Алгоритмы
(Computer Science)

2008 г. Осенний семестр,
Третий курс

Примерный план

- Бинарный поиск
- Сортировка
- Структуры данных
- Двоичное дерево поиска
- Хэш-таблицы
- Алгоритмы для строк
- Суффиксное дерево
- Сложность текста и сжатие
- Графы, поиск в ширину и в глубину
- Связная компонента, покрывающее дерево, Эйлеров цикл
- Оптимальный путь и динамическое программирование
- Полукольца
- Минимальное время работы сортировки
- NP и NP-полные задачи
- Примеры NP-полных задач
- Стохастические алгоритмы

Пример:

упорядочить числа

Алгоритм

1. Идем вдоль массива и находим инверсию
2. Находим в массиве место куда надо вставить «неправильный» элемент
3. Освобождаем место для вставки
4. Вставляем элемент
5. Идем дальше вдоль массива до следующей инверсии
6. Переходим к шагу 2

Алгоритмы везде!

Как попасть на лекцию

1. Выйти из дома и дойти до остановки автобуса
2. Доехать до станции метро
3. Сесть в поезд и доехать до станции «Университет»
4. Дойти до факультета
5. Подняться на 4-й этаж
6. Войти в аудиторию

Алгоритм предполагает умение

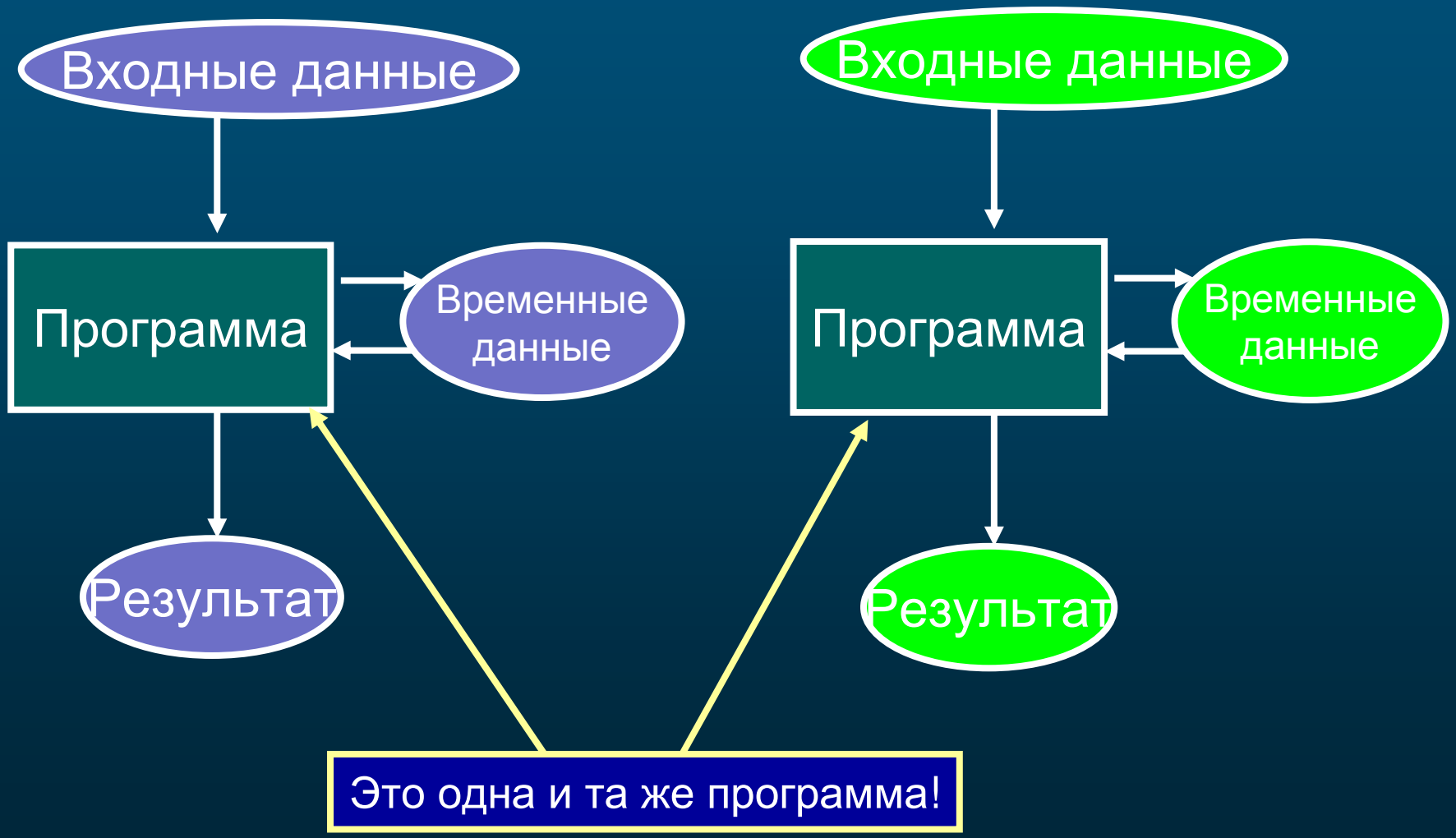
- Надо уметь ходить
- Надо знать в какой автобус сесть
- Надо уметь войти в метро и сесть в правильном направлении
- Надо знать как дойти до факультета

Алгоритмы для компьютера

Нам важно, что

- Компьютер состоит из процессора и памяти
- Процессор умеет делать некоторые элементарные операции (сложение, сравнение и т.п.)
- Есть линейная память
- Большинство компьютеров имеет один или небольшое (ограниченное) количество процессоров

Алгоритм для компьютера



Свойства алгоритма

- Для любых входных данных алгоритм должен закончить свою работу за конечное время
- Следствие. Алгоритм использует конечное количество памяти
- Результат должен соответствовать поставленной задаче

Элементы булевой алгебры

- Булева переменная принимает одно из значений ***true*** (истина) или ***false*** (ложь)
- Основные операции :
  - ***or*** – логическое или
  - ***and*** – логическое и
  - ***not*** – отрицание

Значение булевой функции обычно записывают в виде матрицы…

or	x	
y	|	0	1
	0	0	1
	1	1	1

and	X		
Y	&	0	1
	0	0	0
	1	0	1

not	~
0	1
1	0

… или таблицы

X, Y	**and**
0 0	0
0 1	0
1 0	0
1 1	1

X, Y	**or**
0 0	0
0 1	1
1 0	1
1 1	1

X	not
0	1
1	0

X,Y,Z	f (X,Y,Z)
0 0 0	0
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	1
1 0 1	0
1 1 0	0
1 1 1	1

Пример

$$F(X,Y) = (X \,\&\, Y) \,/\, (X \,\&\, \sim Y) = ?$$

	x		
y		0	1
	0	0	1
	1	0	1

$$F(X,Y) \equiv X$$

Несколько дополнительных функций

XOR	x	
y	^	0	1
	0	0	1
	1	1	0

EQ	x	
y	==	0	1
	0	1	0
	1	0	1

	x		
y	=>	0	1
	0	1	1
	1	0	1

Задачи

- Сколько всего разных булевых функций от двух переменных ? от n переменных?
- Написать таблицу значений для функций:
  - f1= ((x & y) | (x & z)) & ~ z
  - f2= (x & y) | (~x & ~y)
  - f3= (x & y) | ~(x &~y) | ~(~x & y) | (~x & ~y)
  - f4= (x & y) | (x &~y) | (~x & y). что э то за функция?

Задача потруднее

- Построим булеву функцию от *n* переменных *x1,x2,…,xn* следующим образом:
- для начала напишем ***все*** комбинации вида ([~]x1 & [~]x2 & … & [~]xn), где [~] означает, что в каких-то комбинациях берем отрицание, а в каких-то – нет. Сколько всего разных комбинаций такого вида?
- Затем все полученные комбинации объединяем знаком |.
- Чему равно значение этой функции. Потренируйтесь на n=1,2,3. Выдвиньте гипотезу. Докажите ее.

Важные тождества

```
~ ( X | Y )  ≡  ( ~X )  &  ( ~Y )
~ ( X & Y )  ≡  ( ~X )  |  ( ~Y )
   X | true  ≡  true
   X & true  ≡  X
   X | false ≡  X
   X & false ≡  false
     X ^ Y   ≡  ( X | Y ) & (~X  | ~Y )
      X==Y   ≡  ( X & Y ) | (~X & ~Y )
       X ^ Y ≡ ~(X==Y)
```

Теорема

Любая логическая функция любого количества аргументов может быть представлена в виде комбинации операций ***not, or, and***

Соображения к доказательству

x y z	*f*	*f =*				
000	1	(	~x &	~y &	~z)	|
001	0		false			
010	0		false			
011	0		false			
100	1	(	x &	~y &	~z)	|
101	1	(	x &	~y &	z)	|
110	0		false			
111	1	(	x &	y &	z)	

Доказательство

- Доказательство по индукции.
- для n=1:

  есть 4 типа функции от одной переменной:

  `true, false, x, ~x.`

  `(x | ~x); (x & ~x); (x); (~x)`
- Для любого n

  $f(x_1, x_2, \ldots, x_{n-1}, x_n) =$

  $(f(x_1, x_2, \ldots, x_{n-1}, true) \ \& \ x_n) \ |$

  $(f(x_1, x_2, \ldots, x_{n-1}, false) \ \& \sim x_n)$
- Отметим, что $f(x_1, x_2, \ldots, x_{n-1}, true)$ и $f(x_1, x_2, \ldots, x_{n-1}, false)$ являются функциями *n-1* переменной, и по предположению индукции могут быть представлены в виде комбинации базовых операций

Двоичная и шестнадцатеричная арифметика

+	0	1
0	0	1
0	1	10

*	0	1
0	0	0
1	0	1

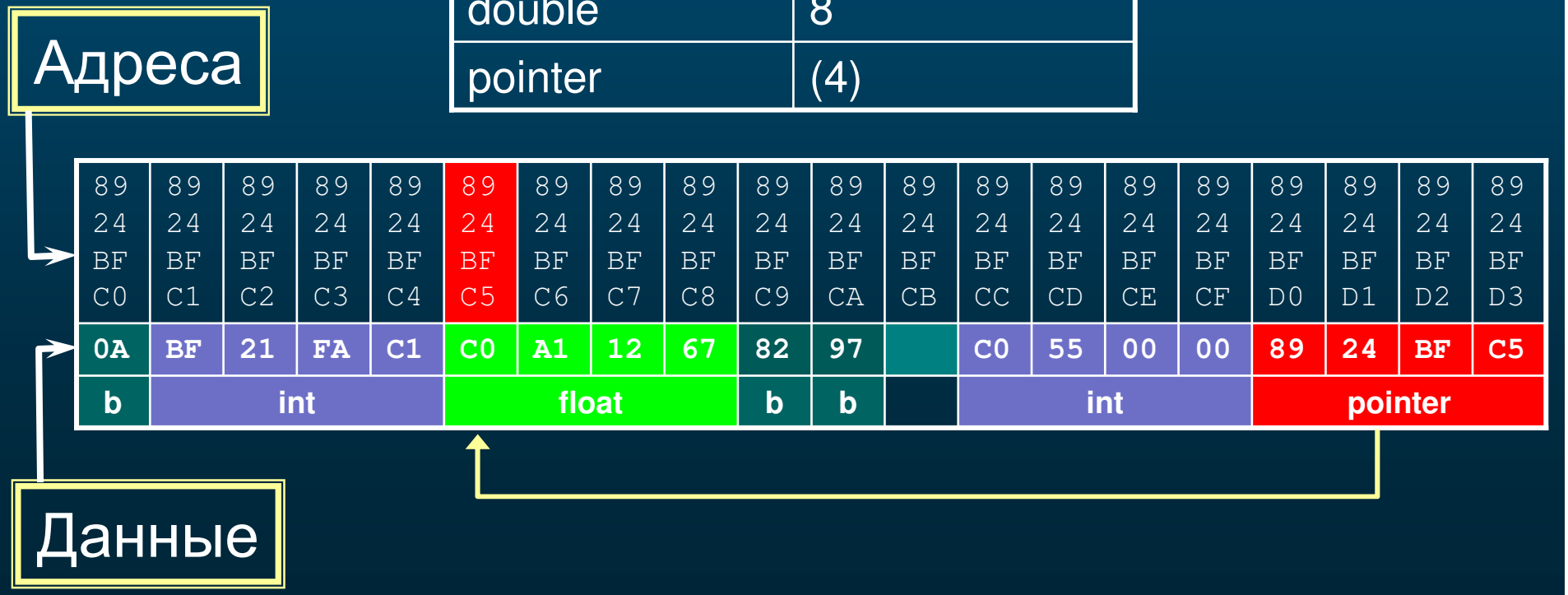
0=0000	4=0100	8=1000	C=1100
1=0001	5=0101	9=1001	D=1101
2=0010	6=0110	A=1010	E=1110
3=0011	7=0111	B=1011	F=1111

Основные типы данных

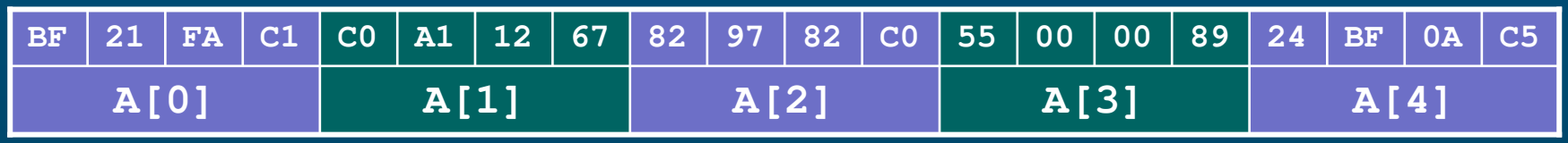
- Байт
- Целое число без знака
- Целое число со знаком
- Число с плавающей точкой

Организация данных в памяти
(очень приблизительно)

тип	размер
byte	1
int	4
float	4
double	8
pointer	(4)



Массивы



- Пронумерованное множество заранее определенного размера:

`A[0..5]`

- К элементу можно обратиться по его индексу (номеру)

`A[0]`

`A[i]`

- Массив лежит в памяти подряд!
- Размер всех элементов одинаковый

Более сложные данные (структуры)

Телефонный номер: <имя><код города><номер>

валя	**301**	**7654**
иванов	**095**	**1234**
петров	**312**	**2345**
сидоров	**312**	**4556**
тяня	**301**	**5667**

и	в	а	н	о	в									0	9	5	1	2	3	4
п	е	т	р	о	в									3	1	2	2	3	4	5
с	и	д	о	р	о	в								3	1	2	4	5	5	6
т	а	н	я											3	0	1	5	6	6	7
в	а	л	я											3	0	1	7	6	5	4

Отступление про использование указателей

тяня	pWash	5667
валя	pWash	7654
иванов	pMoscow	1234
петров	pChicago	2345
сидоров	pChicago	4556

Таблица кодов

pWash	301
pMoscow	095
pChicago	312

Изменение одного кода приводит к одновременному изменению кодов в основной таблице

Структуры

- Описание структуры:

```
Telephone
{
    byte name[14];
    byte areaCode[3];
    byte phone[4];
}
```

- Отдельная запись выглядит так:

```
Telephone Ivanov;
```

- Доступ к элементам структуры будем записывать так:

```
code = Ivanov.areaCode;
```

Массив

- Пронумерованное множество заранее определенного размера

```
Telephones[5]
```

- К элементу можно обратиться по его индексу (номеру)

```
Ivanov=Telephones[0]
Persone=Telephones[i]
```

- Массив лежит в памяти подряд!
- Размер всех элементов одинаковый
- Массив не обязательно упорядочен

Поиск в массиве

Задача: найти Сидорова

1. i=0
2. Берем элемент № i
3. Проверим не Сидоров ли он
4. Если Сидоров, то **Ура**
5. Иначе переходим с следующему элементу: i=i+1
6. Если массив не кончился, то идем к 2
7. **Увы!**

Другая запись

```
Telephone SearchPersoneByName
                                (String searchName )
{
  for(i=0; i<nTelephones; i=i+1)
  {
      if(Telephones[i].searchName == name)
                return Telephnes[i];
  }
  return “not found”;
}
```

Анализ алгоритма

- Сколько записей надо просмотреть, чтобы найти Сидорова?
- Если Сидорова нет, то nTelephones
- Если он есть, то примерно nTelephones/2

Итого: время поиска пропорционально N

$$T=O(N)$$

Чем хороши упорядоченные (сортированные) массивы ?

Вспомним, как ищется слово в словаре.

Алгоритм:

- Делим массив пополам.
- Проверяем, попало ли слово на середину, и если да, то **Ура**
- Смотрим, следует ли искать слово в левой или в правой половине
- ***Ищем в соответствующей половине***

Для такого поиска хорошо бы иметь процедуру поиска в части массива

Схема поиска слова "Добро"

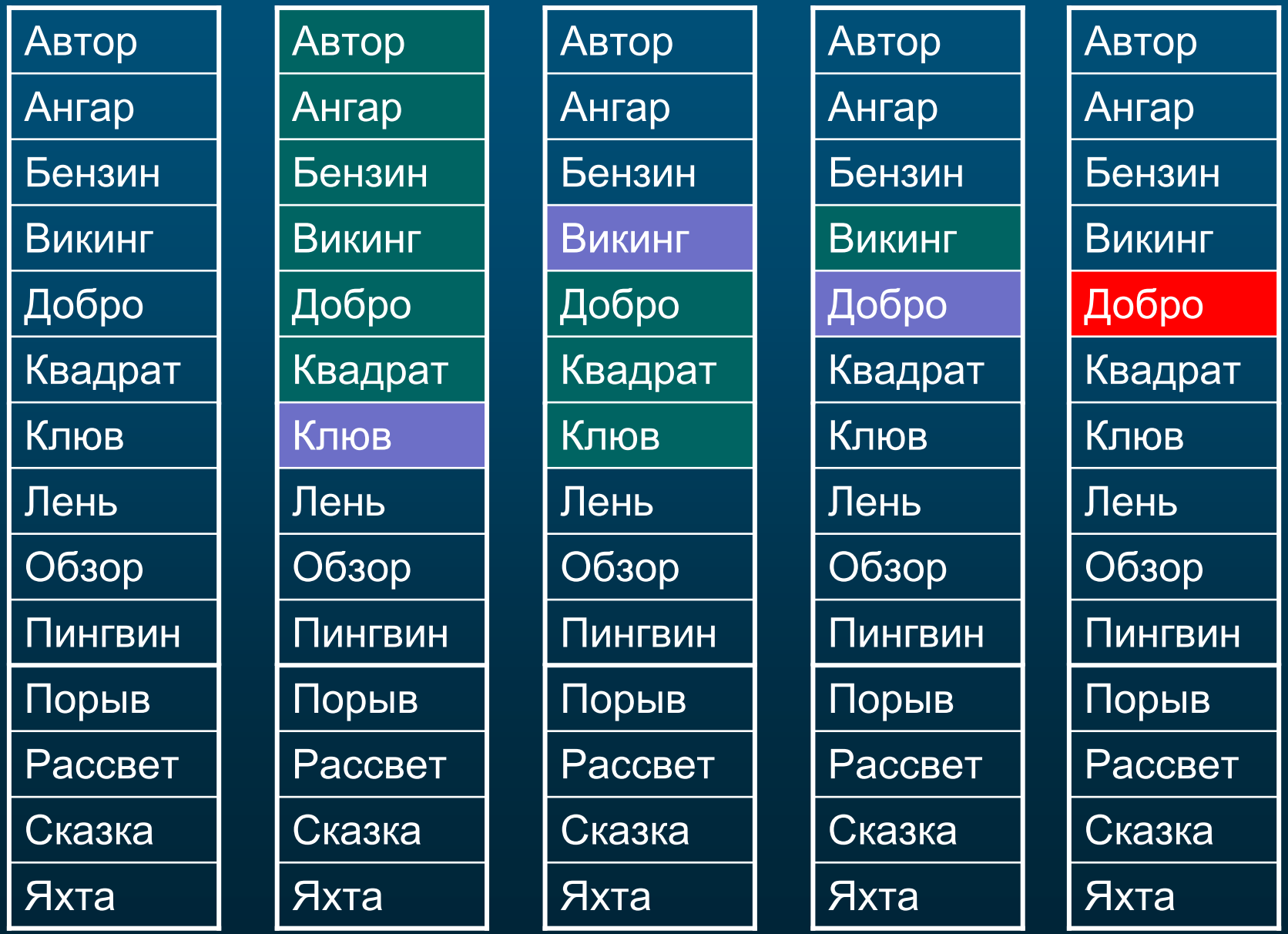
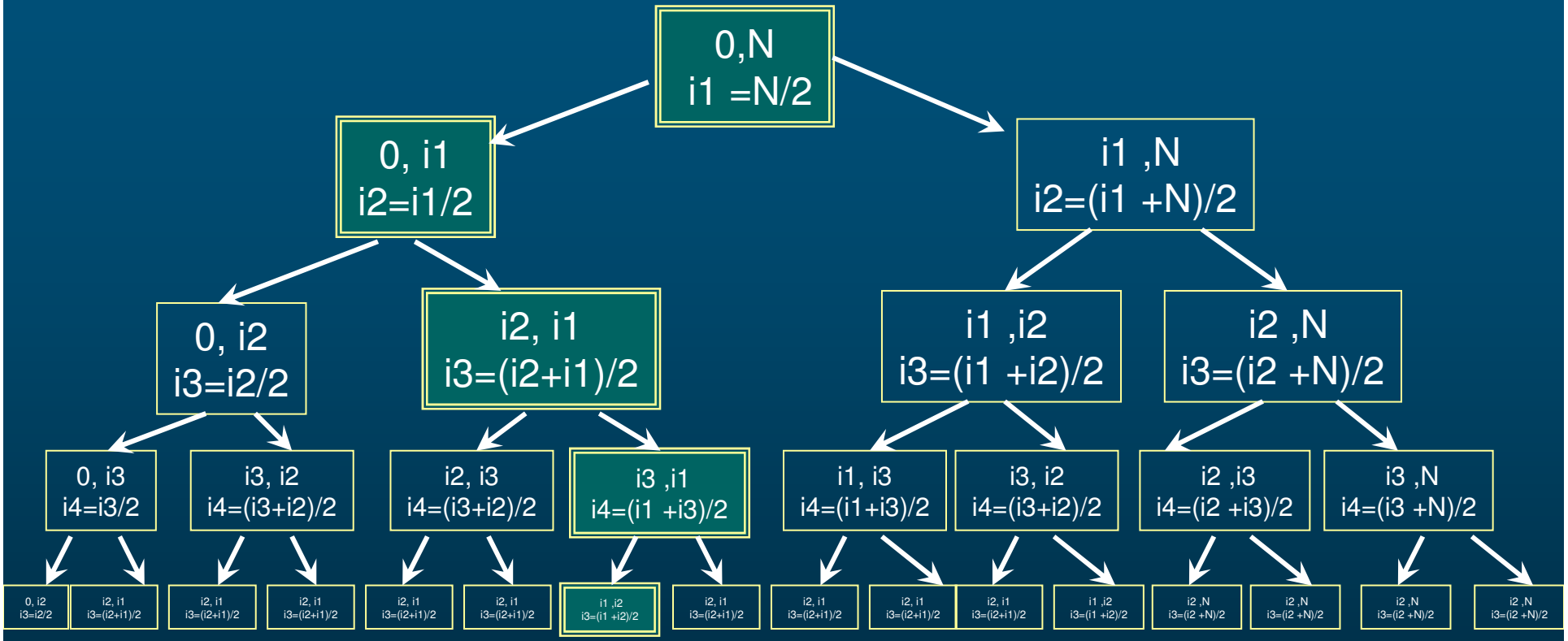
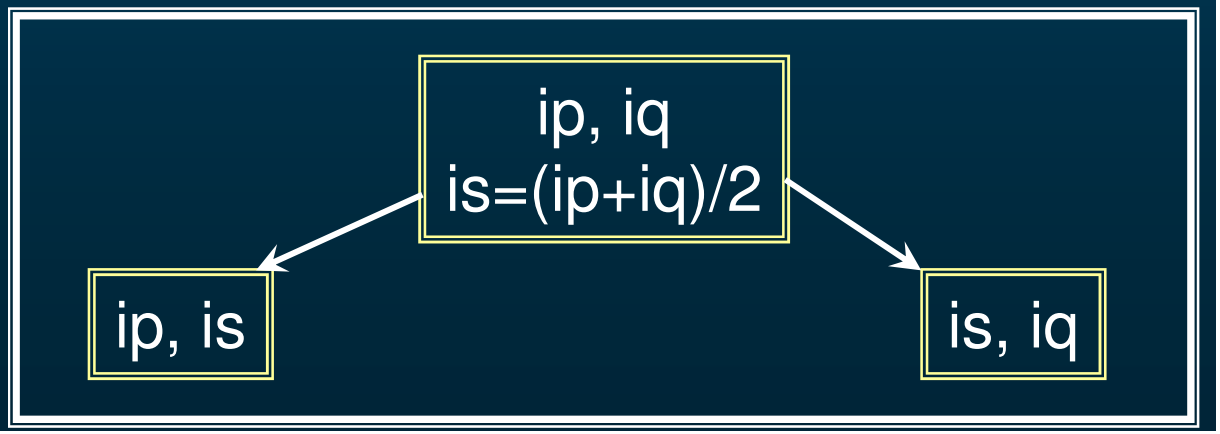


Схема алгоритма



Дерево бинарного поиска



Бинарный поиск

```
Telephone BSearch(String name)
   {return BSearch(name,0,nTelephones);}

Telephone BSearch (String name, int i0, int i1)
{
   if(i0 == i1) return "not found";
   int i=(i0+i1)/2;
   if(Telephones[i].name == name)
       return Telephones[i];
   if(Telephones[i].name > name)
       return BSearch (name, i0, i);
   else
       return BSearch (name, i+1, i1);
}
```

Рекурсия

- В программе мы вызывали функцию из себя самой. Это называется ***рекурсией***
- Еще пример рекурсии – рекурсивное представление факториала:

```
factorial(n)={1, n==1; n*factorial(n-1)}
```

- Рекурсивные вычисления – это продолжение метода математической индукции.

