемся назад от последней точки до начальной точки **(0, 0)**, мы получим длинную линию, которая проходит по горизонтали, вертикали и диагонали. Процедура возврата будет такой:

```
if Table[i-1][j-1] <= Table[i-1][j] and Table[i-1][j-1] <= Table[i][j-1]
GoalKicker.com - Algorithms Notes for Professionals 240
    i := i - 1
    j := j - 1
else if Table[i-1][j] <= Table[i-1][j-1] and Table[i-1][j] <= Table[i][j-1]
    i := i - 1
else
    j := j - 1
end if
```

Мы продолжим до тех пор, пока не дойдем до **(0, 0)**. Каждый переход имеет свое значение:

- Горизонтальное движение - это удаление. Это означает, что последовательность **Test** ускорилась на этом интервале.
- Вертикальное движение – это вставка. Это означает, что последовательность **Test** замедлилась на этом интервале.
- Диагональное движение – это совпадение. На этом интервале **Test** и **Sample** были одинаковыми.

	0	1	1	2	2	3	5
0	0	inf	inf	inf	inf	inf	inf
1	inf	0	0	1	2	4	8
2	inf	1	1	0	0	1	4
3	inf	3	3	1	1	0	2
5	inf	7	7	4	4	2	0
5	inf	11	11	7	7	4	0
5	inf	15	15	10	10	6	0
6	inf	20	20	14	14	9	0

Псевдокод:

```

Procedure DTW(Sample, Test):
n := Sample.length
m := Test.length
Create Table[n + 1][m + 1]
for i from 1 to n
    Table[i][0] := infinity
end for
for i from 1 to m
    Table[0][i] := infinity
end for
Table[0][0] := 0
for i from 1 to n
    for j from 1 to m
        Table[i][j] := d(Sample[i], Test[j])
        + minimum(Table[i-1][j-1],
                  Table[i][j-1],
                  Table[i-1][j])
    end for
end for
Return Table[n + 1][m + 1]

```

Мы также можем добавить ограничение на место нахождения. То есть, мы требуем, что если $\text{Sample}[i]$ совпадает с $\text{Test}[j]$, то $|i - j|$ не больше, чем некоторый параметр w .

Сложность:

Сложность вычисления динамической трансформации временной шкалы равна $O(m * n)$, где m и n – длины последовательностей. Более быстрые техники вычисления DTW – это PrunedDTW, SparceDTW и FastDTW.

Применение:

- Распознавание речи

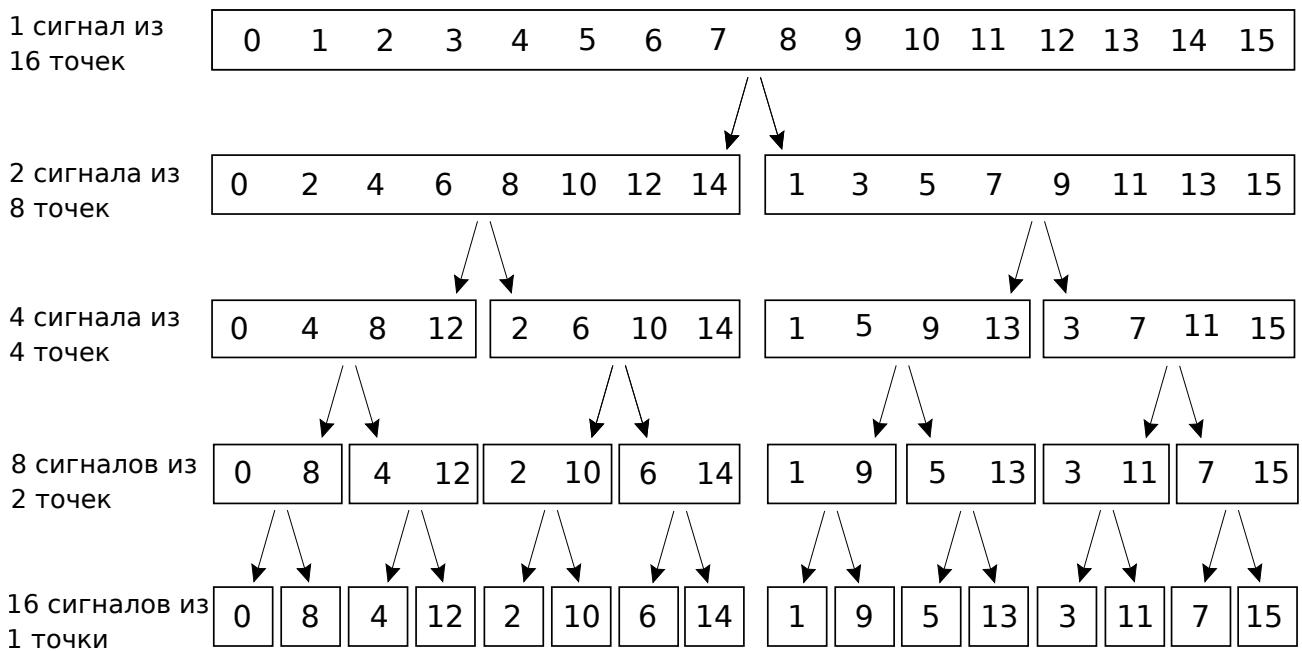
- Корреляционная атака по энергопотреблению

Глава 55: Быстрое преобразование Фурье

Действительная и комплексная форма ДПФ (Дискретное преобразование Фурье, DFT) может быть использована для частотного анализа или синтеза любых дискретных и периодических сигналов. БПФ (Быстрое преобразование Фурье, FFT) – это реализация ДПФ, которая быстро выполняется на современных процессорах.

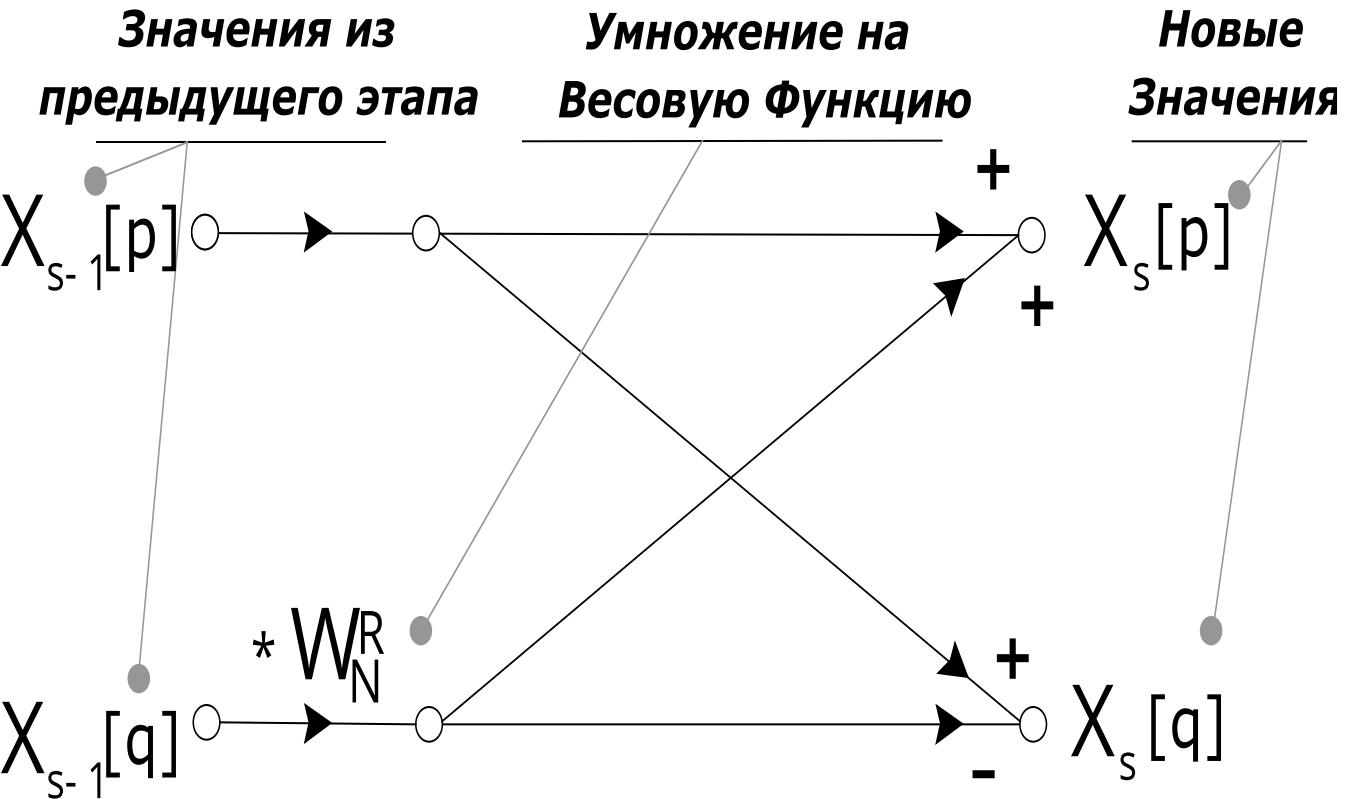
Раздел 55.1: Алгоритм Radix-2 для БПФ

Самым простым и, возможно, самым известным методом вычисления БПФ является алгоритм Radix-2 с прореживанием по времени. Radix-2 работает путем разложения сигнала N-точечной временной области на N сигналов временной области, каждый из которых состоит из одной точки.



Декомпозиция сигнала, или «прореживание по времени», достигается путем реверсирования битов индексов для массива данных временной области. Таким образом, для шестнадцати-точечного сигнала значение 1 (в двоичной системе 0001) заменяется на 8 (1000), значение 2 (0010) заменяется на 4 (0100) и так далее. Перестановка чисел с использованием метода реверса битов может быть легко выполнена на программном уровне, но ограничивает использование алгоритма Radix-2 для БПФ сигналами длины $N = 2^M$.