Анализ алгоритма

- В худшем случае время работы алгоритма равно количеству делений массива пополам, т.е. высоте дерева поиска.
- Если высота дерева известна и равна k, то возможно k делений, и для размера массива N имеем оценку:

$$2^{k-1} \le N \le 2^k$$

или

$$k-1 \le \log_2 N \le k$$

Откуда получаем, что время работы алгоритма

$$T(N)=O(\log N)$$

Сравнение с наивным алгоритмом поиска

- Наивный алгоритм: $T_{naive}(N)=O(N)$
- Бинарный поиск: $T_{binary}(N)=O(logN)$
- Если размер массива 10^6 (это один миллион), то

Это время на элементарный шаг

$$T_{naive} = 10^{-5} * 10^6 = 10$$

$$T_{binary} = 10^{-5} * log(10^6) = 2*10^{-4}$$

Это время на элементарный шаг

Как правильно сортировать масиив? MergeSort

Представим себе, что

- массив разбит на два отсортированных подмассива:
- и мы умеем сливать два отсортированных массива.

Тогда

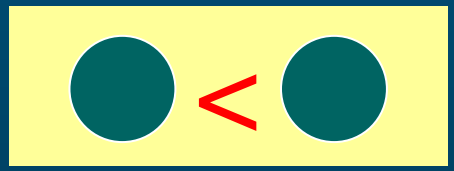
- Разбиваем массив пополам
- Сортируем каждую половину
- и сливаем

А как сортировать каждую половину?

Разбиваем ее пополам, каждую половину от половины сортируем и потом сливаем.

```
MergeSort(A,i1,i2)
{
  if(i1==i2) return;
  i=(i1+i2)/2;
  A1=MergeSort(A, i1, i);
  A2=MergeSort(A, i+1, i2);
  Merge(A1,A2,A);
}
```

Слияние массивов



Задание: Написать программу слияния массивов

Анализ Алгоритма

- Время работы:

$$T(N)=2*T(N/2)+O(N)$$

- Есть теорема, показывающая, что в этом случае

$$T(N)=O(N*\log (N))$$

- ***Недостаток:*** *нужен дополнительный массив*

Теорема о рекурсии Merge-Sort

Если $T(N)=2*T(N/2)+O(N) \le 2*T(N/2)+bN$

то существует C такое, что для любого N

$$T(N) \le C\, N \log N$$

Индукция :

При N=3 $T(N)$ равен чему-то, например,

$T(3) = c\, 3 \log 3.$

Допустим, утверждение верно для $n/2$.

Тогда

$$T(n) \le 2\, T(n/2)+ bn \le$$
$$\le C\, n \log n/2 + bn =$$
$$= C\, n \log n + n\,(b - C \log 2) \le$$ [полагая $C > b/\log 2$]
$$\le C\, n \log n$$

Быстрая сортировка QSort

- Свойство отсортированного массива. Массив отсортирован тогда и только тогда, когда ***для любого элемента q все элементы массива слева не больше его, а все элемента справа – не меньше***
- Идея: для какого-нибудь ***q*** переставим элементы массива так, чтобы все, что слева было меньше всего, что справа (при этом не будем заботиться об упорядоченности левой и правой частей)

QSort

- Берем первый элемент в массиве и попробуем сделать перестановки так, чтобы слева от него были элементы меньше, а справа – больше его.
- Ищем с правого конца первый элемент, который меньше выбранного и меняем их местами
- Ищем слева первый элемент, который больше выбранного и меняем их местами
- Повторяем процедуру до тех пор, пока массив не окажется разбитым на два подмассива.
- После разбиения можно применить ту же процедуру к левой и правой частям массива.

QSort

```
QSort (Array A)
{QSort (A, 0, N);}

QSort (Array a, int i0, int i1)
{
  if(i0==i1) return;
  int i=Partition (A,i0,i1);
  QSort (A,i0,i);
  QSort (A,i+1,i1);
}
```

Задание: написать процедуру Partition

Анализ алгоритма QSort

- Точный анализ выходит за рамки курса
- Алгоритм делит массив на части, но не равные. В среднем происходит деление почти пополам, так что в среднем оценка $T=O(N \log(N))$
- Худший случай – когда массив уже отсортирован. В этом случае время работы алгоритма $T=O(N^2)$
- Можно за время $O(N)$ перетасовать массив случайным образом и затем отсортировать его за время $T=O(N \log(N))$

Что лучше?

- По мере добавления элементов вставлять их на правильное место
- Сначала собрать массив, а потом отсортировать.

Сборка массива:

на каждом этапе надо найти место для вставки: *T~log i*
сделать вставку *T~i*

Итого

$$T = \sum_i (\alpha i + \beta \ln i) = \alpha \frac{n(n-1)}{2} + \beta \sum_i \ln i = O(n^2)$$

Упорядоченный массив - недостатки

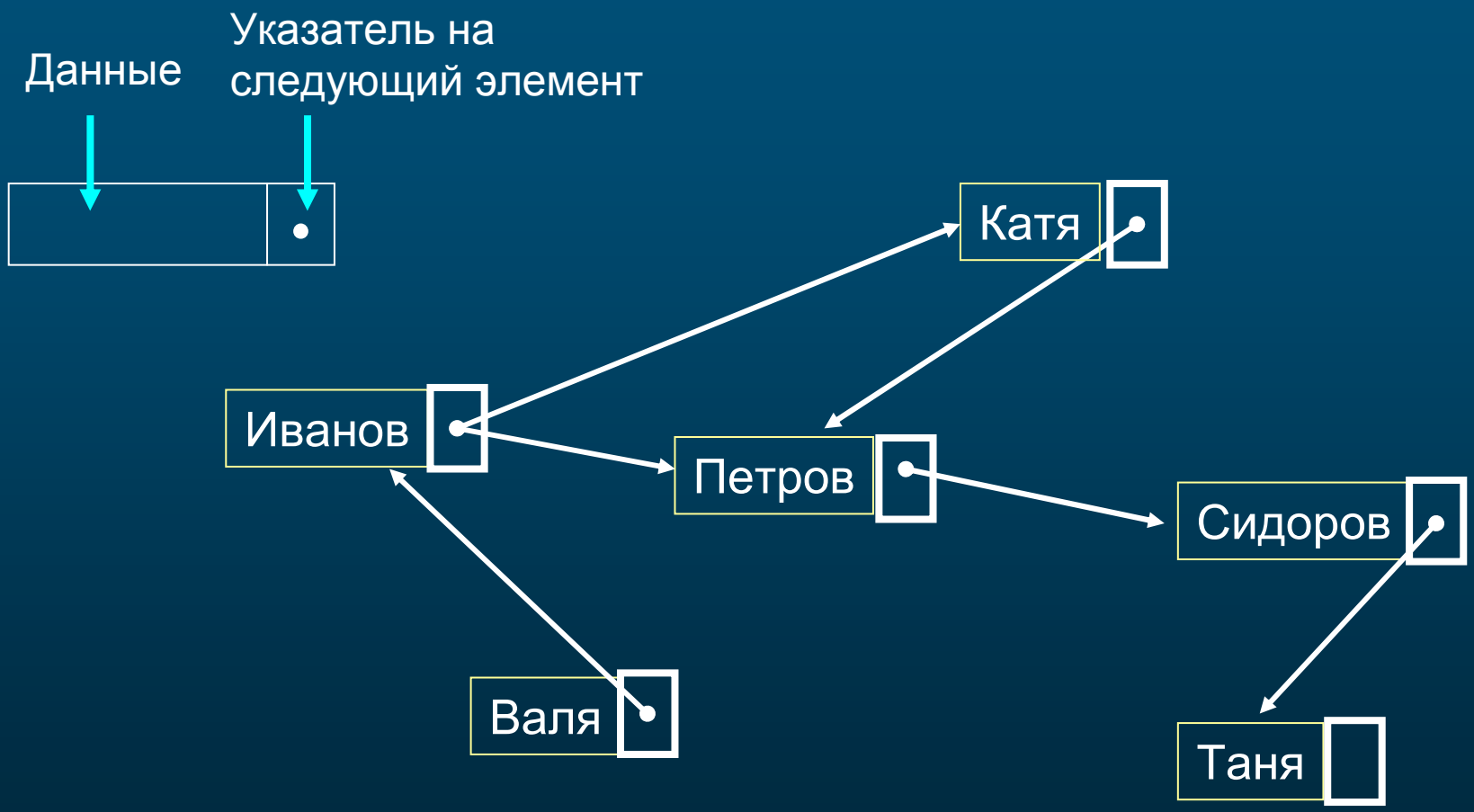
Добавление элемента требует усилий

Надо:

1. Найти место куда вставить ($T=O(\log N)$)
2. Освободить место для вставки ($T=O(N)$)
3. Вставить элемент

Нет гарантии, что размер массива позволит вставить элемент

Связный список



Связный список (односвязный список)

```
LinkListItem
{
  Telephone Data;
  pLinkListItem  next;
}

LinkListItem root;
```

Добавление элемента

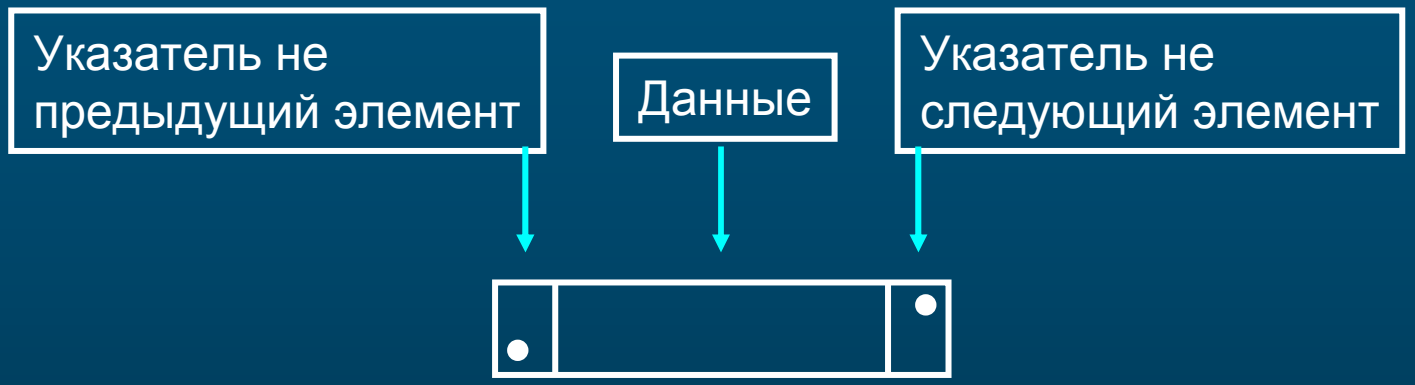
```
InsertAfter(LinkList item,
            LinkList newItem)
{
LinkList nextItem = item.next;
item.next = newItem;
newItem.next = nextItem;
}
```

Удаление элемента

Для удаления элемента надо найти предыдущий элемент (*T=O(N)*)

```
Remove(LinkList item)
{
   LinkList current;
   for(current=root; current!=null;
   current=current.next)
   {
       if(current.next==Item)
              {
              current.next=Item.next;
              return;
              }
   }
   ERROR
}
```

Двусвязный список



Двусвязный список

```
BiLink
{
  pBiLink previous;
  Telephone Data;
  pBiLink next;
};

BiLink root;
```

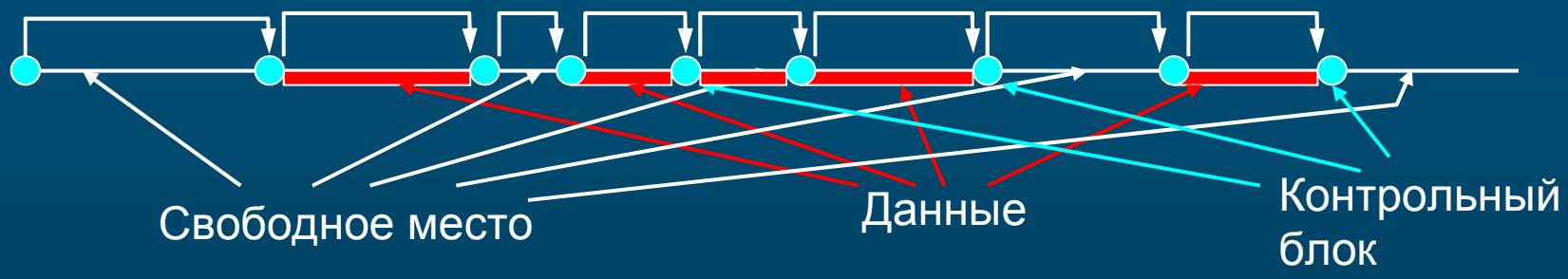
Удаление элемента

```
Remove(BiLink item)
{
   BiLink prevItem = item.previous;
   BiLink nextItem = item.next;
   if( prevItem != null)
      prevItem.next = nextItem;
   if( nextItem != null)
      nextItem.previous = prevItem;
}
```

Задача.

Напишите алгоритм вставки элемента

Распределение памяти



- Память организована в виде цепочки занятых и свободных участков
- Участки имеют в своем составе контрольные блоки
- Контрольные блоки содержат:
  - Размер свободного участка
  - Указатель (адрес) следующего контрольного блока
  - Указатель (адрес) предыдущего контрольного блока

Операции с памятью

- CB{
  size, // *размер блока памяти*
  next, // *адрес следующего контрольного блока*
  prev, // *адрес предыдущего контрольного блока*
  free, // *признак занятости блока памяти*
  }
- Операции:
  - Занять память
  - освободить память

Освобождение памяти

- Надо:
  1. проверить если предыдущий блок свободен, то объединить блоки
  2. Если следующий свободен, то объединить блоки
  3. Пометить блок как свободный

Захват памяти

- Дан размер области, который надо занять
- Надо:
  1. пройти по цепочке блоков и найти подходящий блок памяти: он (1) должен быть свободен; (2) его размер не должен быть меньше, чем запрашиваемая область
  2. Разбить блок на две части, одну из которых пометить занятой, а другую - свободной

Очередь (Queue)

- Очередь. Множество элементов, позволяющее добавлять один элемент и извлекать один элемент. При этом соблюдается правило: первым извлекается элемент, пришедший первым (FIFO). (Вспомните очередь в столовой)
- Строится на базе двусвязного списка

Задача:

Построить структуру типа очередь. Должны быть реализованы функции Put и Get. Функция Get ***удаляет извлеченный элемент***

Стек (Stack)

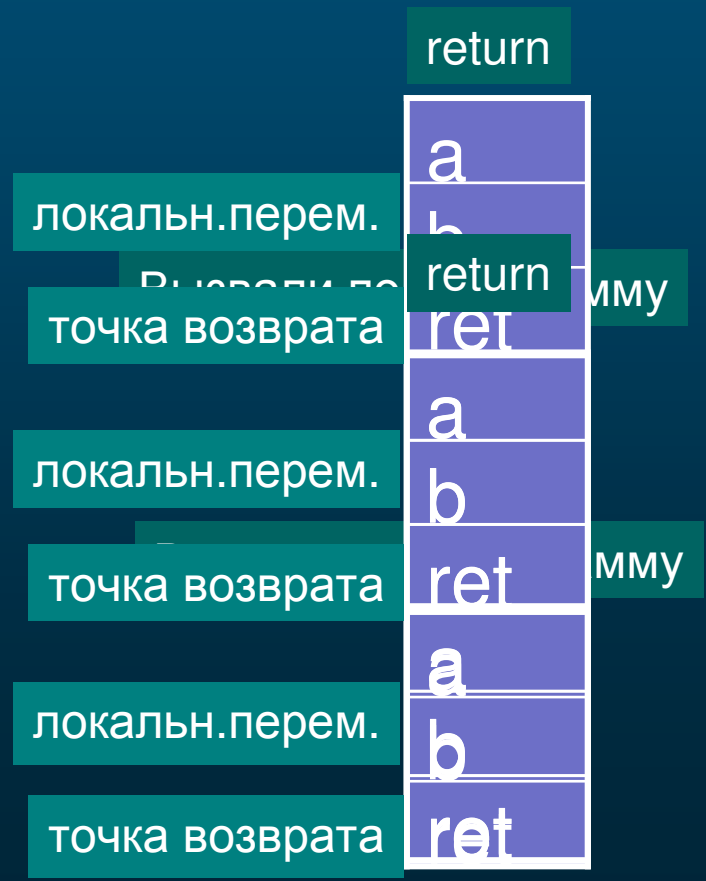
- Стек. Множество элементов, позволяющее добавлять один элемент и извлекать один элемент. При этом соблюдается правило: первым извлекается элемент, пришедший последним (LIFO). (Представьте себе стопку тарелок)
- Строится на базе (обратного) связного списка

Задача:

Построить структуру типа Стек. Должны быть реализованы функции Push и Pop. Функция Pop удаляет извлеченный элемент

В компьютере

- очередь – очередь задач
- стек – стек вызовов процедур



Проблема: Плохо искать

- У нас нет возможности найти середину списка!

Еще структура

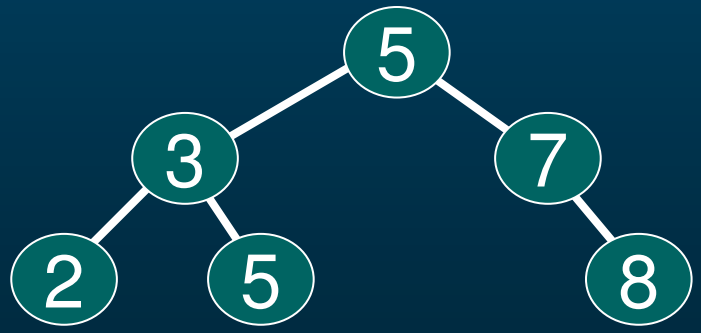
Бинарное дерево

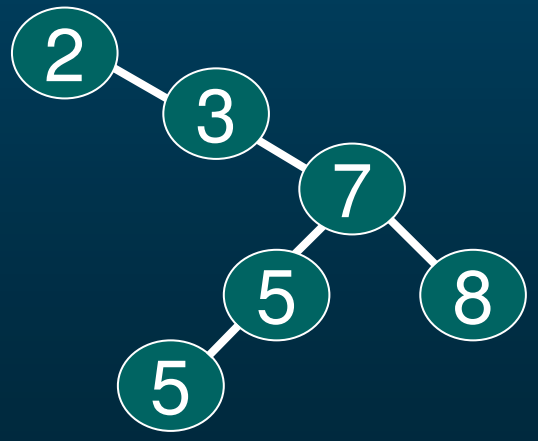
```
BTree
{
  BTree parent;
  BTree left;
  BTree right;
  Data  data;
}
```

Если дерево организовано правильно, то и вставлять и удалять и искать сравнительно легко

Двоичное дерево поиска

- Бинарное дерево
- Для любой вершины выполнено:
  - ***left and all childs < this,* если *left* существует**
  - ***this <= right and all childs,* если *right* существует**





Поиск в двоичном дереве

- Если ключ равен вершине, то Ура
- Если он меньше вершины, то переходим на левую дочернюю вершину
- Иначе переходим на правую дочернюю вершину

Поиск в двоичном дереве

```
TreeSearch(treeElm, key)
{
  if(treeElm == null) return "not found";
  if(treeElm.data == key ) return treeElm;
  if(treeElm.data < key)
     return TreeSearch(treeElm.right,key);
  else
     return TreeSearch(treeElm.left ,key);
}
```

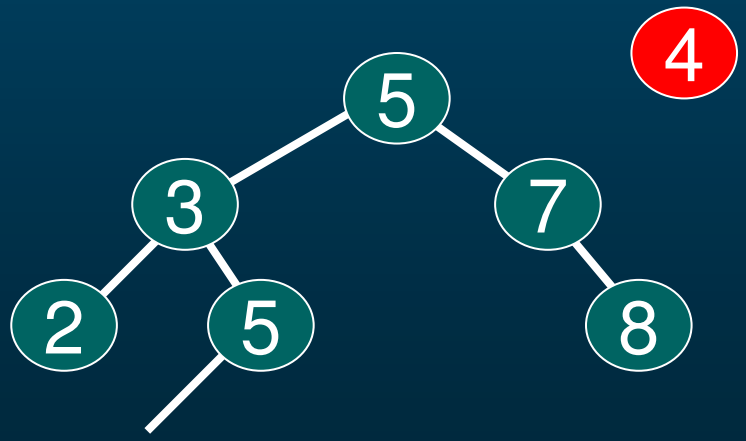
Анализ Алгоритма:

Время поиска есть $T=O(h)$,

где h – высота дерева

Добавление элемента

- Идем по дереву вниз также, как и при поиске до первого свободного места, и туда вставляем новый элемент в качестве листа дерева.



Добавление элемента (рекурсивная форма)

```
Tree_Insert(tree_elm, new_elm)
{
  if(new_elm > tree_elm)
  {
     if(tree_elm.right==null)
     {
          tree_elm.right=new_elm;
          new_elm.parent=tree_elm;
          return;
     }
     Tree_Insert(tree_elm.right, new_elm);
  }
  else … //Аналогично для left
}
```

Добавление элемента (в виде цикла)

```
InsertTree(tree_elm, new_elm)
{
  while(true)
  {
     if(tree_elm <= new_elm)
     {
          if(tree_elm.right == null)
          {
               tree_elm.right=new_elm;
               return;
          }
          tree_elm = tree_elm.right;
     }
     else … //аналогично для left
}
```

Замечания про рекурсию

- Рекурсивный тип алгоритма представляется более прозрачным.
- Часто можно записать алгоритм в виде цикла, но не всегда это легко сделать
- Использование цикла предпочтительно, поскольку несколько быстрее, и нет опасности переполнения стека.

Поиск минимального (максимального) элемента в поддереве

- Минимальный элемент в поддереве – самый левый узел
- Максимальный элемент – самый правый узел

```
MinElm(treeElm)
{
        while(treeElm.left!=null)
                treeElm=treeElm.left;
        return treeElm;
}
```

Поиск следующего (по возратанию) элемента

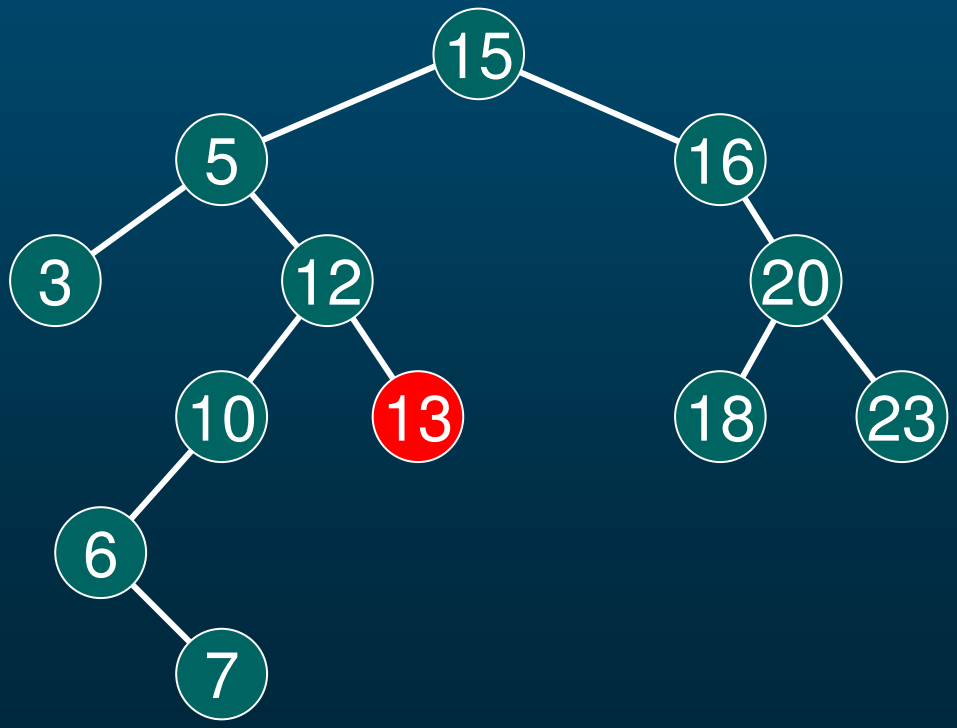
- Следующий элемент либо самый левый лист в правом поддереве, либо родитель(Почему?)

```
GetNext(treeElm)
{
      if(treeElm.right!=null)
            return MinElm(treeElm.right);
      if(treeElm.parent != null &&
         treeElm.parent.left==treeElm)
            return treeElm.parent;
      return null;
}
```

Задача: написать алгоритм поиска предшественника: GetPrev(treeElm)

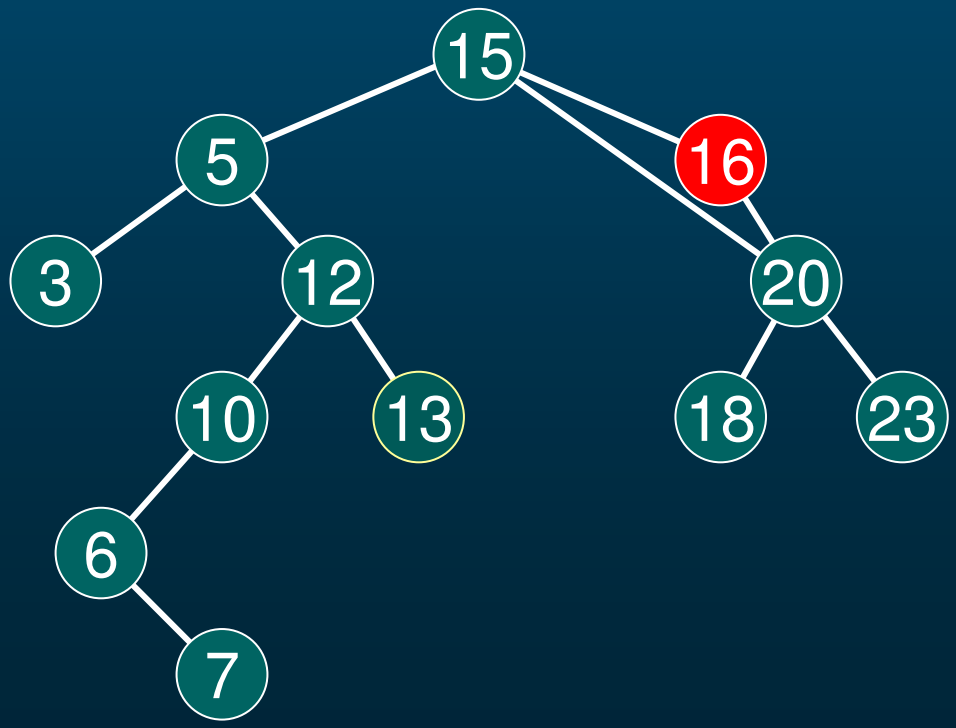
Удаление элемента.
1. Элемент не имеет детей

Надо просто его удалить. И все…



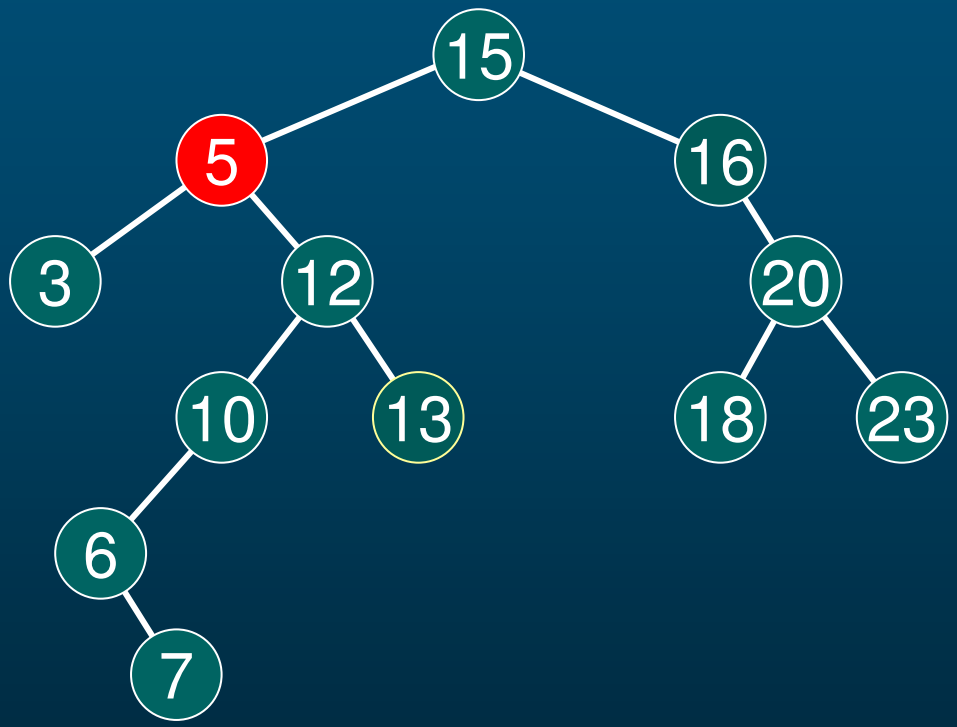
Удаление элемента.
2. Элемент имеет одного ребенка

Элемент удаляется, а его ребенок усыновляется его родителем



Удаление элемента.
3.Элемент имеет двух детей

- На его место ставится его предшественник



Задание: записать алгоритм удаления элемента

Результаты

- Поиск, удаление и вставка элемента требует времени порядка

  $T=O(h)$, где h – высота дерева
- Для одного и того же набора данных данных можно построить несколько деревьев.
- Обычно деревья хорошие:

  $h=O(\log(N))$
- Деревья можно балансировать

Задача: написать программу, которая печатает все элементы

Задача*: написать программу, которая печатает все элементы в порядке возрастания

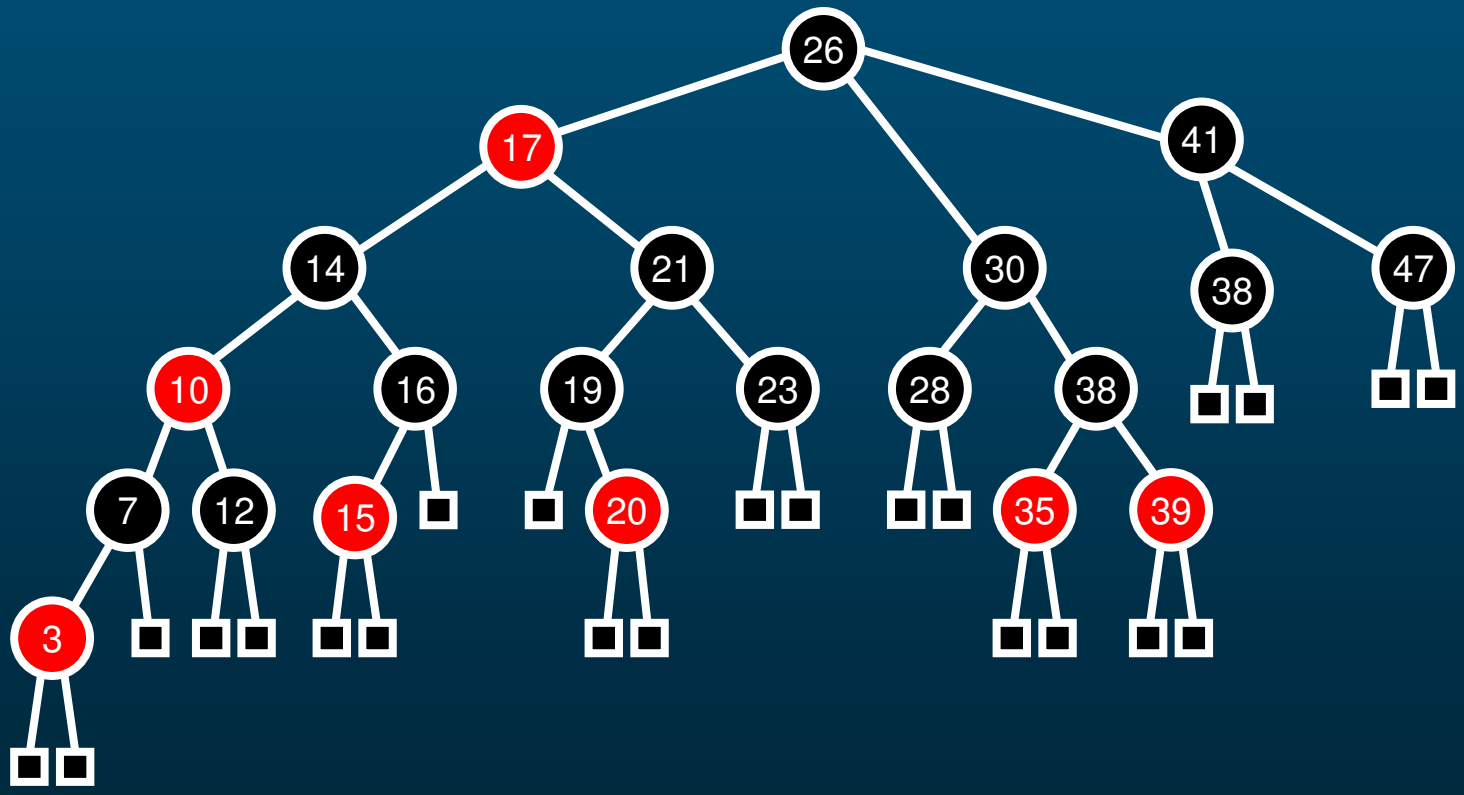
Красно-черные деревья.

- Бинарное дерево поиска
- Добавлены фиктивные листья
- Любая вершина либо черная либо красная
- Листья (фиктивные) – черные.
  Следствие: ***любая внутренняя вершина имеет двух детей.***
- У красной вершины дети черные
- Все пути от корня до листьев имеют одинаковое количество черных вершин (черная высота)

Следствие.

Любое поддерево является RB-деревом

Красно-черные деревья.



Красно-черные деревья. высота дерева

- bh – черная высота дерева
- Лемма КЧ-дерево с n внутренними вершинами имеет высоту не более

$$h \le 2 \cdot \log (n+1)$$

- Доказательство:

по индукции докажем, что число внутренних вершин в поддереве не меньше, чем

$$n \ge 2^{bh} - 1.$$

Для поддерева из одного листа – очевидно.

Пусть поддерево имеет bh=k. Тогда оба ребенка имеют bh не менее k – 1 (красный ребенок имеет bh=k-1, черный – bh=k). Тогда по предположению индукции количество внутренних вершин

$$n \ge 2^{k-1}-1+2^{k-1}-1+1=2^{k}-1.$$

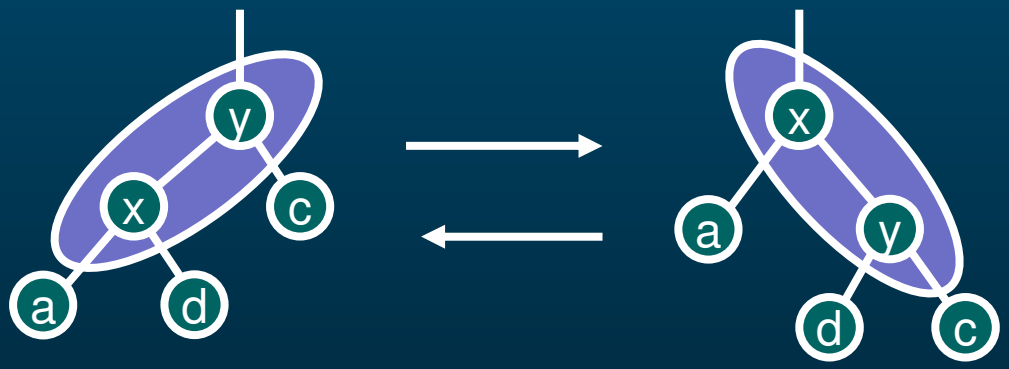
По меньшей мере половину всех вершин на пути (кроме корня) составляют черные вершины. Поэтому $h \le 2 \cdot bh$. Следовательно

$$n \ge 2^{h/2} - 1, \qquad h \le 2 \cdot \log(n+1)$$

Красно-черные деревья. преобразование деревьев

- Преобразование дерева:

Вращение



Задача: написать алгоритм для вращения. На входе вершина x

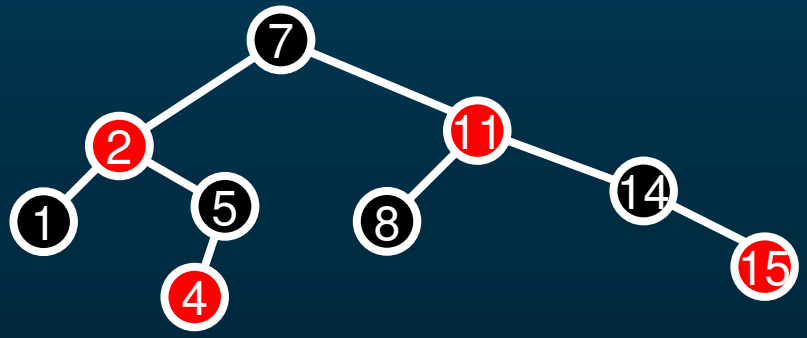
Красно-черные деревья:
Добавление элемента

- Добавляем как в обычное дерево поиска и красим в красный цвет. Восстановление RB-свойства, если только одно нарушение типа красный ребенок имеет красного родителя.

Дядя красный.
перекрашиваем
предыдущий уровень
и родителя

Родитель красный,
"дальний" дядя черный.
Вращаем и перекрашиваем
родителя (7) и деда (11)

Родитель красный,
"ближний" дядя черный.
Вращаем



Красно-черные деревья. Вставка элемента

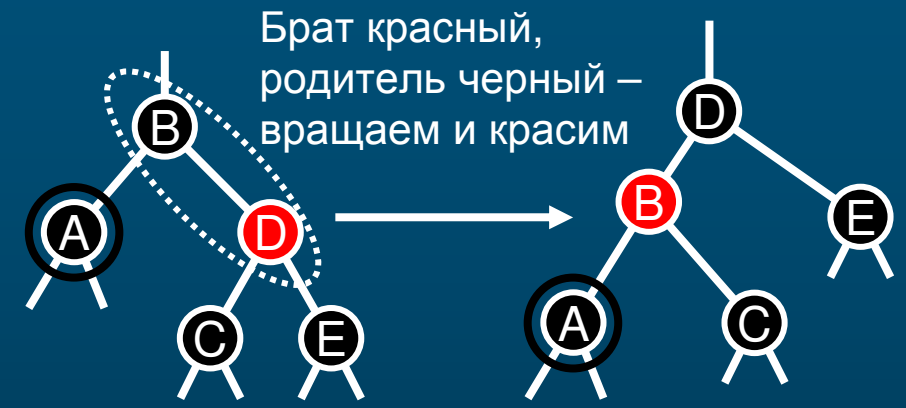
1. При восстановлении RB всегда движемся наверх
2. RB – свойства на поддеревьях сохраняются
3. Время работы алгоритма восстановления не больше высоты дерева $O(\log(n))$.

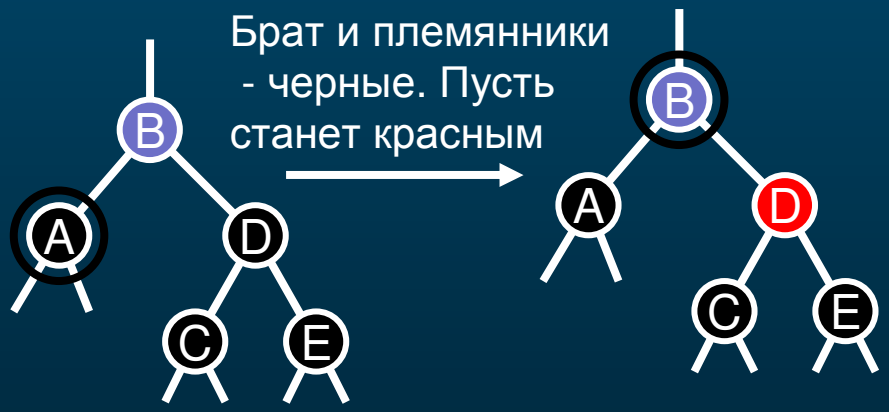
Красно-черные деревья. Удаление элемента

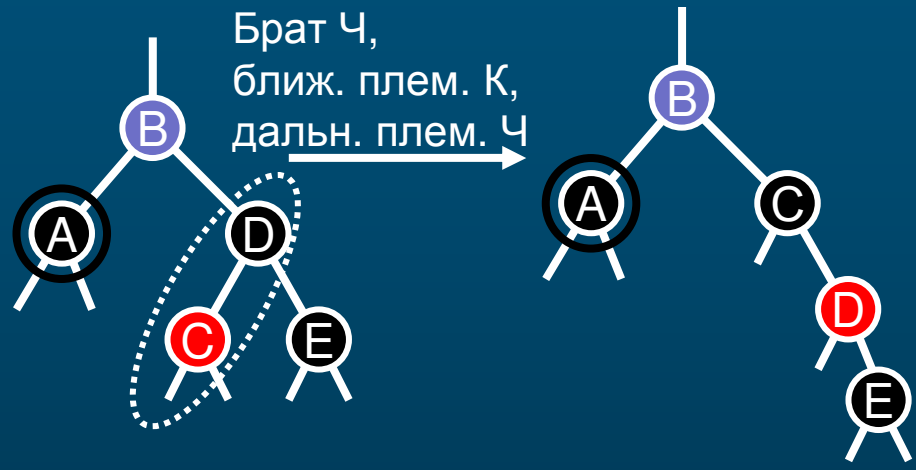
- Удаляем как обычно
- Удаление красной вершины ничего не меняет
- Удаление черной вершины меняет черную высоту.
  - Если на ее место встала красная вершина, то перекрашиваем ее в черную
  - Если на место удаленной встала черная, то надо вращать и перекрашивать. Эту новую вершину называем "дважды черной"

Красно-черные деревья.

Восстановление RB после удаления







Дважды черная

Цвет не важен

Красно-черные деревья: Операции

- Добавление элемента – как обычно, новая вершина красная. Далее – восстановление дерева
- Удаление элемента – как обычно, потом восстановление дерева. После удаления может измениться bh, поэтому алгоритм несколько другой.

Динамические порядковые статистики

Если в бинарном дереве хранить дополнительную информацию, то можно делать другие вещи.

- Поиск по номеру элемента (напр. найти пятый по возрастанию элемент)
  - Достаточно хранить в узлах размер (количество элементов) поддерева
  - При модификации дерева необходимо обновлять эту информацию

Задачи:

1. Написать алгоритм поиска элемента заданного номера
2. Написать алгоритм определения номера элемента

Хэш-Таблицы I

Хэш-Таблицы II

- Если заранее знать все мыслимые ключи, то можно было бы заранее создать индексную таблицу (быть может с пустыми строками), то нет проблем со вставкой, удалением и поиском (все операции происходят за ***O(1),*** поиск за ***O(log(nKey))***)
- **Проблема:** Полный список ключей заранее неизвестен и может быть очень велик

Key	Ref
k1	&d1
k2	
k3	&d3
k4	&d4
k5	
k6	&d6
k7	&d7
k9	&d9

Хэш-Таблицы III

- Если сделать отображение:

```
key -> int h
```

причем заранее известно, что

```
0 <= h <= hMax
```

- То время поиска будет ***O(1)***
- Отображение не обязательно взаимнооднозначное

Key	Hash	Ref
k1	0	&d1
k2	1	
k3	2	&d3
k4	3	&d4
k5, k6	4	
k7, k8	5	&d7
k9	6	&d9

Хэш-Таблицы IV

- **Пример:**

```
gagttttatcgcttccatgacgcagaagttaacactttcg
01234567890-1234567890-1234567890-1234567890
```

Хеш-функция

aa	0	ga	8
ac	1	gc	9
ag	2	gg	10
at	3	gt	11
ca	4	ta	12
cc	5	tc	13
cg	6	tg	14
ct	7	tt	15

Хэш-таблица

0	25,30	8	0,18,24,
1	19,31,33	9	10,21,
2	1,23,26,	10	
3	7,16,	11	2, 27,
4	15,22,32	12	6,29,
5	14,	13	8,13,37
6	9,20,38	14	17,
7	11,34	15	3,4,5,12,28,35,36

Хэш -Таблицы V

- **Пример**: найти **cat**.
- Вычисляем хэш-значение h=4.
- Лезем в таблицу и находим, что cat ***может*** встретиться в позициях 15,22,32
- Проверяем, что в этих позициях лежит, и убеждаемся, что 15 позиция подходит

- **Примерно так работает BLAST**

Хеш-Таблицы VI

- Видно, что во многих ячейках больше одного элемента (коллизия)
- Решение: в соответствующей ячейке хранится связный список.
- Хэш-таблица строится за время ***T=O(N)***
- Поиск происходит за время, пропорциональное заполненности таблицы

T=O(1)*k

Строки

- Задача поиска подстроки.
- ***Дано:*** длинная строка $S=s_0,s_1,\ldots,s_n$ и короткая строка $P=p_0,p_1,\ldots,p_k$, $k <= n$. (S – string, P – pattern)
- ***Найти:*** непрерывный фрагмент строки S, полностью совпадающий с P: $s_i=p_0, s_{i+1}=p_1,\ldots,s_{i+k}=p_k$

Наивный алгоритм

```
bool SubStrCmp(String s, int pos, String p)
{
  for(i=0; i<= Length(p); i++)
    if(s[i+p] != p[i]) return false;
  return true;
}
int NaiveSubstring(String s, String p)
{
  for(i=0; i<Length(s)-Length(p); i++)
          if(SubStr_Cmp(s,i,p)) return i;
     return "Not found";
}
```

- Время работы ***в худшем случае*** ***T=O(n*k);***

Оценка ***среднего*** времени работы

- Среднее время работы равно

  (количество вызовов SubStr_Cmp=n) *
  (среднее время исполнения SubStr_Cmp)
- Среднее время исполнения SubStr_Cmp =

$$1 \cdot q + 2 \cdot p \cdot q + 3 \cdot p^2 \cdot q + \ldots + (m-1) \cdot p^{m-2} \cdot q + m \cdot p^{m-1} =$$

$$(1 + 2p + 3p^2 + \ldots + (m-1)p^{m-2}) \cdot q + m \cdot p^{m-1} =$$

$$q \cdot d/dp(p + p^2 + \ldots + p^{m-1}) + m \cdot p^{m-1} \approx 1/q$$

  $p=1/A$; $q=1-p= (A-1)/ A$; A – размер алфавита
- Среднее время работы равно

$$E(T)=O(n^* A/(1 - A));$$

Алгоритм Рабина-Карпа

- Допустим, что алфавит состоит только из 9 цифр: A={1,2,3,4,5,6,7,8,9}, тогда слово – есть число и две строки равны, если соответствующие числа равны. 0 исключен, поскольку в этом случае много разных строк будет порождать одно число: 0067=067=67.
- Слово *P* может быть за время *O(k)* преобразовано в число *N(P)*
- Построка $\{s_0 \ldots s_k\}$ может быть за то же время преобразована в число *N(S(0,k));*
- Сравнение строк есть сравнение чисел. Если *N(P)==N(S(i,i+k))*, то имеем вхождение
- Далее мы двигаемся вдоль строки *S* и получаем число N(S(i+1,k+i+1). Это можно сделать за O(1). (как?)