Значение одноточечного сигнала во временной области равно его значению в частотной области, поэтому массив разложенных одиночных точек временной области является массивом точек частотной области. Однако N единичных точек необходимо преобразовать в один N-точечный частотный спектр. Оптимальное преобразование полного частотного спектра осуществляется с использованием операции «бабочка». На каждом этапе преобразования алгоритм Radix-2 выполняет некоторое число двухточечных «бабочек», используя похожий набор экспоненциальных весовых функций Wn^R .



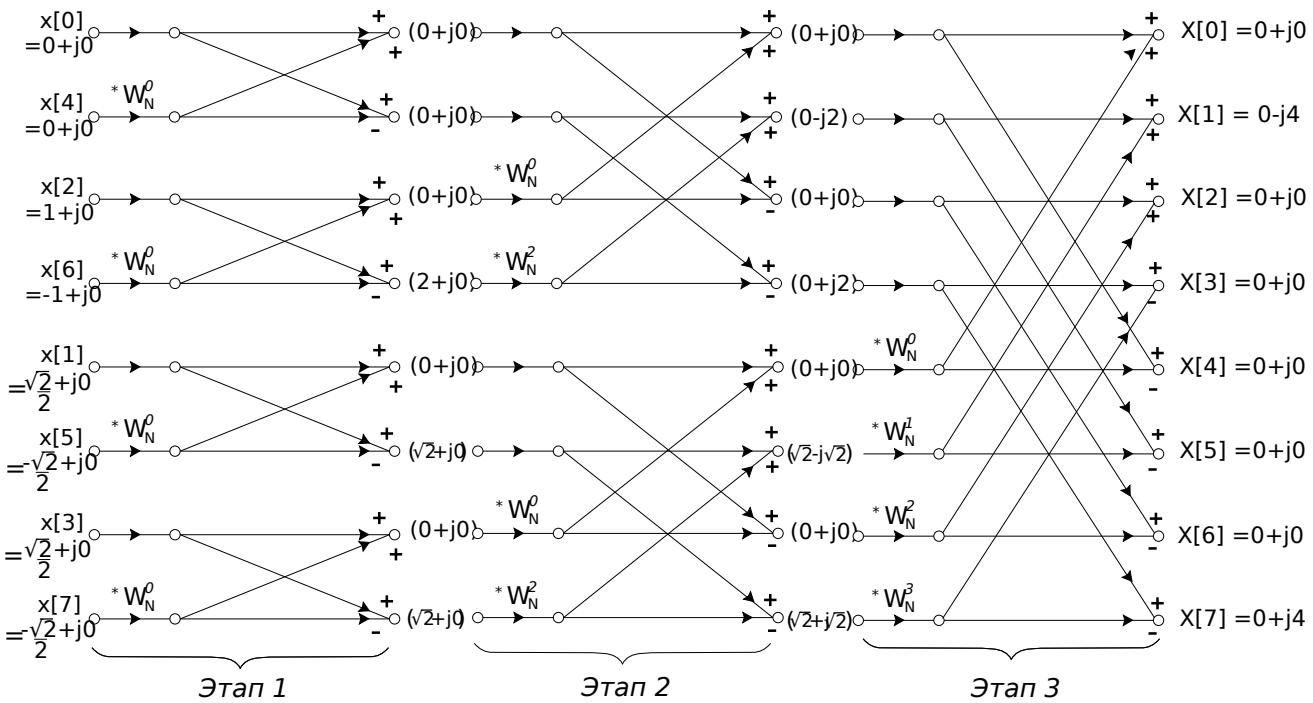
БПФ удаляет избыточные вычисления дискретного преобразования Фурье, используя периодичность W_n^R . Спектральное преобразование выполняется за $\log_2(N)$ шагов, на каждом из которых высчитывается $X[K]$ – результат операции «бабочка»; действительные и мнимые данные частотной области в прямоугольной форме. Для преобразования в магнитуду и фазу (полярные координаты) требуется найти абсолютное значение $\sqrt{(Re^2 + Im^2)}$ и аргумент $\tan^{-1}(Im/Re)$.