Алгоритм Рабина-Карпа

```
int StrToNum(String p, int l)
{
  n=0;
  for(i=0; i<l; i++)
    n=n*10+(p[i]-'0');
  return n;
}

int RabinMatch(String s, String p)
{
  M=StrToNum(p,Length(p));
  N=StrToNum(s,Length(p));
  nn=10^Length(p);
  for(i=0; i<Length(s)-Length(p); i++)
    {
    if(M==N) return i;
    N=(N*10 + (s[i]-'0')) % nn;
    }
  return "Not found";
}
```

Свойства алгоритма

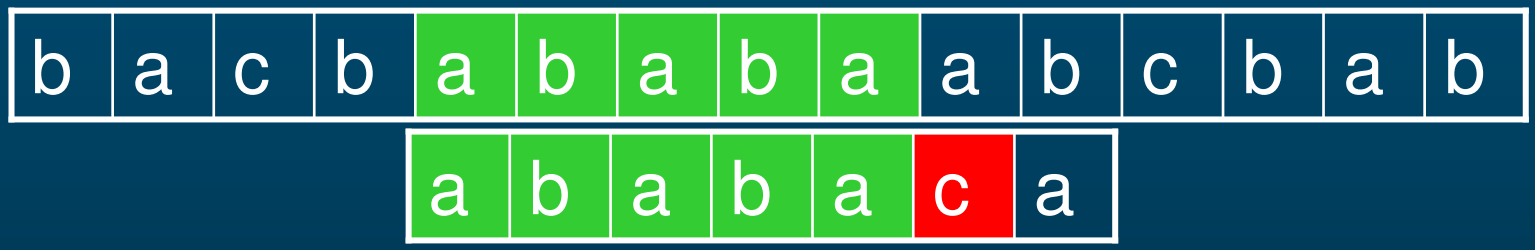
- Время работы ***T=O(n)***
- Поиск больших паттернов приводит к появлению больших чисел (больше, чем позволяет разрядная сетка)
- Очень хорош для поиска в нуклеотидных последовательностях – алфавит содержит 4 буквы, поэтому использование int позволяет искать паттерны длиной до 16, а применение long – до 32. Кроме того операции *, /, % заменяются на соответствующие логические операции сдвиг и логическое &

Задача*

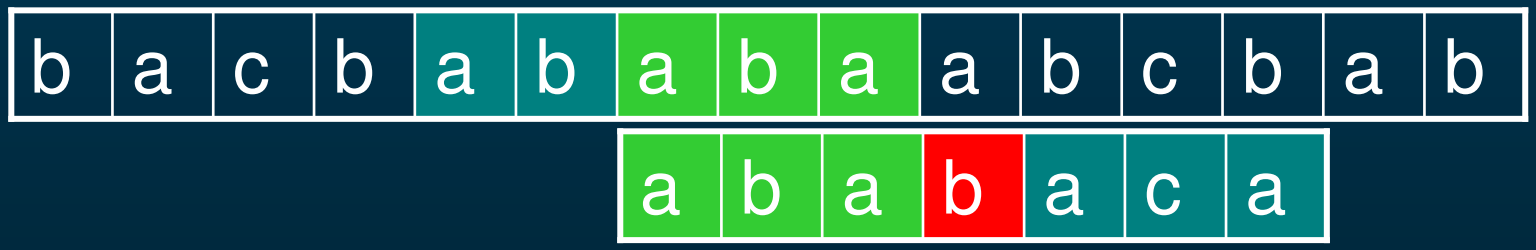
- Написать программу поиска точного вхождения в нуклеотидную последовательность

Алгоритм Кнута-Морриса-Пратта

- Если q первых символов совпало, а в позиции q+1 есть несовпадение, то из анализа только слова P можно сказать насколько имеет смысл сдвигать образец

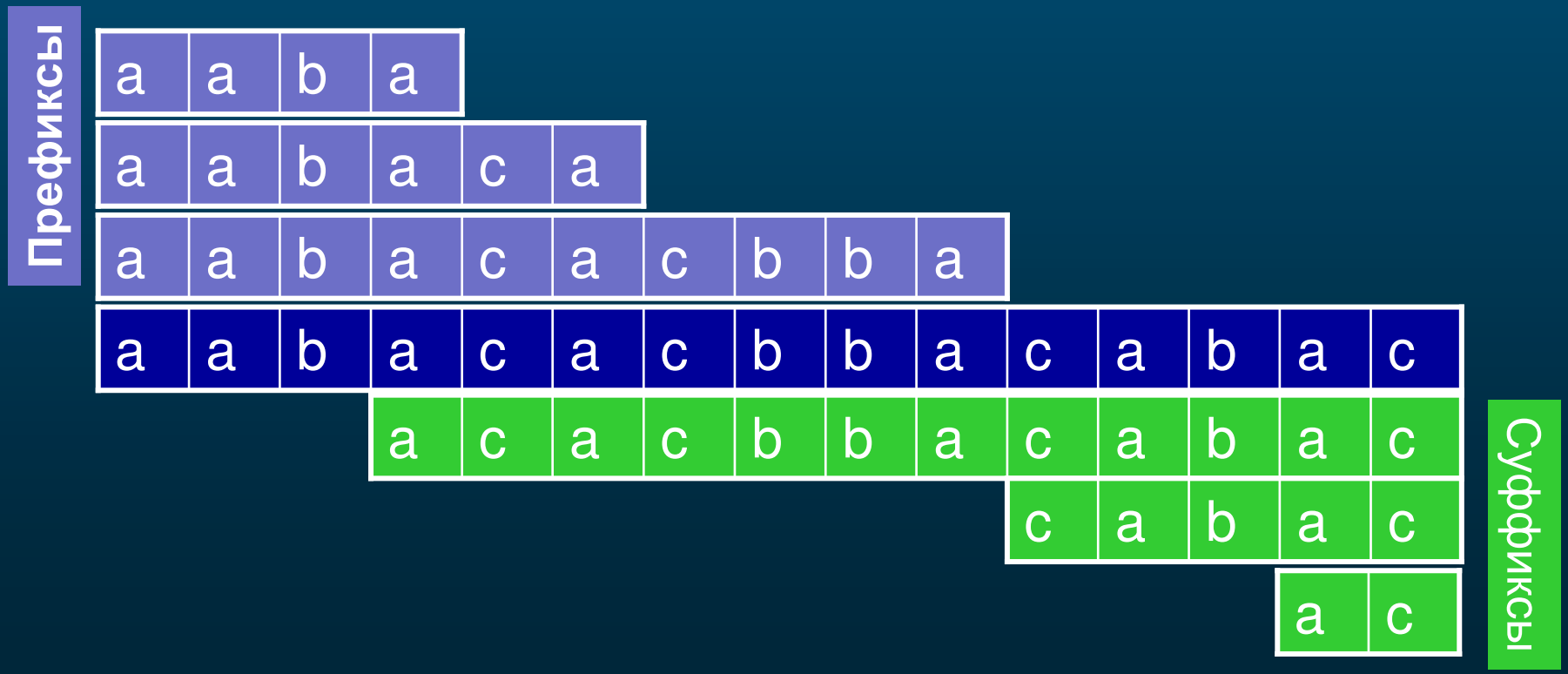


- Ясно, что сдвигать на 1 бессмысленно, а для следующей попытки правильно сдвинуть на 2



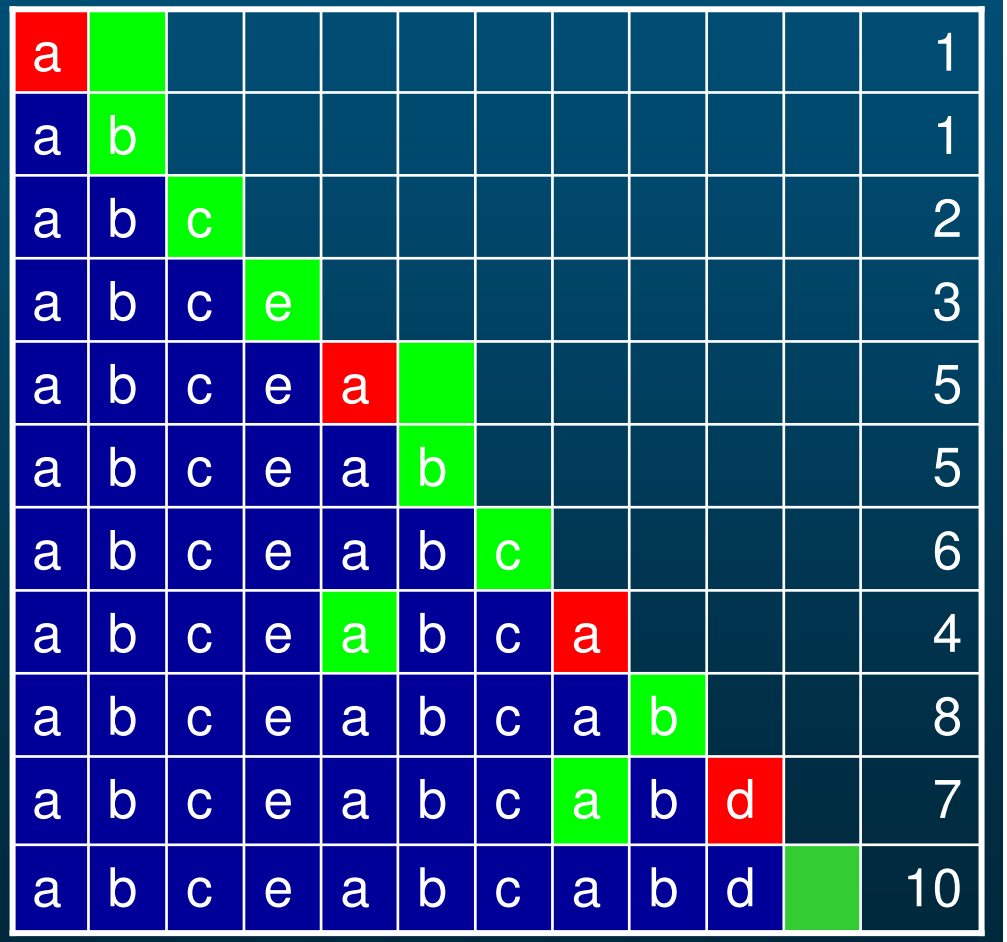
Префикс и суффикс

- Префикс – любое подслово, начинающееся с 0
- Суффикс – окончание слова



Алгоритм Кнута-Морриса-Пратта

- Попробуем вычислить величину сдвига при условии совпадения q символов.



Определения

- $sp_i(P)$ – длина наибольшего суффикса подслова $P[1..i]$ совпадающего с префиксом P
- $sp'_i(P)$ – длина наибольшего суффикса подслова $P[1..i]$ совпадающего с префиксом P, при условии $P[i+1]!=P[sp'_i+1]$

a_1	b_2	c_3	a_4	e_5	a_6	b_7	c_8	a_9	b_{10}	d_{11}
префикс P					суффикс $P[1..9]$					

сдвиг	1	2	3	3	5	5	5	5	5	8	10
sp_i	0	0	0	1	0	1	2	3	4	2	0
	a_1	b_2	c_3	a_4	e_5	a_6	b_7	c_8	a_9	b_{10}	d_{11}

Алгоритм Кнута-Морриса-Пратта

- Если несовпадение встретилось в позиции $i+1$ в слове P, то можно смело переместить образец на величину

$$i + 1 - (sp'_i + 1) = i - sp'_i$$

- Кроме того, после сдвига образца совсем не обязательно повторять сравнение первых sp_i символов.
- В любом случае происходит не более 2n сравнений (оценка в худшем)

Препроцессинг

- Сдвигая образец вдоль самого себя и проводя сравнение слева на право можно вычислить sp_i. Этот алгоритм работает за время не более $T=O(m^2)$, $m=|P|$;
- Есть алгоритм препроцессинга, работающий за время не более $T=O(m)$.

sp_i	0	0	0	1	0	1	2	3	4	2	0
	a_1	b_2	c_3	a_4	e_5	a_6	b_7	c_8	a_9	b_{10}	d_{11}
		a_1	a_1	a_1	a_1	a_1	a_1	a_1	a_1	a_1	a_1

Препроцессинг

Пусть для всех *k* нам известно *sp(k)* *(sp(k)<k !)*
тогда для *sp(k+1)* применяем рекурсию:

Препроцессинг

```
sp[1]=0;
for (k=1; k<m-1; k++){
  x=P [k];
   v=sp [k];
  while(P[v+1]!=x && v!=0)
     v=sp [v];
  if (v>0 && P[v+1]==x)
     sp[k+1]=v+1;
  else
     sp[k+1]=0;
  }
```

Время работы алгоритма

$T(m)=O(m)$

Некоторые обозначения

- x [y – x префикс y
- x] y – x суффикс y
- Если T – строка, то T_i – префикс строки длиной i
- Обозначение Σ^* - множество всех возможных слов над алфавитом Σ

Конечные автоматы

- Конечный автомат = $\{Q, q_0, A, \Sigma, \delta\}$
- Q – конечное множество (состояния)
- $q_0 \in Q$ – начальное состояние
- $A \subseteq Q$ – подмножество допускающих состояний
- Σ – конечный входной алфавит
- δ – функция: $Q \times \Sigma - Q$ функция переходов

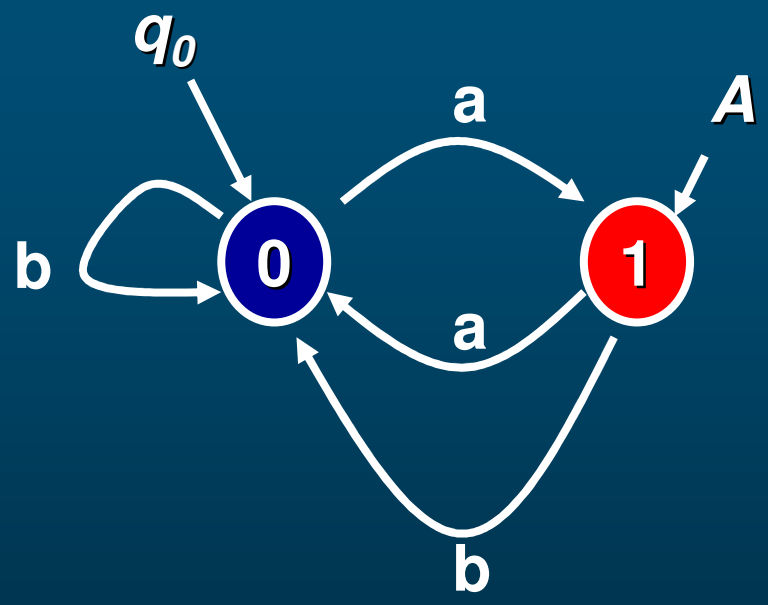
Конечные автоматы. Пример.

Функция переходов

	a	b
0	1	0
1	0	0

Вход

Состояния



abba – допускается
baaab – не допускается

Допускаются слова, в конце которых нечетное число символов a

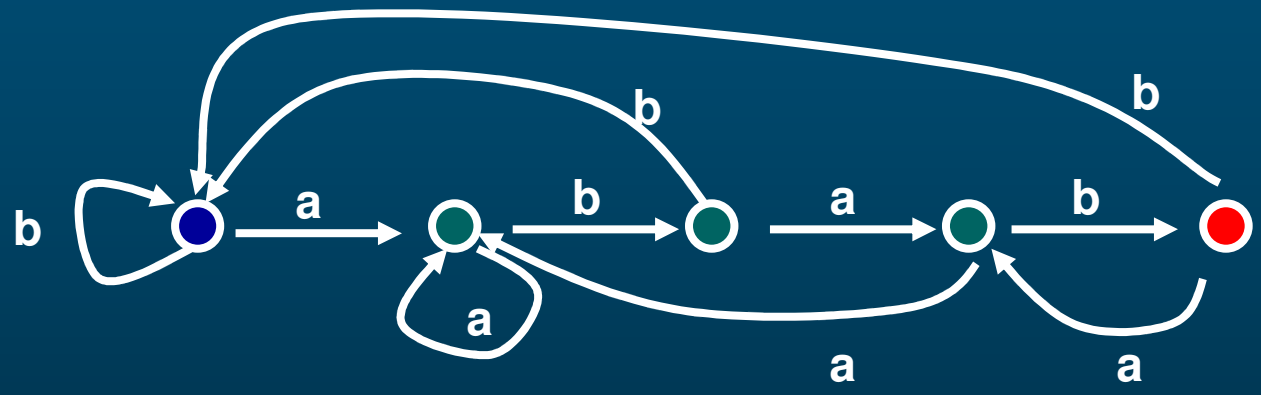
Конечные автоматы. 2.

- функция $\varphi: \Sigma^* \rightarrow Q$ функция конечного состояния: $\varphi(w)$ – конечное состояние автомата при предъявлении ему слова w
- Рекурсия :

  $\varphi(\varepsilon)=q_0$

  $\varphi(wa)=\delta(\varphi(w),a)$ для любых w, a

Конечные автоматы для поиска образца



- образец P=abab

Конечный автомат для поиска образца

- Суффикс функция для образца P
  $\sigma: \Sigma^* \rightarrow \{0,1,\ldots,m\}$
- $\sigma(x)=\max\{k: P_k \sqsupset x\}$ – длина максимального суффикса x, который совпадает с префиксом P
- Пример: P=ab,
  $\sigma(\varepsilon)=0;$
  $\sigma(ccaca)=1;$
  $\sigma(ccab)=2;$
- Свойства
  $\sigma(x)=m \leftrightarrow$ суффикс x совпадает с P
  если $x \sqsupset y$, то $\sigma(x) \le \sigma(y)$

Конечный автомат для поиска образца

- Множество состояний $Q=\{0,1,\ldots,m\}$
- Начальное состояние $q_0=0$;
- Допускающие состояния $A=\{m\}$
- Функция переходов δ:
  $\delta(q,a)= \sigma(P_q a)$

Свойства суффикс-функции

- Лемма 1. $\sigma(xa) \le \sigma(x)+1$

  Допустим $\sigma(xa) > \sigma(x)+1$. Тогда отбросим последний символ от наибольшего суффикса xa, совпадающего с префиксом P. Тогда получим суффикс строки x, который длиннее $\sigma(x)$ и является префиксом P, что противоречит определению.

- Лемма 2. Пусть $q= \sigma(x)$. Тогда $\sigma(xa)= \sigma(P_q a)$.

  Действительно, поскольку $\sigma(xa) \le \sigma(x)+1=q+1$, $\sigma(xa)$ не изменится, если отбросить начало от строки xa, оставив только последние $q+1$ символов (поскольку $q= \sigma(x)$), т.е. $\sigma(xa) = \sigma(P_q\, a)$

- Теорема: $\varphi(T_i) = \sigma(T_i)$.

  Доказательство по индукции.

  При i=0 – очевидно.

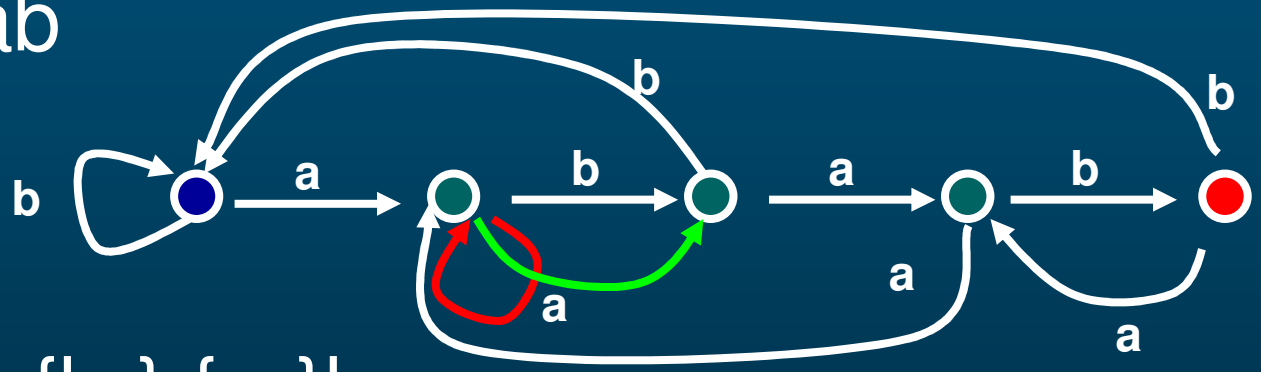
  Если верно при некотором i, то по лемме 2 будет верно и при i+1:

  $\varphi(T_{i+1})= \delta\,(\varphi(T_i),t_{i+1})= \delta(\sigma\,(T_i),t_{i+1})= \sigma(P_q t_{i+1})= \sigma(T_{i+1})$.

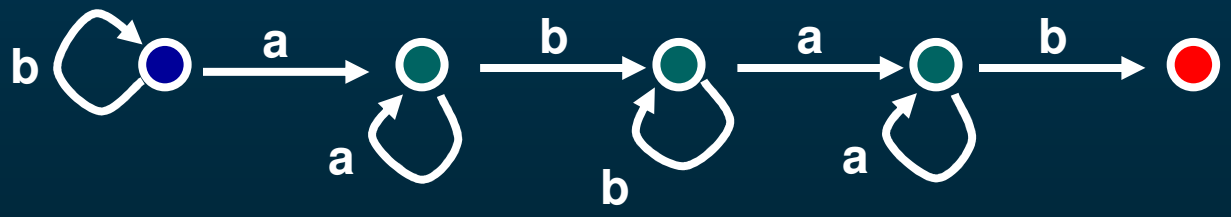
Следствие. Автомат приходит в Допускающее состояние только если суффикс T_i совпадает с образцом P

Обобщение

- Можно искать вырожденные образцы.
- P=a?ab

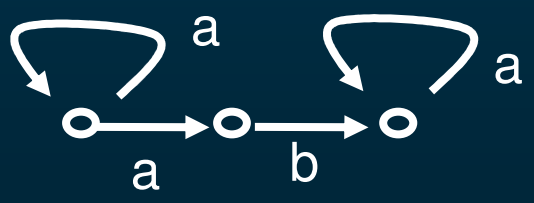


- P={a_n} {b_n} {a_n}b



Недетерминированные конечные автоматы

- Недетерминированным конечным автоматом является автомат, функция переходов которого есть отображение $\{q,a\} \rightarrow \{q^*\}$, где q^* - множество множеств возможных q.
- Функция переходов – из состояния в множество (может быть пустое) состояний
- Возможен пустой переход (без снятия символа)

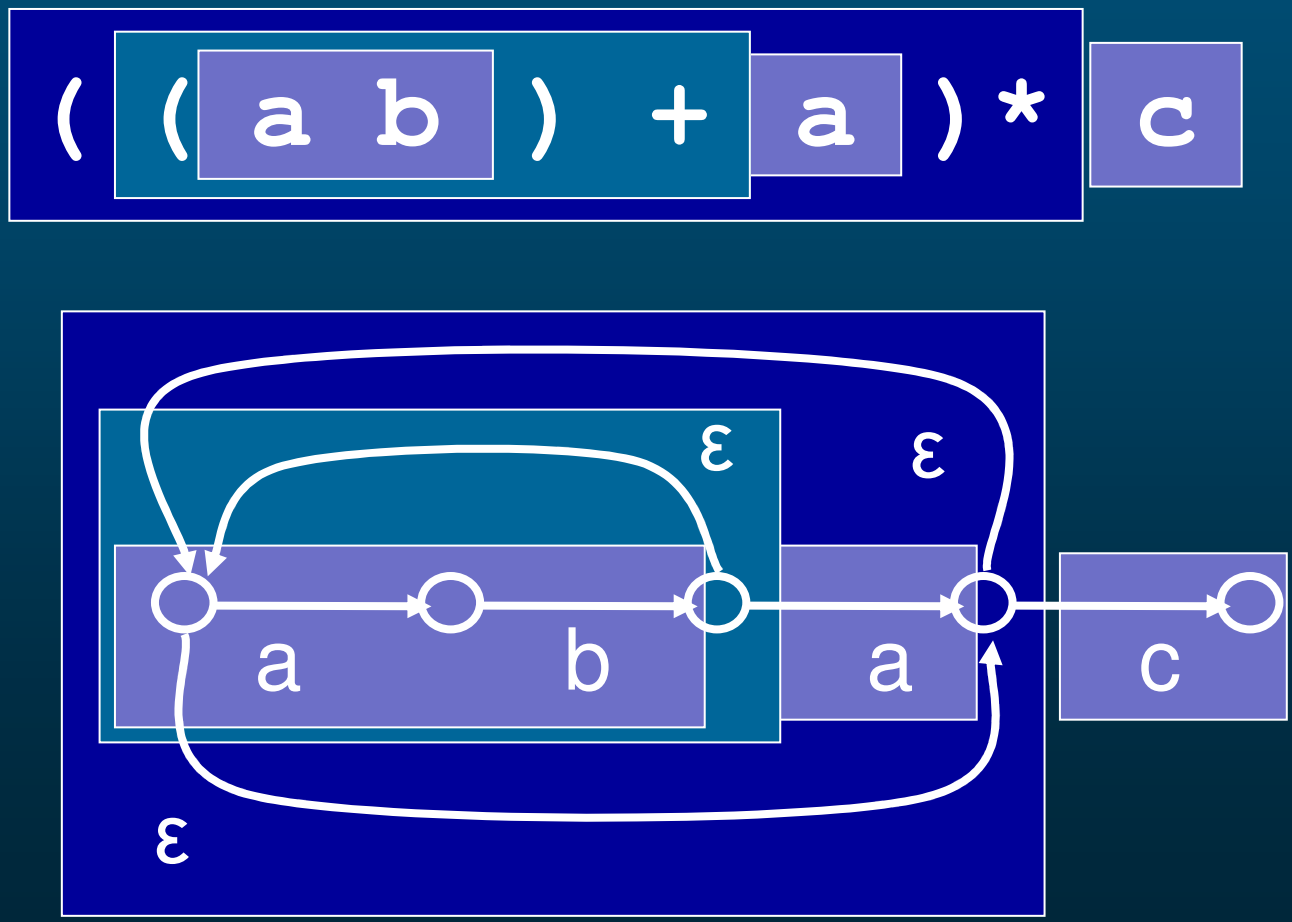


Регулярное выражение

`R::= символ | RR | (R) | R* | R+`

- просто символ
- два регулярных выражения написанные друг за другом
- регулярное выражение в скобках
- регулярное выражение с приписанным *спец символом* '*'
- регулярное выражение с приписанным *спец символом* '+'

Разбор регулярных выражений: построение автомата

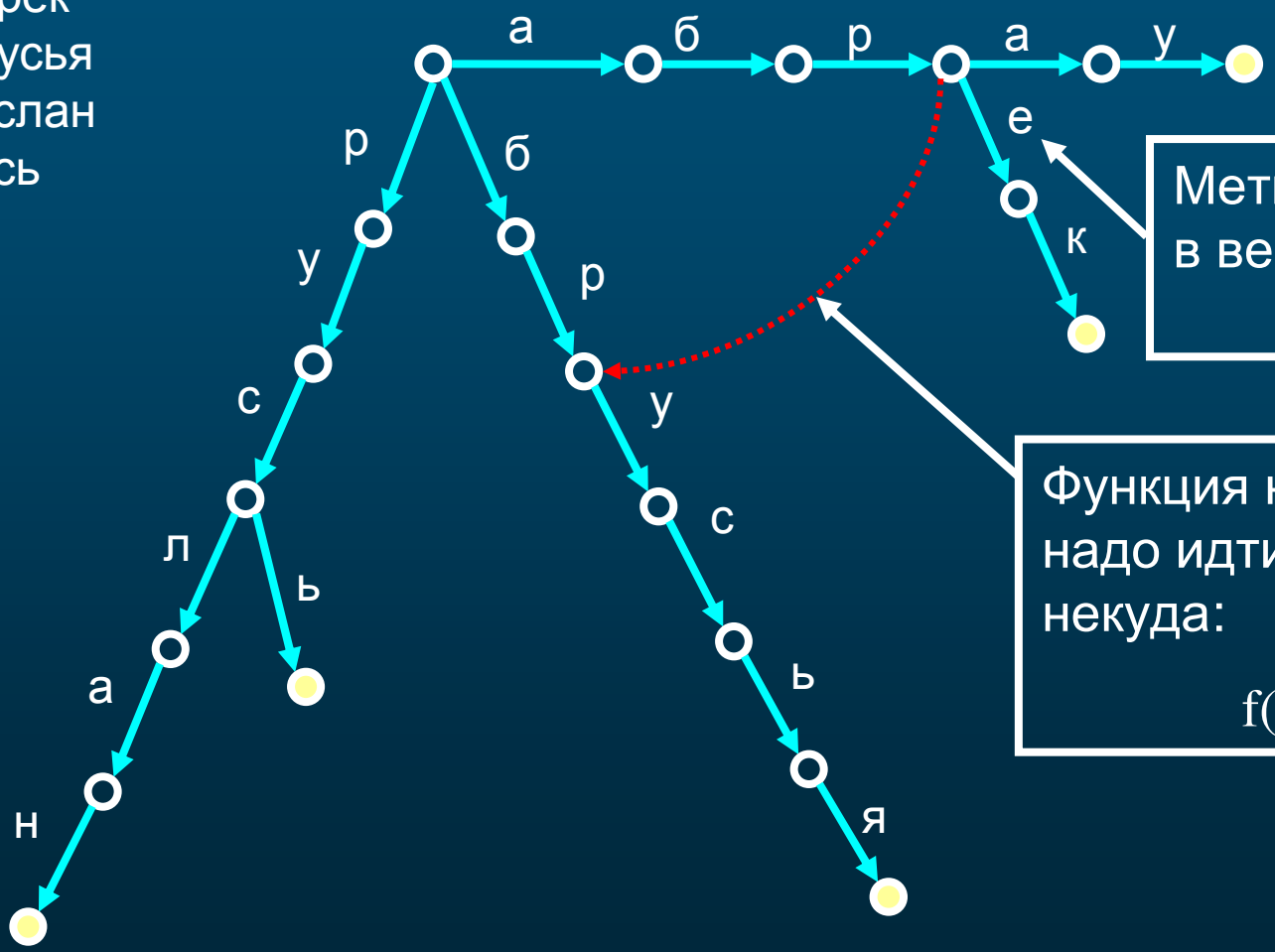


Разбор регулярных выражений: Обход автомата

- На первом этапе строим множество N_0 – q_0+множество состояний, достижимых из q_0 по ε-переходам
- При считывании очередного символа перестраиваем множество состояний
- Если множество состояний пересекается с множеством допустимых состояний, но репортируем находку

Обобщение на множество ключей

абрау
абрек
брусья
руслан
русь



Метка ребра, идущего в вершину:

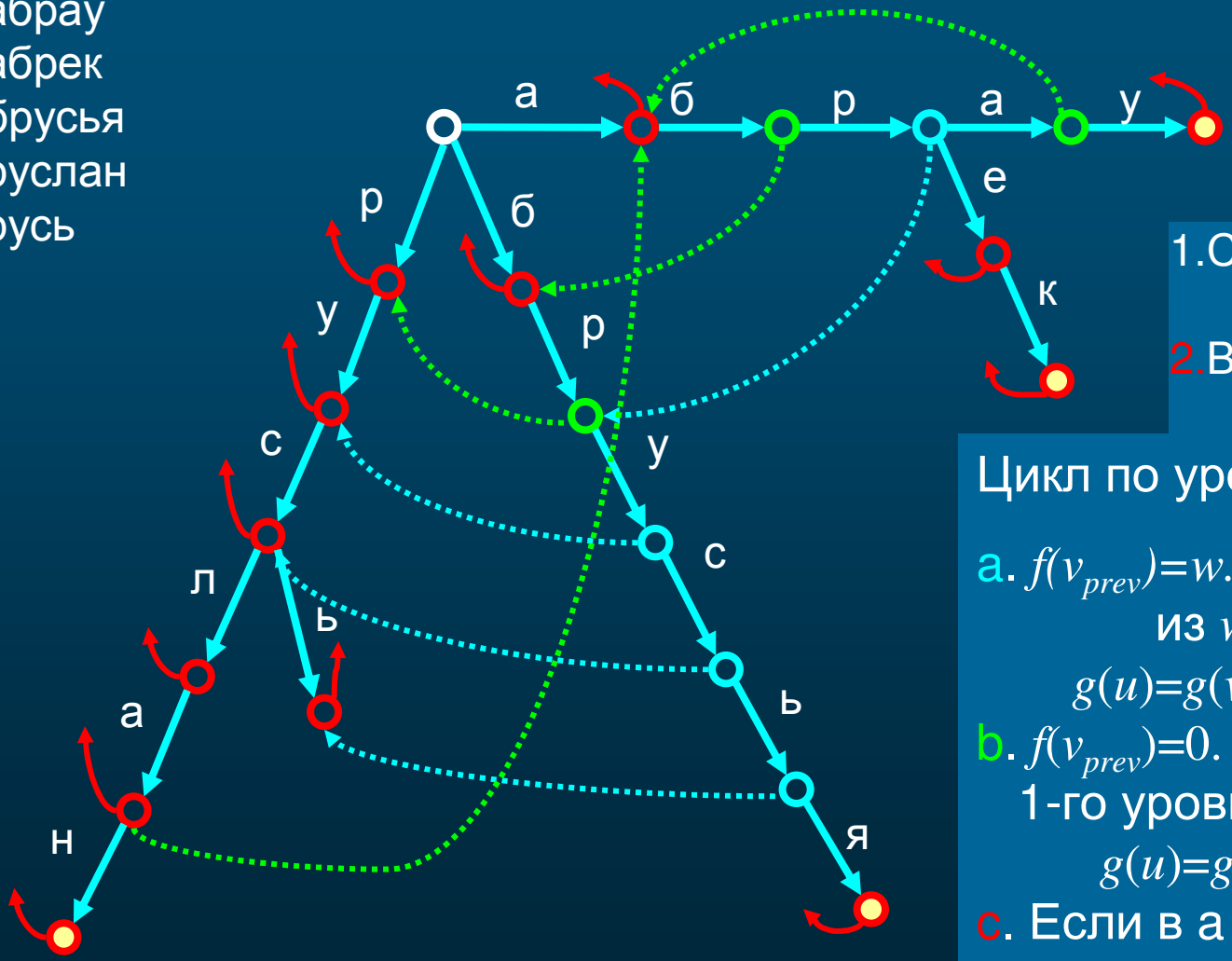
$g(v): V \rightarrow \Sigma$

Функция неудач – куда надо идти, если идти некуда:

$f(v): V \rightarrow V$

Автомат Ахо-Корасик

абрау
абрек
брусья
руслан
русь



1.Сортируем вершины по уровням
2.Вершины 1-го уровня – в корень

Цикл по уровням и вершинам:

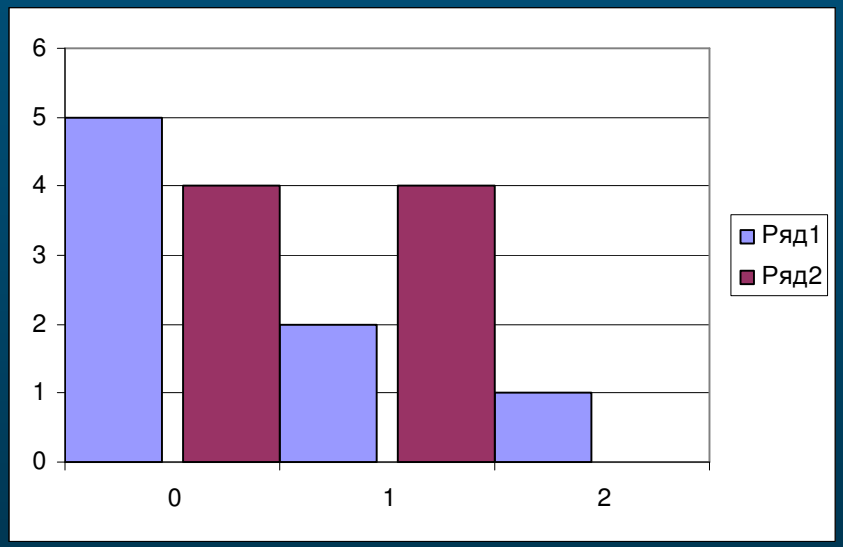
a. $f(v_{prev})=w$. ищем вершину u из w_{next}:
$g(u)=g(v); f(v)=u;$

b. $f(v_{prev})=0$. ищем вершину 1-го уровня u:
$g(u)=g(v); f(v)=u;$

c. Если в a и b ничего не нашли,
$f(v)=0;$

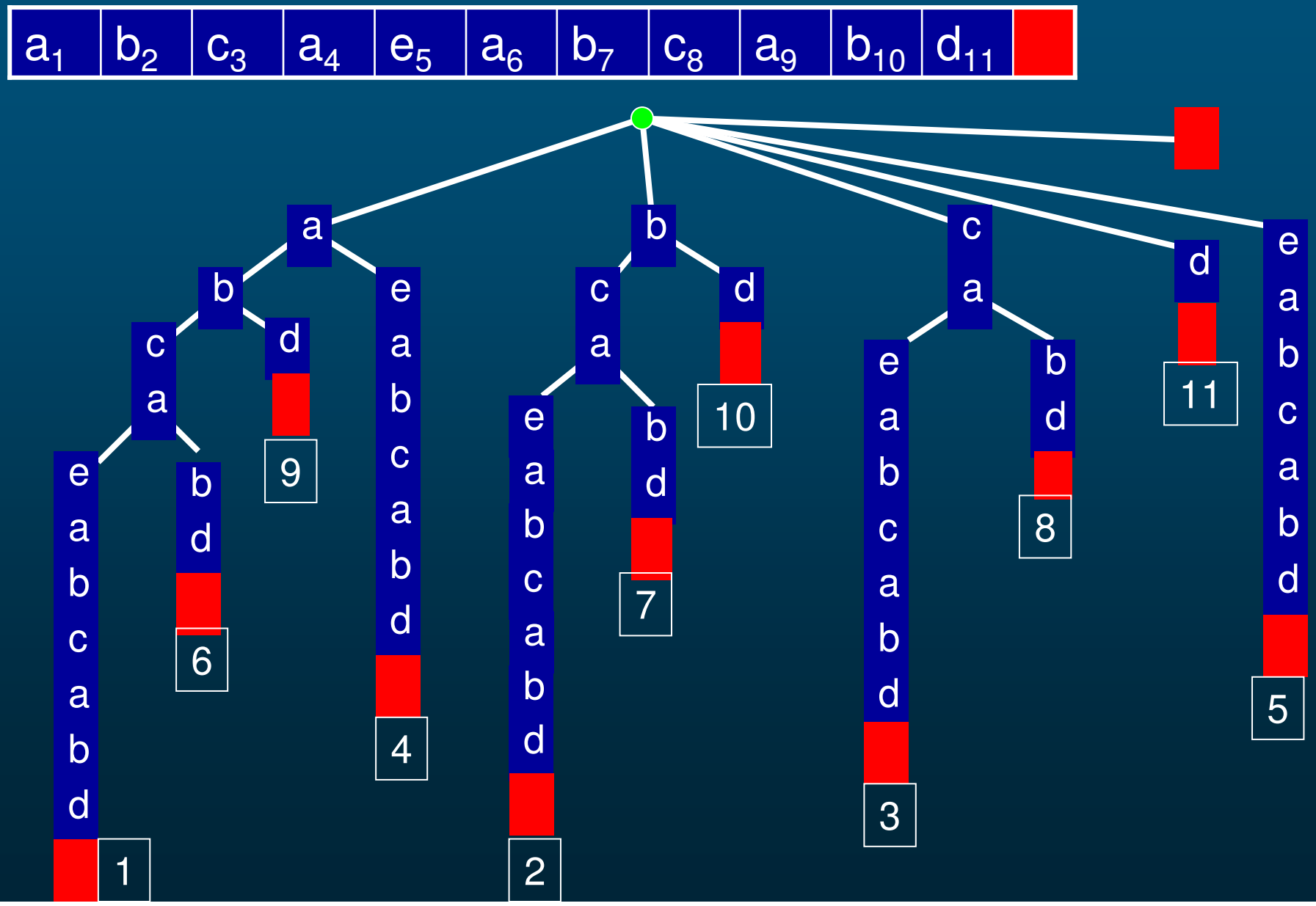
Статистика слов

	AA	AB
AAA	2	0
AAB	1	1
ABA	0	1
ABB	0	1
BAA	1	0
BAB	0	1
BBA	0	0
BBB	0	0
	4	4



E(AA)=0.5 D(AA)=0.5
E(AB)=0.5 D(AB)=0.25

Суффиксные деревья



Свойства суффиксного дерева

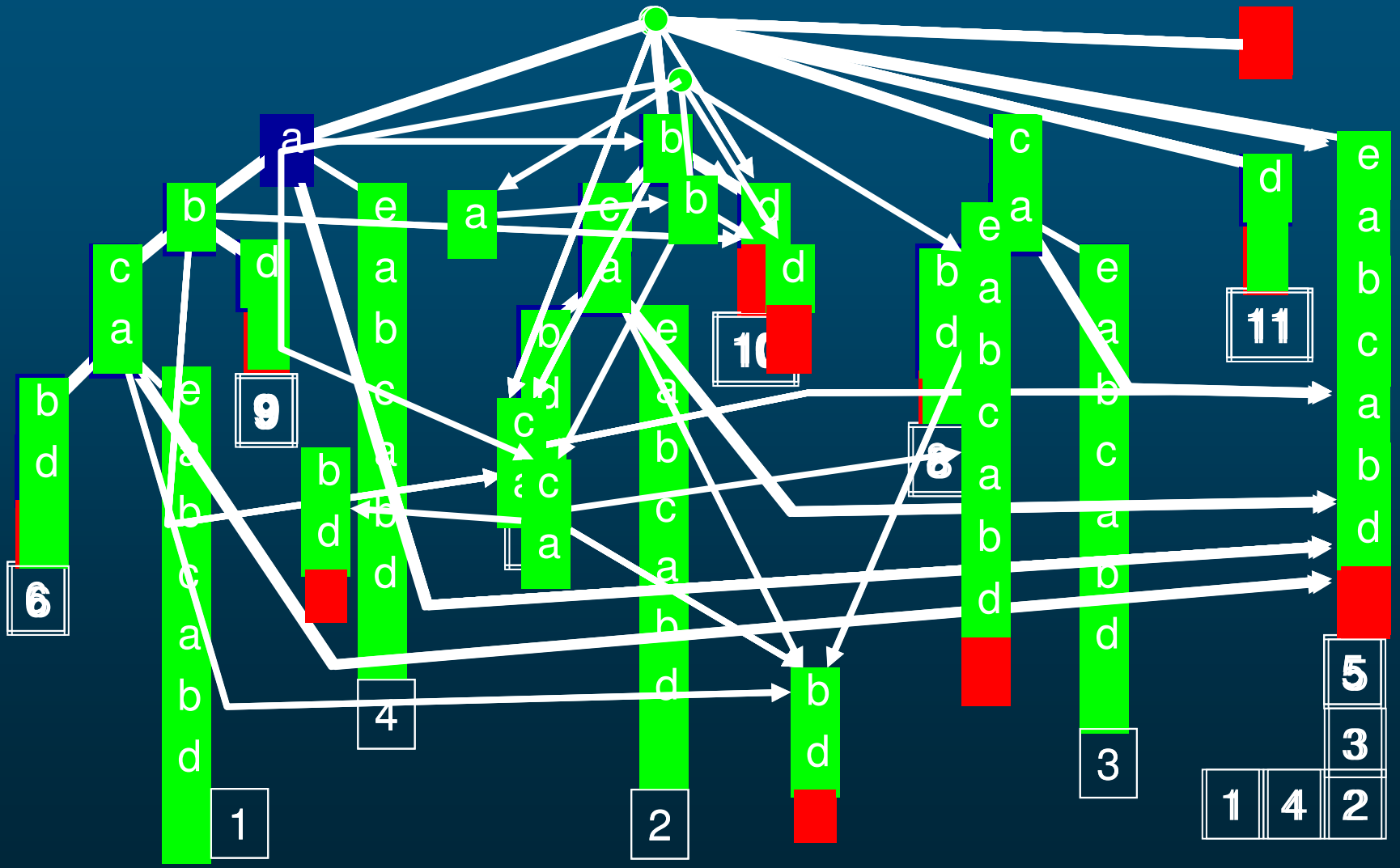
- Листьев столько же сколько и позиций
- На ребрах определены строки (метка ребра)
- Каждый путь от корня до листа – суффикс
- Можно построить такое дерево за время $T=O(n)$
- На листьях указаны позиции начал соответствующих суффиксов

Задача*
Придумать алгоритм построения суффиксного дерева (любой)

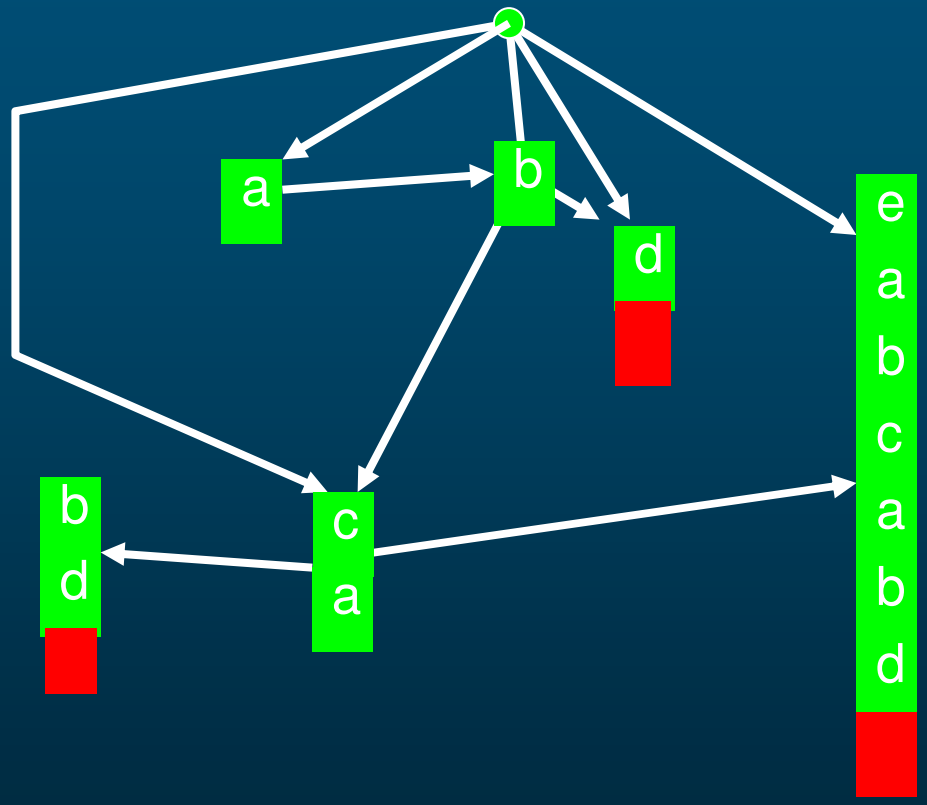
Зачем этот наворот?

- Если для строки однажды построить такое дерево, то им можно пользоваться для поиска образца за время $T=O(||P||)$ – пропорционально длине искомого слова
- Можно решать задачу поиска общего подслова для двух слов. Для этого слова объединяют и строят суффиксное дерево для объединенного слова. Вершины, имеющие пометки адресов к обоим словам определят общие подслова.
- Повтор – путь между двумя не-листьями

Преобразованиее в ориентированный граф



Конечный автомат



Суффиксный массив

11	i
8	ippi
5	issippi
2	ississippi
1	mississippi
10	pi
9	ppi
7	sippi
4	sissippi
6	ssippi
3	ssissippi

Храним только позиции (слово-то у нас есть)

Когда ищем слово, то делаем бинарный поиск (сколько времени надо для поиска образца длиной m?)