Экспоненциальный весовой коэффициент: $W_N^R = e^{-j(2\pi R/N)} = \cos(2\pi R/N) - j\sin(2\pi R/N)$

N: Количество точек в БПФ

R: Текущий коэффициент WN: зависит от N, текущего шага БПФ и разделения бабочек на этом шаге

Ниже показана полная структурная диаграмма «бабочек» алгоритма Radix 2 для восьми точечного БПФ. Обратите внимание, что входные сигналы были переупорядочены в соответствии с процедурой прореживания по времени, описанной выше.



БПФ обычно работает с комплексными входными данными и выдает комплексный результат. Для вещественных сигналов мнимая часть может быть установлена на 0, а действительная часть – на входной сигнал $x[n]$, однако можно оптимизировать алгоритм, используя преобразования только действительных данных. Значения Wn^R , используемые в процессе преобразования, могут быть определены с помощью экспоненциально взвешенного уравнения.

Величина R (экспоненциально взвешенная степень) определяется текущим этапом в спектральном преобразовании и результатом «бабочки».

Образец кода (C/C++)

Пример кода на C/C++ для вычисления БПФ при помощи алгоритма Radix-2 можно найти ниже. Это простая реализация, которая работает для любого N, где N – степень двойки. Этот алгоритм работает примерно в 3 раза медленнее, чем самая быстрая реализация БПФ, но все же это хорошая основа для будущей оптимизации и изучения алгоритма.

```
#include <math.h>

#define PI 3.1415926535897932384626433832795
#define TWOPI 6.283185307179586476925286766559
#define Deg2Rad 0.017453292519943295769236907684886
#define Rad2Deg 57.295779513082320876798154814105
#define log10_2 0.30102999566398119521373889472449 // Log10 of 2
#define log10_2_INV 3.3219280948873623478703194294948 // 1/Log10(2)

struct complex
{
public:
    double Re, Im;
};

bool isPwrTwo(int N, int *M)
{
```

```

*M = (int)ceil(log10((double)N) * log10_2_INV);
int NN = (int)pow(2.0, *M);
if ((NN != N) || (NN == 0))
    return false;
return true;
}

void rad2FFT(int N, complex *x, complex *DFT)
{
    int M = 0;
    if (!isPwrTwo(N, &M))
        throw "Rad2FFT(): N must be a power of 2 for Radix FFT";

    int BSep;
    int BWidth;
    int P;
    int j;
    int stage = 1;
    int HiIndex;
    unsigned int iaddr;
    int ii;
    int MM1 = M - 1;

    unsigned int i;
    int l;
    unsigned int nMax = (unsigned int)N;

    double TwoPi_N = TWOPi / (double)N;
    double TwoPi_NP;

    complex WN;
    complex TEMP;
    complex *pDFT = DFT;
    complex *pLo;
    complex *pHi;
    complex *pX;

    for (i = 0; i < nMax; i++, DFT++)
    {
        pX = x + i;
        ii = 0;
        iaddr = i;
        for (l = 0; l < M; l++)
        {
            if (iaddr & 0x01)
                ii += (1 << (MM1 - l));
            iaddr >>= 1;
            if (!iaddr)
                break;
        }
        DFT = pDFT + ii;
    }
}