Построение возможно за линейное время

Сложность текста

- Попробуйте описать следующий текст:
  - `aaaaaaaaaaaattttttttatatatatatatatatatatat`
- 12 a, 8 t ,10 at
- А такой:
  - `gcttccatgacgcagaagtttgcttgtttatcaaaaactg`
- Гораздо длиннее (проще его просто прочитать)

Пример

```
a,(1,1),(1,2),(1,4),(1,4),t,(13,1),(13,2),(13,4)
```

```
aaaaaaaaaaaaattttttttt
```

- 9 команд, 20 букв
- Самые сложные тексты – случайные

Описание текста

Пусть Z = {0,1}* - множество всех слов на 0,1

Определим $E \subset Z \times Z$ так, что

1. $(x, y_1) \in E, \quad (x, y_2) \in E = \; y_1 = y_2$
2. $\exists$ программа $\Pi : \Pi(x) = y, \; если \; (x, y) \in E$

Это называется системой описания текстов. На самом деле похоже на разархивирование

Замечание. Для одного текста y существует множество описаний x.

Сложность текста относительно программы:

$$KS_{\Pi}(y) = \min\{\|x\| \mid \Pi(x) = y\}$$

Сложность

Системы описания:

Π_1 не хуже Π_2, если

$$\exists C : \forall x \, KS_{\Pi_1}(x) \leq KS_{\Pi_2}(x) + C$$

Теорема если есть две системы описания, то можно построить систему, которая не хуже каждой.

Доказательство: в начале описания поставим признак: какую программу надо использовать (тем самым увеличим длину описания), а за тем напишем описание в соответствии с выбранной программой

Сложность по Колмогорову

Все мыслимые программы можно пронумеровать и построить универсальную программу, которая считывает сначала номер, а потом исполняет соответствующую программу. Это называется ***Универсальной программой.***

Сложность по Колмогорову: длина самого короткого описания текста (относительно ***фиксированной*** *универсальной программы*). Сложность не вычислима.

Сложность и случайность

- Количество слов W сложности K:

$$2^{K-C} \leq W \leq 2^{K+1}$$

- Случайный текст – текст, любое описание которого не короче самого текста.

Пример программы сжатия

Пример программы.

- Элементарные команды описания:
  - Добавить заданный символ
  - Скопировать фрагмент текста из уже сгенерированной последовательности

Сжатие информации по Лемпелю-Зиву

- Для сжатия информации строится суффиксное дерево (время $O(n)$).
- Пусть уже упаковано i-1 символов. Нам надо вычислить следующую команду (s_i, l_i). Берем суффикс $S(i..n)$ и пропускаем его через дерево $T(S(1,i\text{-}1))$(как при поиске слов) и либо находим максимальное слово, которое является префиксом $S(i..n)$, и тогда ставим команду скопировать это слово. В противном случае генерируем новую букву.
- Суффиксное дерево можно генерировать одновременно с поиском.

Графы

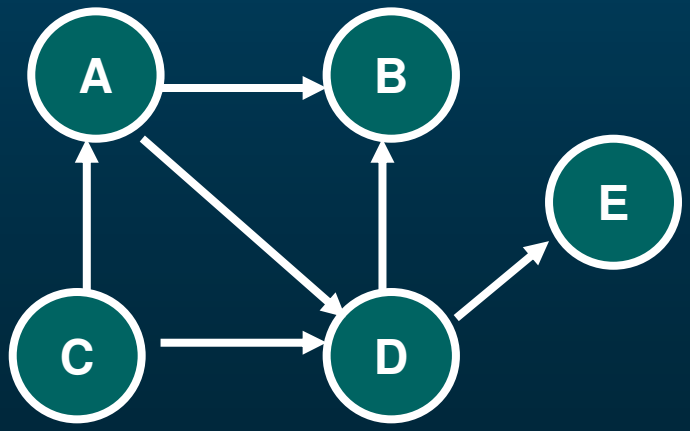
- Формальное определение: (*ориентированным*) *Графом* на множестве вершин M называется отображение

$$E=\{M\bullet M\Rightarrow\{\textbf{\textit{true, false}}\}\}$$

- Пары вершин v1,v2, для которых E(v1,v2)==true называются *ребрами*
- Если отображение коммутативно (E(v1,v1) == E(v2,v1)), то граф называется *неориентированным*

Представление графов

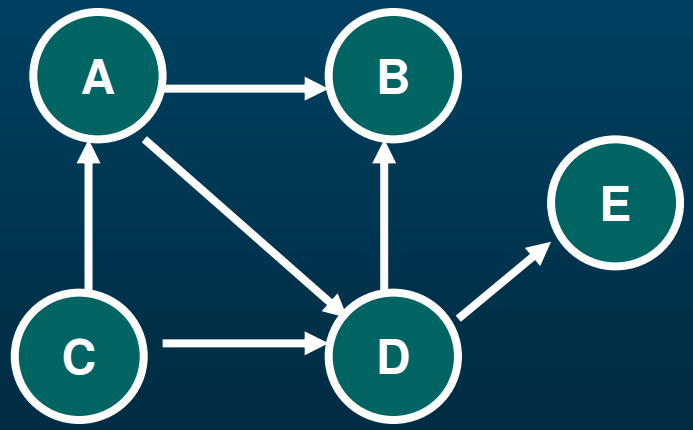
- Матрица смежности: Матрица, отображающая определение графа. Для неориентированного графа $M^T=M$
- Списки ребер
- Списки смежных вершин



Матрица Смежности					
	A	B	C	D	E
A	0	1	0	1	0
B	0	0	0	0	0
C	1	0	0	1	0
D	0	1	0	0	1
E	0	0	0	0	0

Представление графов

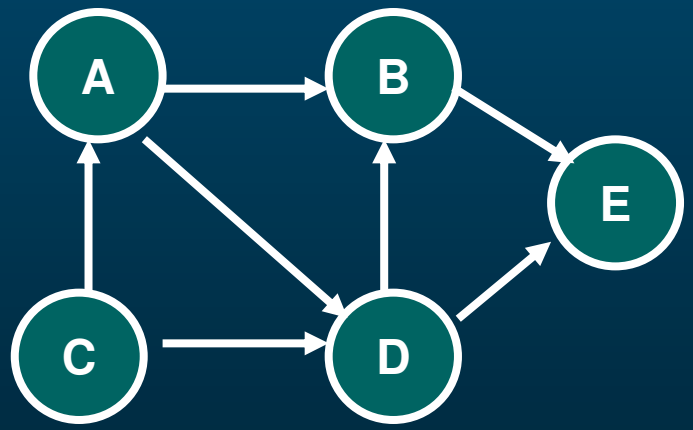
- Списки ребер



A	B
A	D
C	A
C	D
D	B
D	E

Представление графов

- В каждой вершине записываем списки смежных вершин



A:	B	D	/
B:	E	/	
C:	A	D	/
D:	B	E	/
E:	/		

Поиск в ширину

Алгоритм обхода графа:

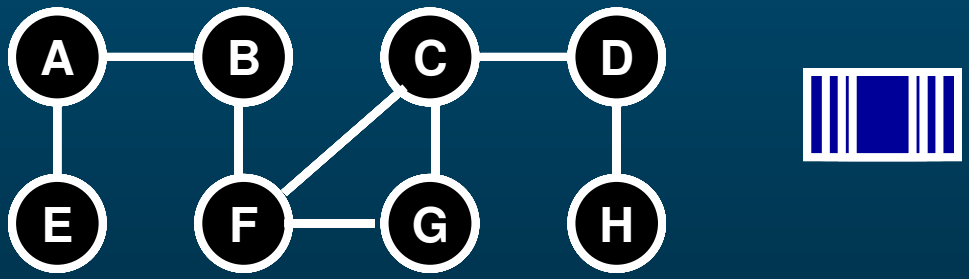
Цвета:

- белый – вершина, которую еще не видели
- серый – вершина, которая находится в стадии обработки
- черный – вершина, для которой обработка завершена

Вначале – все вершины - белые

1. Берем произвольную вершину, помещаем в очередь и красим в серый цвет.
2. Изымаем очередную вершину из очереди (красим в черный цвет) и все смежные не пройденные (белые) вершины помещаем в конец очереди и красим в серый цвет
3. Если очередь не пуста, переходим к 2.
4. Если остались не присмотренные вершины, то берем из них произвольную, помещаем в очередь и переходим к 2

Поиск в ширину



Алгоритм

```
wideSearch(){
   for( all v from V){
    if(v.color==white) wideSearch(v);
   }
}
wideSearch(v){
   queue Q;
   Q.put(v); v.color=gray;
   for(; Q is not empty;){
     u=Q.get();
     for( all w сосед u){
        if(w.color=white){
          q.put(w); q.color=gray;
     }
     u.color=black;
   }
}
```

Определение расстояния между вершинами

Задача: найти расстояние от вершины *s* до вершины *v*.

Делаем поиск в ширину, начиная с *s* до тех пор, пока не встретим *v*.

Вопросы:

1. Если есть путь от s до v, то найдем ли этот путь (т.е дойдем ли вообще до v?
2. Будет ли этот путь кратчайшим?
3. Доказательство по индукции и от противного: если бы это был не кратчайший путь, то вершина на кратчайшем пути встретилась бы раньше при просмотре

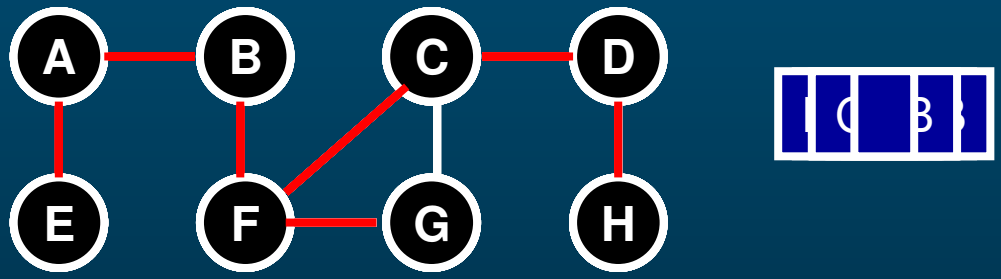
Задача: доказать, что если путь существует, то при поиске в ширину мы дойдем от s до v

Поиск в глубину

Алгоритм обхода графа:

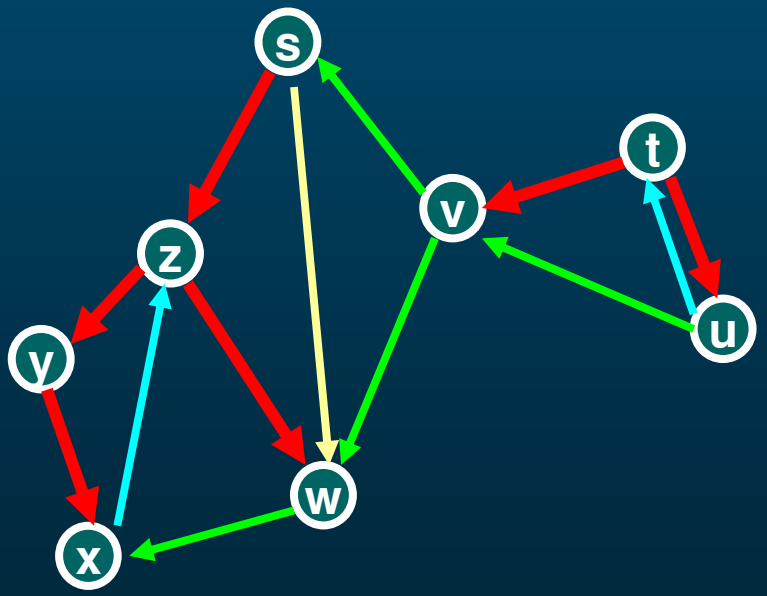
1. Берем произвольную вершину и помещаем в **Стек** (красим в серый цвет)
2. Если есть белые смежные вершины, то Берем произвольного соседа и помещаем в стек (красим в серый) и переходим к 2.
3. Если нет смежных вершин, то берем вершину из стека и заканчиваем ее обработку (красим в черный) и переходим к 2.
4. Если остались не присмотренные вершины, то берем из них произвольную, помещаем в стек и переходим к 2

Поиск в глубину



Свойства поиска в глубину

- В результате поиска в глубину получается дерево (лес). Каждая вершина имеет время окончания обработки.



Типы ребер графа uv

Ребра деревьев

Прямые: v, u принадлежат дереву и t(u) > t(v)

Обратные: v, u принадлежат дереву и t(u) < t(v)

Перекрестные - остальные

Построение покрывающего дерева

- Строим лес поиска в глубину.
- Если есть перекрестное ребро из листа одного дерева в корень другого дерева, то включаем это ребро
- Когда все такие ребра исчерпаны, получим покрывающее дерево.

Связные компоненты

- ***Пример***. Вершины графа – белки из базы данных. Ребро проводится если e-value лучше 1.e-6.
- Тогда связную компоненту можно рассматривать как семейство белков (достаточно условно)

Поиск связной компоненты

- На неориентированном графе делается обходом в ширину.
- На ориентированном графе ко всем ребрам надо добавить перекрестные ребра

Поиск циклов в графе

- В ориентированном графе – ищем покрывающее дерево. Все обратные ребра порождают цикл.
- В неориентированном графе – ищем покрывающее дерево. Все ребра, не принадлежащие дереву, порождают циклы.

Эйлеров цикл

- *Задача Эйлера:* найти на графе цикл, проходящий через все ребра, причем по каждому ребру один раз.
- В неориентированном графе эйлеров цикл существует тогда и только тогда, когда индексы всех вершин четные.
- В ориентированном графе эйлеров цикл существует тогда и только тогда, когда индексы всех вершин равны 0.

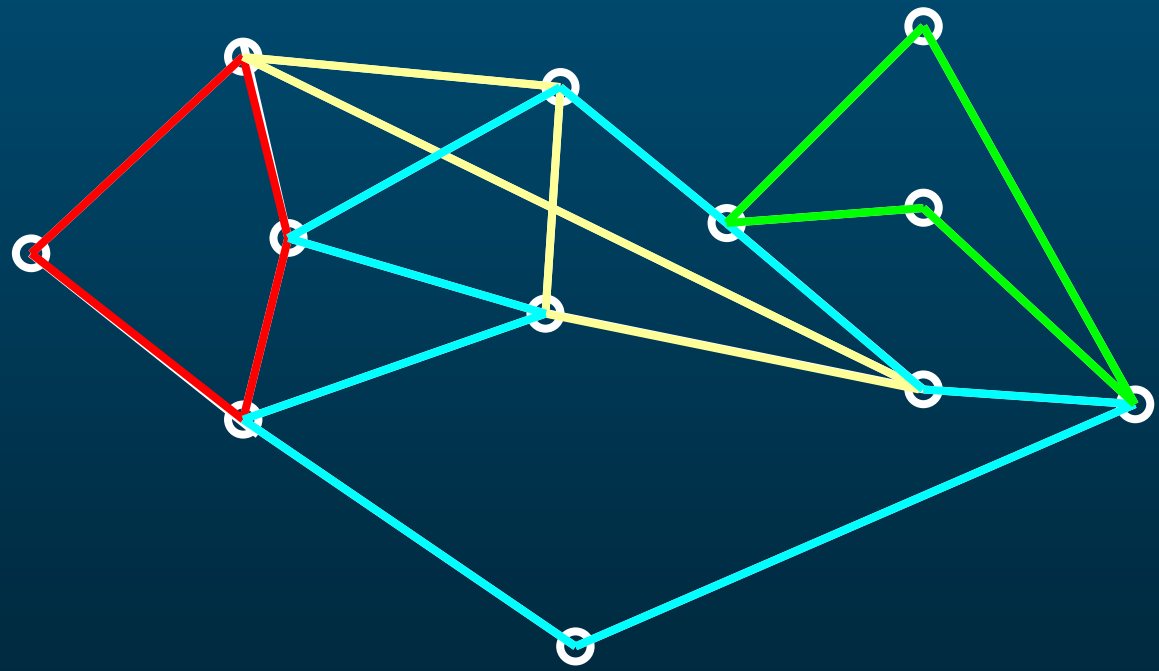
Эйлеров путь

- Существует только если две вершины нечетные
- Проведем ребро между нечетными вершинам и и сведем задачу к поиску цикла.

Эйлеров цикл

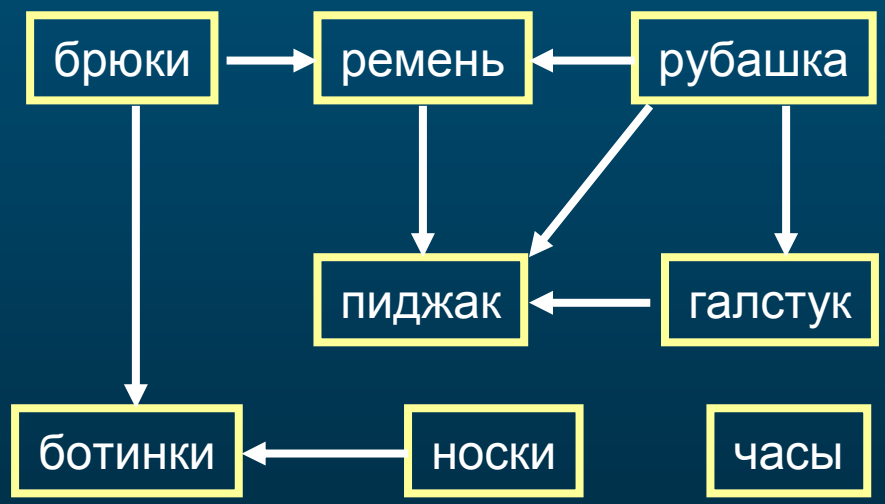
1. Помечаем все ребра, как не пройденные
2. Ищем цикл в графе по не пройденным ребрам. Записываем последовательность ребер в связный список. Он проходит через все вершины
3. Если не все ребра пройдены, то находим вершину, из которой выходит не пройденное ребро. (Доказать, что это существует)
4. Находим цикл, проходящий через заданную вершину.
5. Вставляем в список новый цикл. Переходим к п. 3.

Построение Эйлерового цикла



Топологическая сортировка

- Брюки
- Рубашка
- Галстук
- Ремень
- Носки
- Ботинки
- Пиджак
- Часы



Топологическая сортировка

- Задача: расположить вершины ориентированного графа на горизонтальной оси так, чтобы ребра шли слева направо.
- Если в графе есть циклы, то задача не имеет решения.

Алгоритм

- Делаем поиск в глубину. По окончании обработки вершины пишем ее в начало списка.

рубашка	носки	часы	галстук	брюки	ботинки	ремень	пиджак

Оптимальный путь в графе

Задача:

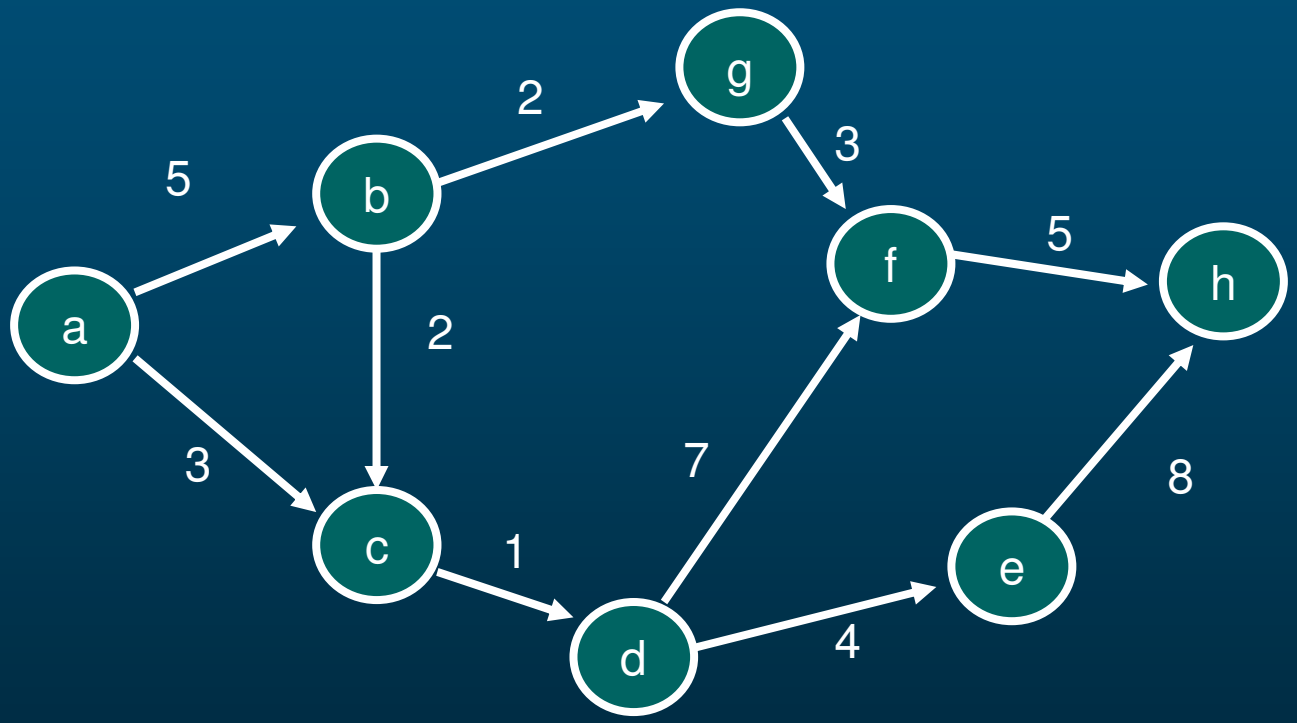
- Дан связный ориентированный граф без циклов. На каждом ребре графа написано неотрицательное число.
- Найти на графе путь между двумя вершинами, имеющий максимальный вес

Оптимальный путь в графе

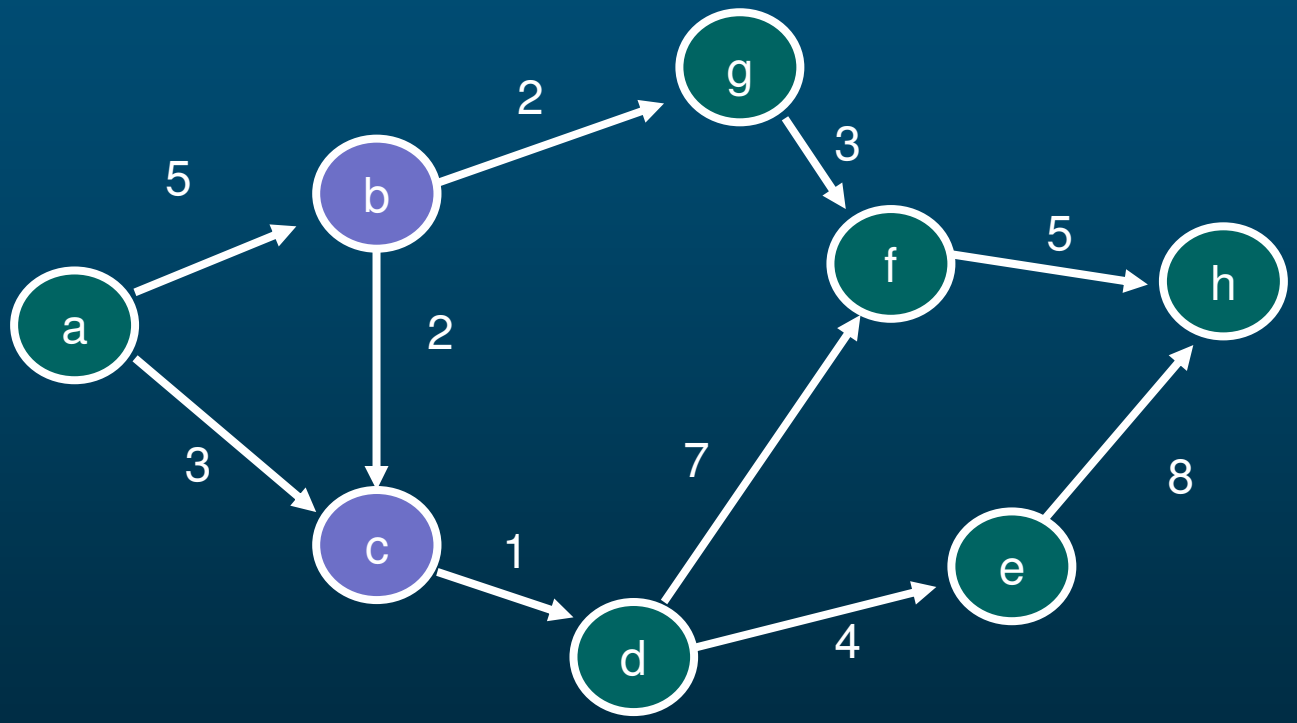
Пример.