```

```

DFT->Re = pX->Re;
DFT->Im = pX->Im;
}
for (stage = 1; stage <= M; stage++)
{
    BSep = (int)(pow(2, stage));
    P = N / BSep;
    BWidth = BSep / 2;
    TwoPi_NP = TwoPi_N*P;

    for (j = 0; j < BWidth; j++)
    {
        if (j != 0)
        {
            WN.Re = cos(TwoPi_N*P*j);
            WN.Im = -sin(TwoPi_N*P*j);
        }
        for (HiIndex = j; HiIndex < N; HiIndex += BSep)
        {
            pHi = pDFT + HiIndex;
            pLo = pHi + BWidth;

            if (j != 0)
            {
                TEMP.Re = (pLo->Re * WN.Re) - (pLo->Im * WN.Im);
                TEMP.Im = (pLo->Re * WN.Im) + (pLo->Im * WN.Re);

                pLo->Re = pHi->Re - TEMP.Re;
                pLo->Im = pHi->Im - TEMP.Im;

                pHi->Re = (pHi->Re + TEMP.Re);
                pHi->Im = (pHi->Im + TEMP.Im);
            }
            else
            {
                TEMP.Re = pLo->Re;
                TEMP.Im = pLo->Im;

                pLo->Re = pHi->Re - TEMP.Re;
                pLo->Im = pHi->Im - TEMP.Im;

                pHi->Re = (pHi->Re + TEMP.Re);
                pHi->Im = (pHi->Im + TEMP.Im);
            }
        }
    }
    pLo = 0;
    pHi = 0;
    pDFT = 0;
    DFT = 0;
}

```

```
    pX = 0;  
}
```

Раздел 55.2: Обратное быстрое преобразование Фурье с двумя корнями

Из-за сильной двойственности преобразования Фурье, корректировка выходных данных прямого преобразования может привести к обратному FFT. Данные в области частот могут быть преобразованы в область времени с помощью следующего метода:

1. Найти комплексное сопряжение данных области частот путем инвертирования воображаемой компоненты для всех экземпляров K.
2. Выполнить прямой FFT на сопряженных данных области частот.
3. Разделить каждый выходной результат этого FFT на N, чтобы получить истинное значение временной области.
4. Найти комплексное сопряжение выходных данных, инвертируя воображаемый компонент данных временной области для всех экземпляров n.

Примечание: Данные о частотности и временном диапазоне являются сложными переменными. Обычно воображаемая составляющая сигнала временной области после обратного FFT либо равна нулю, либо игнорируется как ошибка при округлении. Повышение точности переменных с 32-битного плавающего значения до 64-битного двукратного значения или 128-битного двойного значения значительно снижает погрешность округления, возникающую в результате нескольких последовательных операций FFT.

Пример кода (Си/C++)

```
#include <math.h>  
  
// PI для вычисления sin или cos  
#define PI      3.1415926535897932384626433832795  
  
// 2*PI для вычисления sin или cos  
#define TWOPi   6.283185307179586476925286766559  
  
// Коэффициент для перевода из градусов в радианы  
#define Deg2Rad 0.017453292519943295769236907684886  
  
// Коэффициент для перевода из радиан в градусы  
#define Rad2Deg 57.295779513082320876798154814105  
  
// Логарифм числа 2 по основанию 10  
#define log10_2 0.30102999566398119521373889472449  
  
// 1/Log10(2)  
#define log10_2_INV 3.3219280948873623478703194294948
```

```

// структура: комплексная переменная (двойная точность)
struct complex
{
public:
    double Re, Im;           // Не так уж и сложно в конце концов
};

void rad2InverseFFT(int N, complex *x, complex *DFT)
{
    // M - это количество итераций, которые нужно выполнить. 2^M = N
    double Mx = (log10((double)N) / log10((double)2));
    int a = (int)(ceil(pow(2.0, Mx)));
    int status = 0;
    if (a != N) // Проверить, является ли N степенью 2
    {
        x = 0;
        DFT = 0;
        throw "rad2InverseFFT(): N must be a power of 2 for Radix 2 Inverse FFT";
    }

    complex *pDFT = DFT;           // Сбросить вектор для указателей DFT
    complex *pX = x;              // Сбросить вектор для указателя x[n]
    double NN = 1 / (double)N;    // Коэффициент уточнения для обратного FFT
    for (int i = 0; i < N; i++, DFT++)
        DFT->Im *= -1; // Найти комплексное сопряжение частотного спектра

    DFT = pDFT;                  // Сбросить DFT(Указатель Частотной Области)
    rad2FFT(N, DFT, x); // Выполнить прямой FFT с замененными переменными

    int i;
    complex* x;
    for (i = 0, x = pX; i < N; i++, x++){
        x->Re *= NN; // Разделить временную область на N для точного вычисления амплитуды
        x->Im *= -1; // Изменить обозначение ImX
    }
}

```

Приложение А: Псевдокод

Раздел А.1: Переменные аффектации

Вы можете описать переменную аффектацию по-разному.

С типом

```
int a = 1
int a := 1
let int a = 1
int a <- 1
```

Без типа

```
a = 1
a := 1
let a = 1
a <- 1
```

Раздел А.2: Функции

До тех пор, пока имя функции, оператор возврата и параметры явные, вы в порядке.

```
def incr n
    return n + 1
```

или

```
let incr(n) = n + 1
```

или

```
function incr (n)
    return n + 1
```

все достаточно понятные, поэтому вы можете использовать их. Постарайтесь не быть двусмысленными с переменной аффектацией.

Титры

Большое спасибо всем людям из Stack Overflow Documentation, которые помогли предоставить этот контент, больше изменений может быть отправлено на web@petercv.com для нового контента, который будет опубликован или обновлен

Abdul Karim	Глава 1
afeldspar	Глава 43
Ahmad Faiyaz	Глава 28
Alber Tadrous	Глава 53
Anagh Hegde	Главы 29 и 39
Andrii Artamonov	Глава 27
Anukul	Глава 40
Bakhtiar Hasan	Главы 9, 11, 14, 17, 19, 20, 22, 40, 41, 42, 47, 52 и 54
Benson Lin	Главы 14, 39 и 44
brijs	Глава 39
Chris	Глава 15
Creative John	Главы 49 и 51
Dian Bakti	Глава 10
Didgeridoo	Главы 2 и 43
Dipesh Poudel	Глава 21
Dr. ABT	Глава 55
EsmaeelE	Главы 2, 29, 30, 39 и 50
Filip Allberg	Главы 1 и 9
ghilesZ	Глава 17
goeddek	Глава 18 и 27
greatwolf	Глава 5
Ijaz Khan	Глава 29
invisal	Глава 31
Isha Agarwal	Главы 4, 5, 6, 7 и 8
Ishit Mehta	Глава 5
IVlad	Глава 16 и 28
Iwan	Глава 30
Janaky Murthy	Глава 6
JJTO	Глава 9
Julien Rousé	Глава 24
Juxhin Metaj	Главы 2 и 30
Keyur Ramoliya	Главы 23, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 45, 47, 48 и 50
Khaled.K	Глава 39
kiner_shah	Глава 12
lambda	Глава 38
Luv Agarwal	Глава 30
Lymphatus	Глава 31
M S Hossain	Глава 17

<u>Malav</u>	Глава 33
<u>Malcolm McLean</u>	Глава 4 and 39
<u>Martin Frank</u>	Глава 21
<u>Mehedi Hasan</u>	Глава 5
<u>Miljen Mikic</u>	Главы 2, 28 и 39
<u>Minhas Kamal</u>	Главы 12 и 46
<u>mnoronha</u>	Главы 23, 29, 31, 32, 33, 34, 35, 36 и 45
<u>msohng</u>	Глава 39
<u>Nick Larsen</u>	Глава 2
<u>Nick the coder</u>	Глава 3
<u>optimistanoop</u>	Главы 29 и 33
<u>Peter K</u>	Глава 2
<u>Rashik Hasnat</u>	Глава 40
<u>Roberto Fernandez</u>	Глава 12
<u>samgak</u>	Глава 29
<u>Samuel Peter</u>	Глава 3
<u>Santiago Gil</u>	Глава 30
<u>Sayakiss</u>	Глава 9 и 14
<u>SHARMA</u>	Глава 30
<u>ShreePool</u>	Глава 39
<u>Shubham</u>	Глава 16
<u>Sumeet Singh</u>	Главы 20 и 41
<u>TajyMany</u>	Главы 12 и 13
<u>Tejas Prasad</u>	Главы 2, 5, 9, 11, 18, 19 и 45
<u>theJollySin</u>	Глава 17
<u>umop apisdn</u>	Глава 39
<u>User0911</u>	Глава 29
<u>user23013</u>	Глава 9
<u>VermillionAzure</u>	Глава 4 и 9
<u>Vishwas</u>	Главы 14, 25 и 26
<u>WitVault</u>	Глава 3
<u>xenteros</u>	Главы 17, 29 и 39
<u>Yair Twito</u>	Глава 2
<u>yd1</u>	Глава 4
<u>Yerken</u>	Главы 16 и 20
<u>YoungHobbit</u>	Глава 29

Вам также может понравиться