- В результате секвенирования большого фрагмента ДНК (например, генома) получен ряд последовательностей
- Строим граф. Вершина – прочитанная последовательность. Ребро проводится, если суффикс первой последовательности равен префиксу второй. Вес равен длине общей части
- Путь с наибольшим весом дает последовательность генома

Оптимальный путь в графе

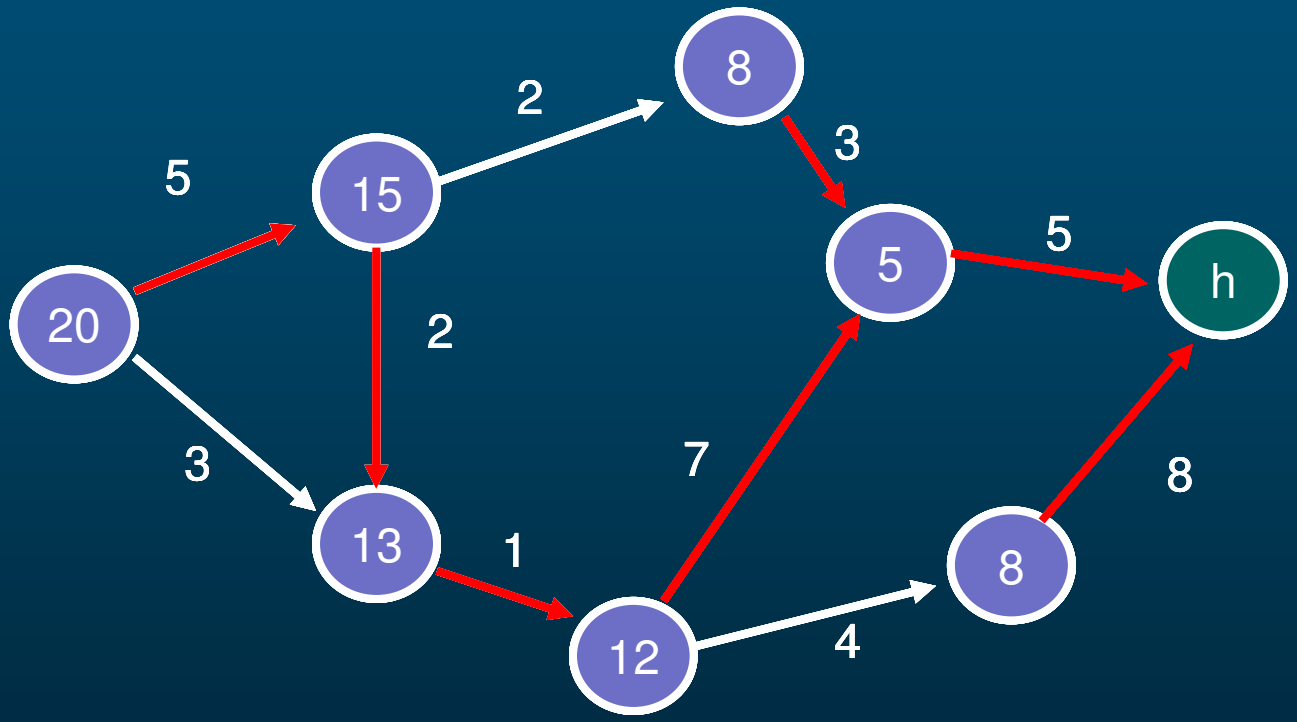


Динамическое программирование



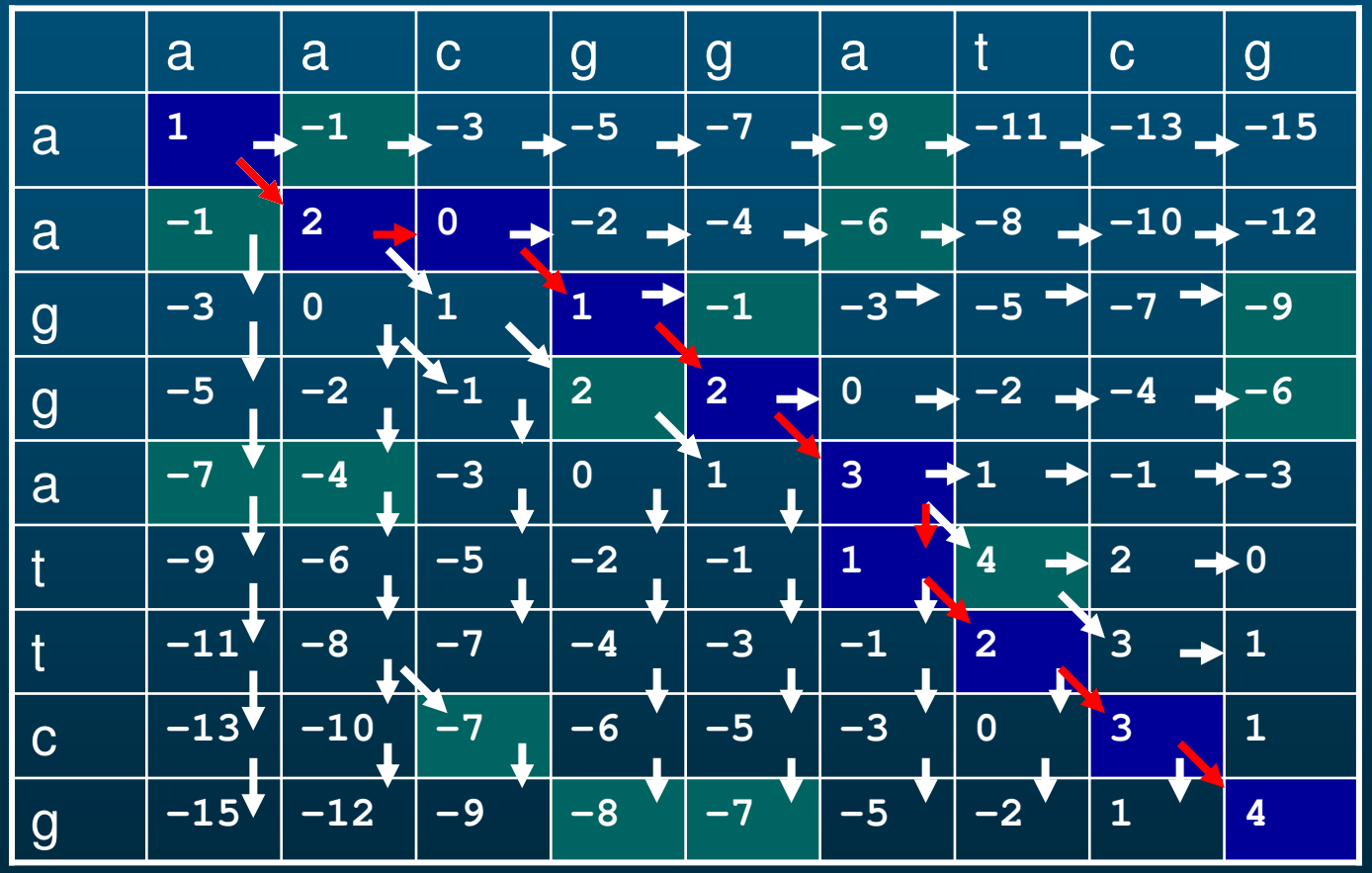
$$W(a,h) = \mathbf{max}\{\ 5 + W(b,h),\ 3 + W(c,h)\ \}$$

Динамическое программирование



$$W(V)=\max_{prev}\{ W(V_{prev})+W(E(V->V_{prev}) \}$$

Выравнивание последовательностей



Del=-2; Match=1; Mismatch=-1

Полукольцо

- Множество ***M*** с двумя операциями: (+), (•)
- Операции обладают свойствами:
  - $(a + b) + c = a + (b + c)$
  - $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
  - $a \cdot (b + c) = a \cdot b + a \cdot c$
  - 0 : $a + 0 = 0 + a = a$;
  - 1: $a \cdot 1 = 1 \cdot a = a$
  - $a \cdot 0 = 0$

Примеры

Множество	Опер. (+)	Опер. (•)
{ true, false }	|	&
натуральные числа	+	•
действительные числа, +∞, -∞	max	+

Задачи

1. Показать, что приведенные примеры действительно полукольца
2. Придумать еще примеры полуколец

Динамическое программирование на графах

- Когда мы искали оптимальный путь на графе, мы использовали две операции:

(max , +)

причем использовалось то, что эти операции образуют полукольцо (ассоциативность и дистрибутивность)

- Можно обобщить рекурсию динамического программирования на:

$W(V)=W(V_1)\ (\bullet)\ W(E_1)\ (+)\ \ W(V_2)\ (\bullet)\ W(E_2)\ \ (+)\ \ldots$

- Если мы заменим эти операции на **(+,•)**, то сможем подсчитать количество путей по графу между указанными точками

Подсчет числа путей в графе

- На каждом ребре записываем 1
- Строим рекурсию

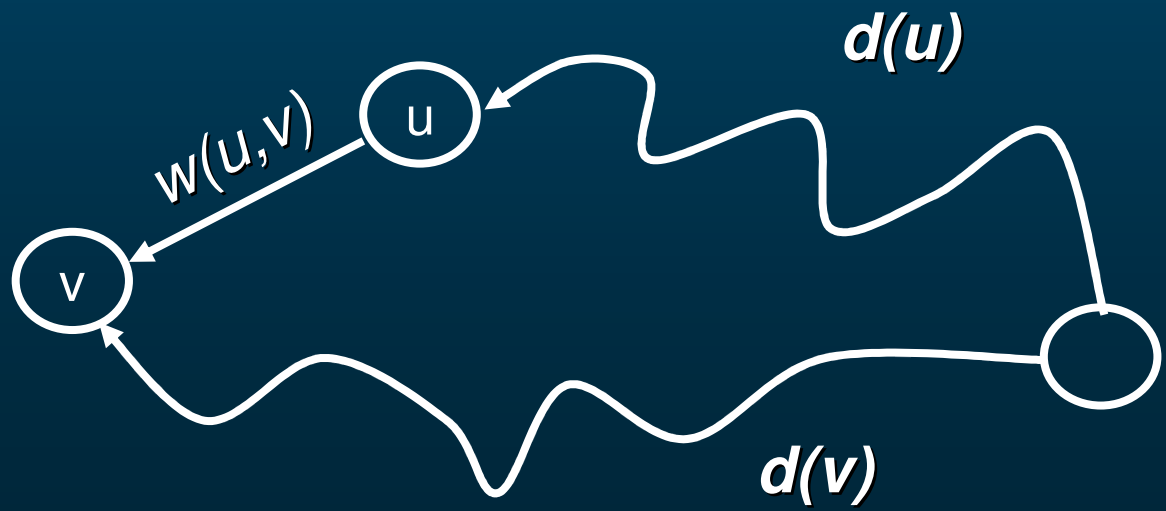
  $W(V)=W(V_1) \cdot W(E_1) + W(V_2) \cdot W(E_2) + \ldots$
- В результате в каждой вершине определяем количество путей, ведущих из нее в финальную вершину (почему?)

Алгоритм Дейкстры

В каждой вершине пишем текущий вес пути до конца.

Релаксация для вершины *v*, соединенной ребром веса *w*(u,v) с вершиной *u:*

```
If(d(v) > d(u)+w(u,v)){
        d(v)=d(u)+w(u,v)
        π(v)=u
        }
```



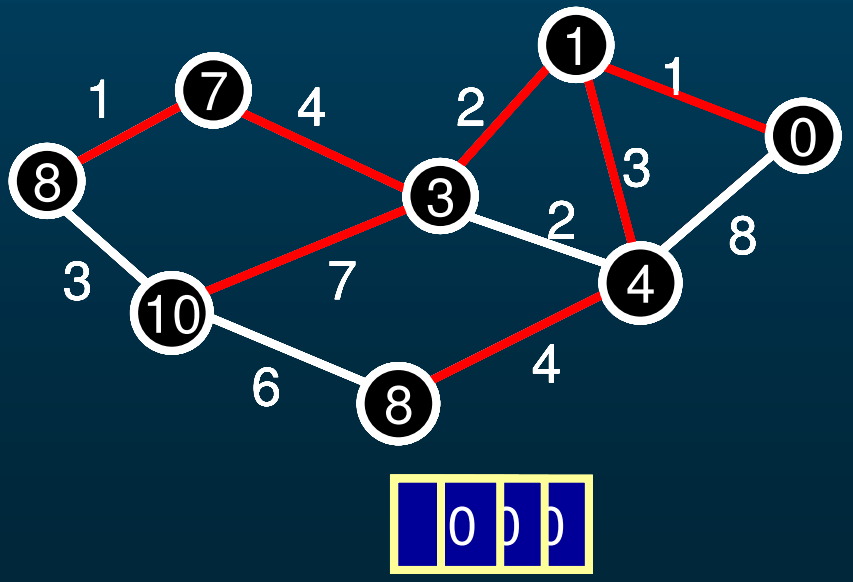
Алгоритм Дейкстры

Алгоритм применим только если веса положительные !

Структура данных *– очередь с приоритетами:*
первым выходит тот, кто имеет наименьший вес, но если несколько наименьших весов, то выходит тот, кто раньше пришел. Реализуется, например, в виде двоичного дерева.
Те, кто окончательно обработан (вышел из очереди) – черные, кто стоит в очереди – красные, кто еще не встал в очередь – голубые.

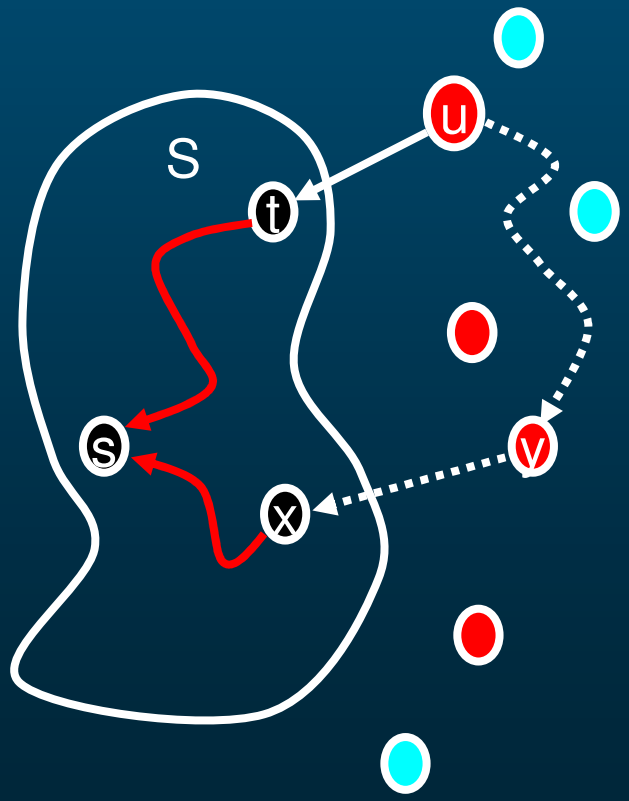
Алгоритм

1. Все вершины необработанны (голубые, вес ∞), кроме последней (красная, стоит в очереди, вес 0),
2. Снимаем вершину из очереди и для всех соседей делаем Relax, и, если они не красные, то помещаем в очередь.



Правильность алгоритма Дейкстры

На каждом этапе красные вершины релаксированы относительно черных!



Пусть S – множество черных вершин, и для них $d=\delta$. Пусть u – красная вершина с наименьшим d:

$u=argmin_{z=красные}\ d(z)$

Тогда $d(u)=\delta(u)$.

1. Никакое ребро в черную вершину не лучше (поскольку красные – релаксированы).
2. Допустим есть путь из u в s, лучше, чем (u→t→s). Тогда этот путь пройдет через красную вершину y (u→y→s).
   Но:
   w(u→y→s) = w(u→y)+w(y→x→s)
   w(y→x→s)=d(y) > d(u)=w(u→t→s) *(u=argmin!)*
   w(u→y) > 0 (веса положительны!)

Поэтому w(u→y→s) > w(u→t→s) – **противоречие!**

Время работы алгоритма Дейкстры

- Каждая вершина проходит одну релаксацию относительно всех соседей, поэтому надо потратить время O(E). Но организация очереди стоит времени, причем перестройка очереди происходит при просмотре каждого ребра. В худшем случае

$$T=O(E \log V)$$

Алгоритм Беллмана-Форда

Задача: найти кратчайший путь, если есть ребра отрицательного веса и нет циклов отрицательного веса, а если есть отрицательные циклы, то этот факт обнаружить

- Повторяем |V| раз релаксацию всех ребер.
- Если на последнем шаге хоть одно ребро релаксировало, то есть отрицательный цикл
- Время T=O(V*E)

Теорема Алгоритм находит оптимальный путь (!)

Доказательство.

Если есть оптимальный путь $\{v_1,..v_n\}$, то на i-шаге будет определена длина пути от i-вершины до конца. Доказательство по индукции. Инициация тривиальна.

Пусть на i-шаге $d(v_i)=\delta(v_i)$. Тогда на следующем шаге произойдет релаксация ребра (v_i, v_{i+1}) и таким образом определится вес оптимального пути от v_{i+1}.

Потоки в сетях. Определения

- Сеть:
  - ориентированный граф ***G(V,E)***, в котором на каждом ребре определено неотрицательное число ***c(e)*** (пропускная способность). Для пар вершин ***{u,v}***, не соединенных ребром, определим ***c(u,v)=0***
  - Выделены две вершины ***s*** и ***t*** (источник и сток).
  - Для любой вершины v существует путь ***s ~> v ~> t***

Потоки в сетях. Определения

- ***Поток*** – функция ***f:V x V → R:***
  - ***f(uv) ≤ c(uv), c(uv) –*** заданное ограничение (пропускная способность)
  - ***f(uv)= – f(vu)***
  - $\sum_{u \text{ из } V} f(u,v) = 0$ для любого v из (V – {s,t})
- Величина потока:

  $|f|=F(s) =\sum_{v \text{ из } V} f(s,v)$

  Задача: доказать, что F(t)= – F(s)

Проблема: найти максимальный поток в сети.

Можно определить поток между множествами, простым суммированием по ребрам

Потоки в сетях

Пусть определен некоторый поток ***f*** в сети ***G(V,E)***. Тогда для любой пары вершин ***u,v*** определим *остаточную пропускную способность*:

$$c_f(u,v) = c(u,v) - f(u,v)$$

Остаточная сеть для ***G*** и ***f*** определяется так:

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$$

Остаточная сеть может содержать встречные ребра. Они соответствуют случаям, когда ***f(u,v) < 0***

Метод Форда-Фалкерсона

Пусть ***G*** сеть, ***f*** – поток G_f – остаточная сеть и *f'* – поток в остаточной сети. Тогда ***f+f'*** является потоком и его величина $|f + f'| = |f| + |f'|$

Надо проверить несколько свойств ***f+f'***:

- *Кососимметричность*
- *Ограничение на пропускную способность*
- *Закон сохранения*

Метод Форда-Фалкерсона

Дано сеть ***G*** и поток ***f***.

Дополняющий путь ***p*** – простой путь из ***s*** в ***t***.

Остаточная пропускная способность $c_f(p)$:

$$c_f(p) = \min_{(u,v)\in p} \{c_f(u,v)\}$$

Если даны сеть, поток и существует дополняющий путь, то определим дополняющий поток f_p

$$f_p(u,v) = \begin{cases} c_f(p), & (u,v)\in p \\ -c_f(p), & (v,u)\in p \\ 0 \end{cases}$$

Разрезы в сетях

Разрезом сети называется разбиение множества вершин $V = S \cup T; \quad T = V - S, s \in S, t \in T$

Ребро ***e=(u,v)*** *принадлежит разрезу*, если

$$u \in S \ \ \& \ \ v \in T$$

Пропускная способность разреза = сумма пропускных способностей ребер разреза

Минимальный разрез – разрез с минимальной пропускной способностью

Потоки между множествами

$$f(X,Y) = \sum_{x \in X} \sum_{y \in Y} f(x,y)$$

Лемма

$$f(X,X) = 0$$

$$f(X,Y) = - f(Y,X)$$

$$\text{если} \quad X,Y,Z \subseteq V \text{ и } X \cap Y = 0 \text{ то}$$
$$f(X \cup Y, Z) = f(X,Z) + f(Y,Z)$$
$$f(Z, X \cup Y) = f(Z,X) + f(Z,Y)$$

$$X \subseteq V - s - t = f(X,V) = 0$$

Теорема о минимальном разрезе

Лемма. Даны: ***G, f, (S,T).*** Тогда поток ***f(S,T)*** равен величине потока ***|f|.***

$$f(S,T) = f(S,V) - f(S,S) = f(S,V) = f(s,V) + f(S-s,V) = f(s,V) = |f|$$

<u>Теорема о максимальном потоке и минимальном разрезе</u>

Пусть ***G=(V,E)*** – сеть и ***f*** – поток в сети.
Тогда следующие утверждения равносильны

1. Поток f максимален
2. Остаточная сеть не содержит дополн. путей
3. Для некоторого разреза ***(S,T)*** сети ***G*** выполнено равенство ***| f |=c(S,T).***

Доказательство

(1)→(2) – от противного (почти очевидно)

(2)→(3): В сети G_f нет пути из s в t. Рассмотрим множество

$$S = \{v \in V : в\, G_f \; \exists путь\; s - \; v\}$$

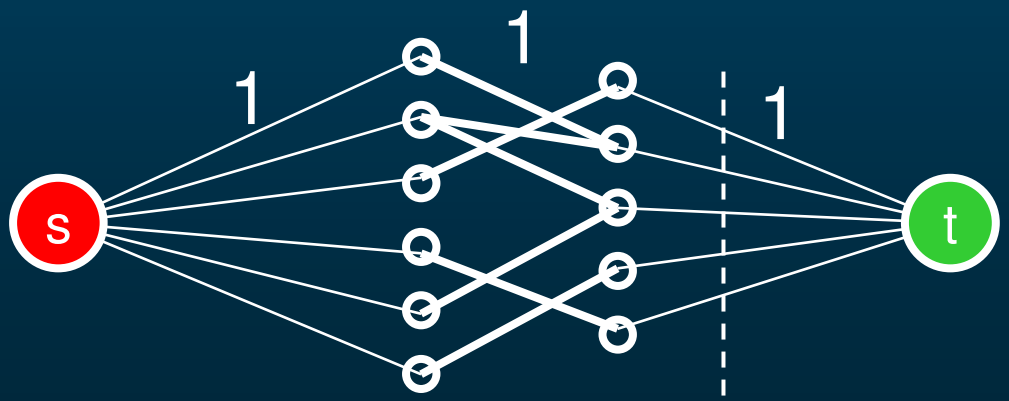
Положим ***T=V – S***. Это деление является разрезом. По построению никакое ребро из ***S*** в ***T*** не принадлежит E_f. Поэтому ***f(u,v)=c(u,v)***. Тогда по предыдущей лемме

$$|f| = f(S,T) = c(S,T)$$

(3)→(1) для любого разреза ***(S, T)*** выполнено ***|f| ≤ c(S, T)***, поэтому из равенства |f|=c(S, T) следует, что поток максимален

Паросочетания в двудольном графе

- Двудольный граф – множество вершин можно разбить на два подмножества, так, что нет ребер внутри этих подмножеств.
- Паросочетания – множество ребер в графе, такое, что нет ребер, имеющих общую вершину.
- Добавляем источник и сток, пропускные способности устанавливаем на всех ребрах графа =1. Ищем максимальный целочисленный поток



Задача сортировки. Нижние оценки

- Задача: дан массив элементов, для которых определены соотношения < и =, так, что для каждой пары элементов *x* и *y* выполняется одно из:

  $x<y$ или $y<x$ или $y=x$.

  расположить элементы так, чтобы для любого i выполнялось

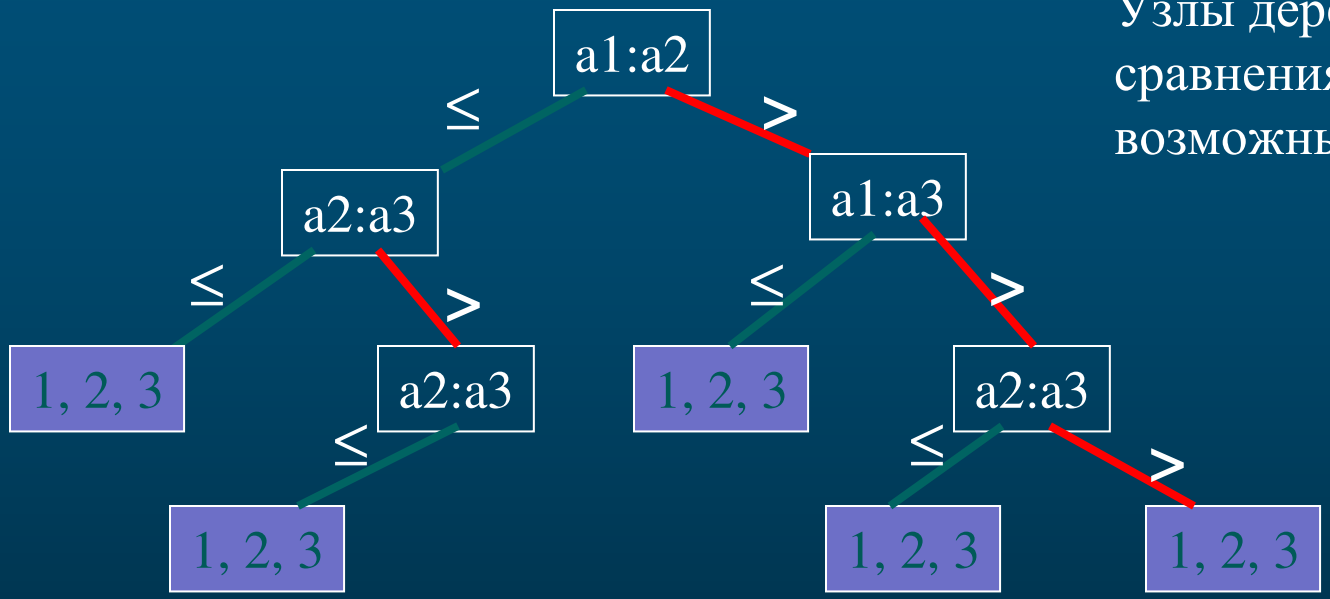
  $a[i] < a[i+1]$ или $a[i] = a[i+1]$

Теорема. Не существует алгоритма, решающего задачу сортировки быстрее, чем за $c\,n\ln(n)$:

$$T(n) = O(n \ln(n))$$

где n – размер массива, c – константа, $T(n)$ – время работы алгоритма

Решающие деревья



Узлы дерева – операции сравнения, листья – возможные перестановки

Любому алгоритму сортировки соответствует дерево решений: надо сделать сравнение и в зависимости от результата что-то сделать (например переставить два элемента) и затем сделать следующее сравнение

Время работы алгоритма сортировки

- Время работы алгоритма сортировки равно высоте h дерева решений.
- Максимальное число листьев в дереве высоты h равно 2^h
- Число листьев в дереве решений равно $n!$, поскольку каждому листу соответствует одна перестановка элементов массива.
- Поэтому для любого алгоритма должно выполняться $n! \leq 2^h$, или $h \geq \log(n!)$.
- $n! > (n/e)^n$
- Окончательно:
  $T(n)=c \cdot h > c \cdot (n \log(n) - n \log(e)) = O(n \log(n))$

Формальные языки

- Вход любого алгоритма сводится к строке (любая программа имеет на входе двоичный файл)
- Время работы алгоритма правильно оценивать как функцию длины входной строки
- Алфавит – некое (конечное) множество $\{\alpha_i\}$. Слово длины n – последовательность из n символов алфавита. Множество всех слов всех длин обозначается $\{\alpha_i\}^*$
- Язык *L* – подмножество $\{\alpha_i\}^*$: $L \subset \{\alpha_i\}^*$
- Поскольку алфавит конечен, то есть отображение множества всех слов в алфавите $\{\alpha_i\}$ на множество всех слов в алфавите {0,1}. Далее будем рассматривать языки над алфавитом {0,1}.

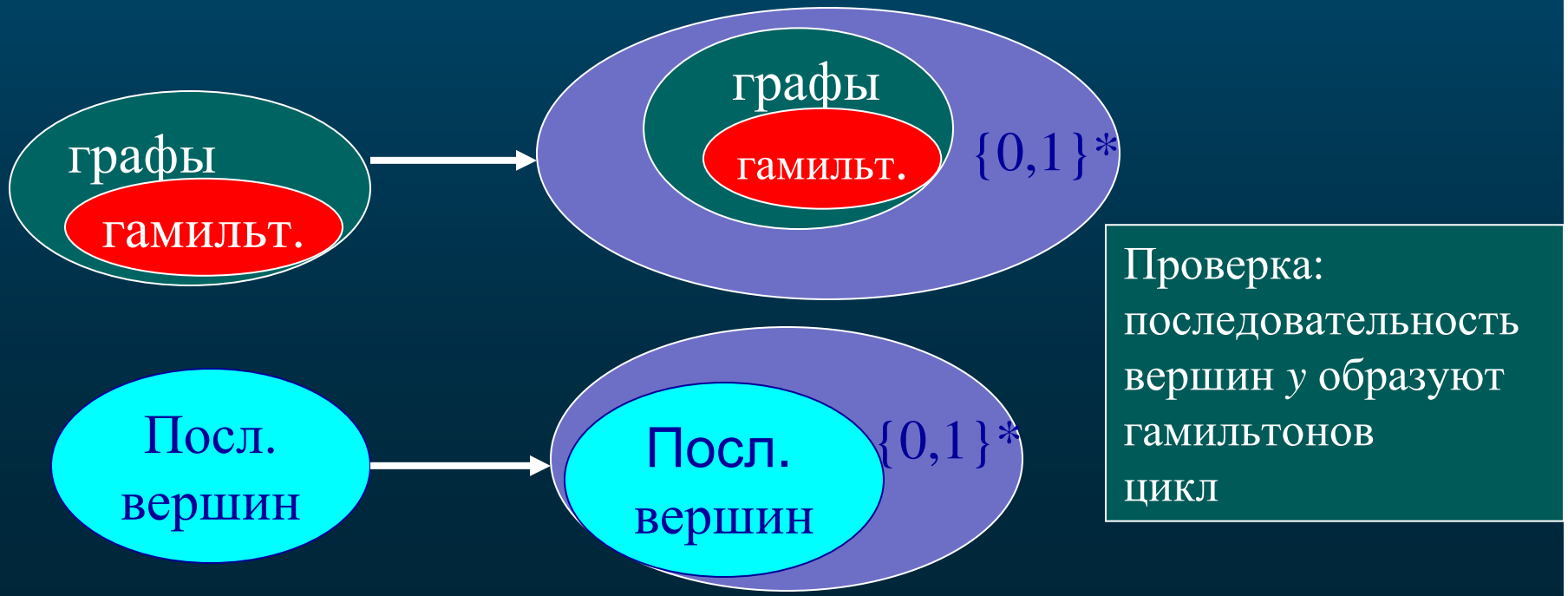
Алгоритмы и языки

- Алгоритм A ***допускает*** слово $x \in L$, если $A(x)=1$.
- Алгоритм A ***отвергает*** слово $x \in L$, если $A(x)=0$.
- Алгоритм может не допускать и не отвергать некоторый слова. Возможно, что алгоритм не дает никакого ответа (например, зацикливается)
- Алгоритм ***допускает*** язык L, если он допускает те и только те слова, которые принадлежат L.
- Алгоритм ***распознает*** язык L, если он допускает все слова из L и отвергает все другие слова.
- Язык ***допускается*** за полиномиальное время, если существует допускающий алгоритм A, причем его время работы порядка $O(n^k)$, где n – длина слова.
- Язык ***распознается*** за полиномиальное время если существует распознающий алгоритм A и число k такие, что его время работы порядка $O(n^k)$, где n – длина слова.

Проверяющие алгоритмы

- *Проверяющий* алгоритм *A(x,y)* это алгоритм с двумя аргументами: *словом* $x \in L$ и *сертификатом* $y \in \{0,1\}^*$.
- Проверяющий алгоритм *допускает* слово *x*, если *существует* сертификат *y*, такой, что *A(x,y)*=1 для $x \in L$ и не существует сертификата для других слов

Пример.



Попросту говоря …

- Проверяющий алгоритм – это такой алгоритм, который может сказать является ли представленное (Оракулом) решение правильным, но при этом он ничего не говорит о том как получить решение.

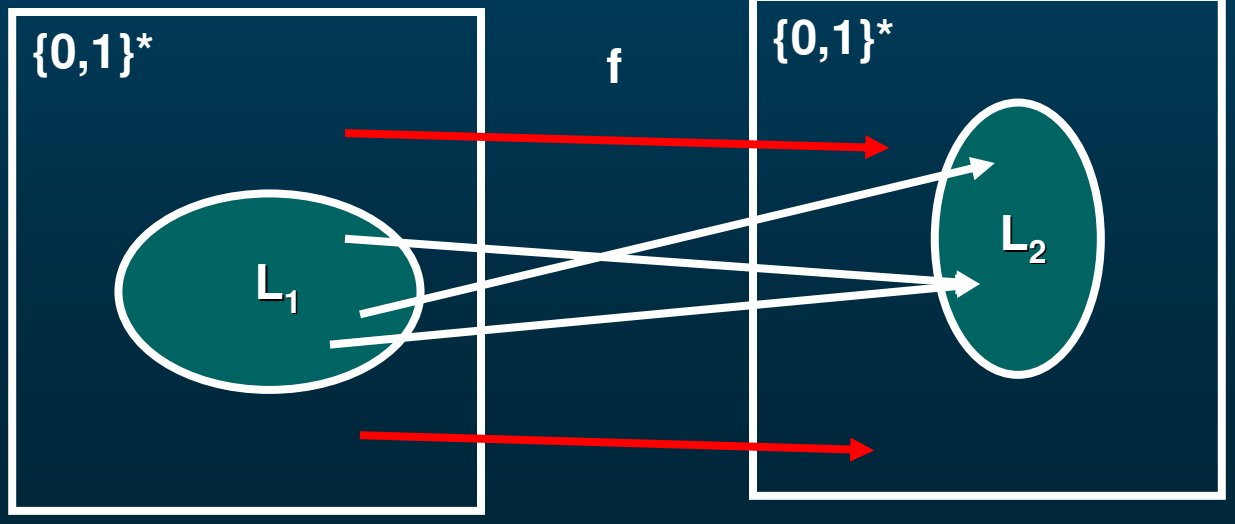
Сводимость

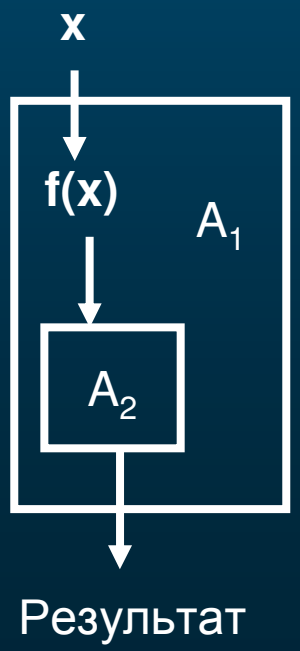
- Язык L_1 сводится к L_2 за полиномиальное время, если существует отображение f: {0,1}* → {0,1}*, такое, что

$$x \in L_1 \Leftrightarrow f(x) \in L_2$$

причем алгоритм вычисления f полиномиален.

Тогда пишут, что $L_1 \leq_p L_2$

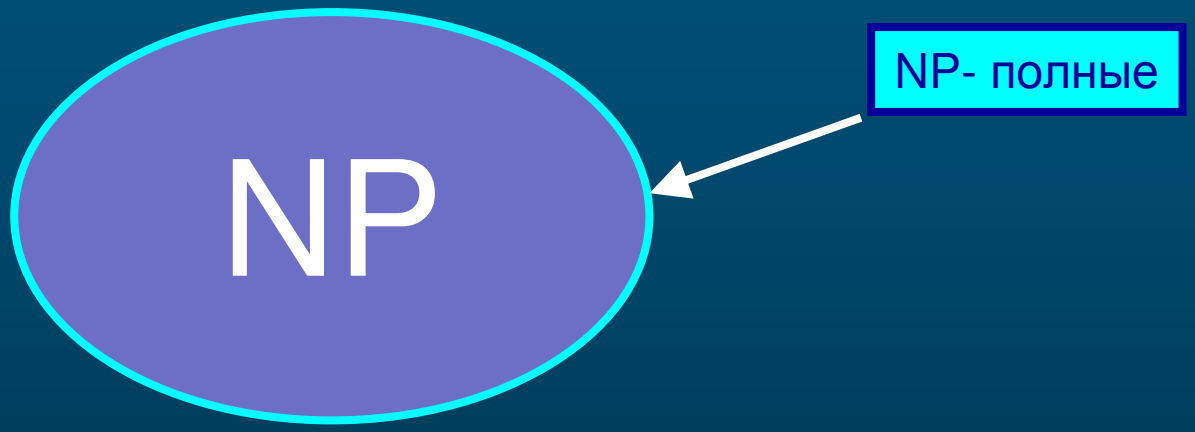




Класс языков NP

- Язык *L* принадлежит классу NP, если существует полиномиальный проверяющий алгоритм A, причем длина сертификата также ограничена полиномом от длины входной строки.
- Язык *L* принадлежит классу NP *полных* (NPC), если:
  1. $L \in$ NP
  2. $L' \leq_p L$ для любого $L' \in$ NP, иными словами, это задачи полиномиально не менее сложные, чем любая NP-задача

NP и NP- полные задачи



- Благодаря сводимости существует ***ОДНА*** и только одна NP- полная задача.
- Если для хотя бы одной NP- полной задачи найдется полиномиальный алгоритм, то ***ВСЕ*** NP- полные задачи можно будет решить за полиномиальное время

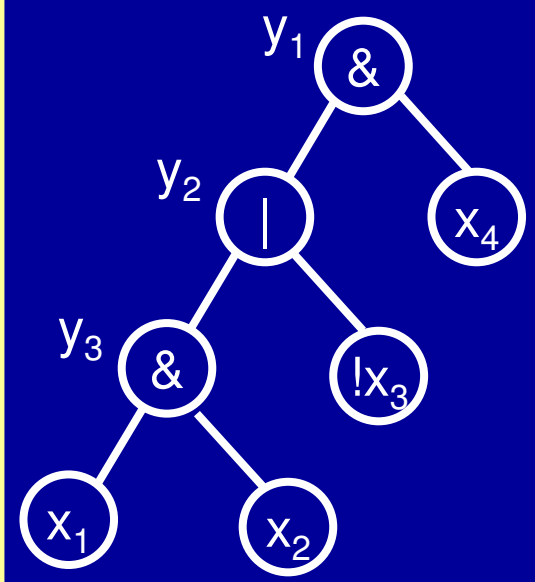
Примеры NP полных задач

- **Задача о выполнимости булевой формулы (SAT)**. Дана булева формула F, содержащая некоторое количество аргументов x_i, стандартные операции $\wedge, \vee, \neg$, между ними и скобки. Определить, существует ли такой набор аргументов x_i, для которых
$$F(x_i) = \textbf{true}$$
- **Задача о выполнимости 3-CNF – формул (3-CNF-SAT).** 3-CNF – формула это формула вида:
$(t_1 \vee t_2 \vee t_3) \wedge \ldots \wedge (t_k \vee t_{k+1} \vee t_{k+2})$, где $t_i = x_i$ или $t_i = \neg x_i$
это частный случай предыдущей задачи, поэтому она решается не медленнее задачи 1. С другой стороны, можно с помощью некоторого дерева и введением дополнительных переменных из любой булевой формулы получить 3-CNF.

Сведение SAT к 3-CNF

Дана формула: $\varphi(x_1, x_2, x_3, x_4)=((x_1 \& x_2) | !x_3) \& x_4$.

1. Строим дерево разбора формулы и вводим новые переменные в узлах дерева:



2. Переписываем формулу:

$$\varphi = y_1 \& (y_1 \equiv (y_2 \& x_4)) \& (y_2 \equiv (y_3 \mid !x_3)) \& (y_3 \equiv (x_1 \& x_2))$$

3. Каждую тройку преобразуем к 3-CNF.

y_1	y_2	x_4	$(y_1 \equiv (y_2 \& x_4))$	
0	0	0	1	
0	0	1	1	
0	1	0	1	
0	1	1	0	(y1 | !y2 | !x4) &
1	0	0	0	(!y1 | y2 | x4) &
1	0	1	0	(!y1 | !y2 | !x4) &
1	1	0	0	(!y1 | !y2 | x4)
1	1	1	1	

Еще примеры NP полных задач

- **<u>Задача о клике (CLIQUE).</u>** Дан граф G. Выяснить есть ли в нем клика, содержащая более k вершин.
  Проверка решения задачи тривиальна, поэтому это NP-задача. Оценим ее полноту. Для этого покажем, что

  $$3\text{-CNF-SAT} \leq_p \text{CLIQUE}, \quad \neg$$

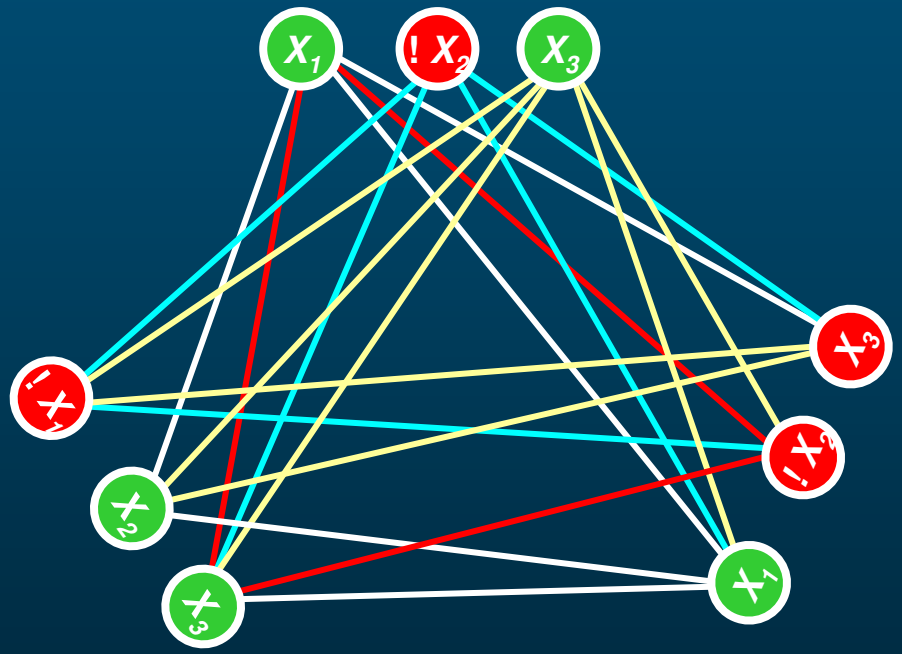
  Пусть дана 3-CNF формула. Построим по ней граф, причем такой, что если в нем есть k-клика, то эта формула выполнима. Формула:

  $$C_1 \wedge C_2 \wedge \ldots\ldots C_k; \quad C_i=(l_{1i} \vee l_{2i} \vee l_{3i})$$

  для каждого C_i рисуем 3 вершины. Вершины v_{ir} и v_{js} соединены ребром, если они принадлежат разным тройкам ($r \neq s$) и если они совместимы (не являются взаимным отрицанием).
  Если формула разрешима, то существует набор переменных, когда все C_i истины. Тогда есть в каждом C_i есть истинный литерал. Соответствующие вершины образуют клику. Обратное тоже верно.

Граф для 3-CNF

$f = (x_1 \,/\, !x_2 \,/\, x_3) \,\&\, (!x_1 \,/\, x_2 \,/\, x_3) \,\&\, (x_1 \,/\, !x_2 \,/\, !x_3)$



$x_1=1;$
$x_2=0;$
$x_3=1;$

"Мы знаем, что задача не имеет решения, но нам надо знать как ее решать"

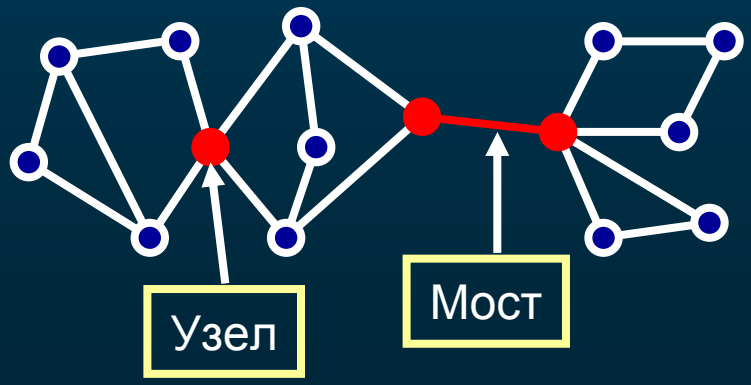
Стругацкие. "Понедельник начинается в субботу"

Что же делать?

- Не все NP- полные задачи для общего случая являются таковыми для некоторых специальных случаев.
  - Пример: задача о максимальной клике является NP- полной для общего вида графов, но является тривиальной для деревьев (Почему?)
- Строить приближенные эвристические алгоритмы
  - Для таких алгоритмов надо делать оценки их качества работы

Декомпозиция графа

- Неориентированный граф можно разбить на узлы, мосты, двусвязные компоненты (Двусвязная компонента – подграф, в котором через две любые вершины можно провести цикл)
- Задачу о клике надо решать в каждой двусвязной компоненте отдельно.



ε-граф

- ε-граф – граф, в котором число ребер равно $|E|=(1+\varepsilon)|V|$, $\varepsilon \ll 1$
- Для таких графов можно построить достаточно эффективные алгоритмы

Стохастические алгоритмы

- Основная идея.
  - С помощью датчика случайных чисел генерируем "решение"
  - С помощью случайных чисел модифицируем "решение". Если новое решение лучше, то принимаем, иначе с некоторой вероятностью отвергаем
  - Снова модифицируем "решение"
  - Повторяем эту процедуру много раз

Искусственный отжиг (Annealing) Общая схема

- Пусть нам надо минимизировать некую функцию (например, стоимость обхода в задаче коммивояжера)
- Определяем допустимое решение, или состояние (например, обход вершин графа)
- Определяем элементарные преобразования. Они должны обладать свойствами:
  - преобразование переводит одно допустимое решение в другое допустимое решение
  - между двумя любыми допустимыми решениями существует цепочка элементарных преобразований
  - Для любого состояние элементарные преобразования должны легко вычисляться

Искусственный отжиг (Annealing) Общая схема

1. Выбираем случайное допустимое решение (состояние i = 1)
2. Определяем его вес W_i
3. Выбираем случайное элементарное преобразование и вычисляем вес нового состояния W^*_{i+1}
4. Если вес нового состояния меньше веса исходного состояния ($W^*_{i+1}<W_i$), то переходим в новое состояние
5. Иначе переходим в новое состояние с вероятностью

$$p = \exp((W^*_{i+1}-W_i)/T),\ T = \alpha/\sqrt{i}$$

6. Если число итераций меньше N переходим к 2

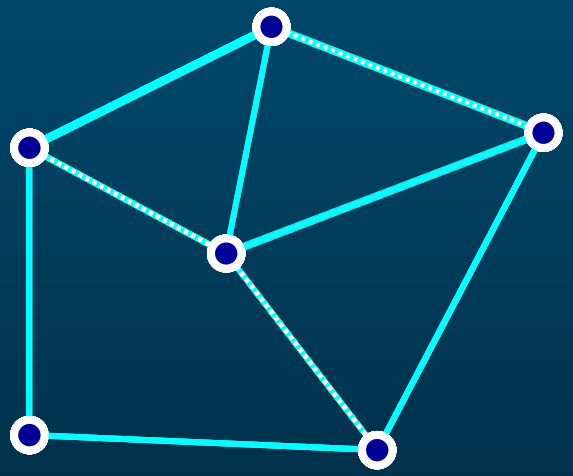
- Повторяем процедуру много раз

Задача коммивояжера

Допустимое решение

- Выбираем произвольный цикл, проходящий через вершины графа не более одного раза
- **Возможные модификации пути:**
- Если между двумя вершинами, включенными в обход есть альтернативные пути (может быть с включением новых промежуточных вершин, или с исключением принятых вершин), то преобразуем путь между этими вершинами.

Задача коммивояжера



Генетические алгоритмы
Основная идея

- Допустимое решение описывается как массив значений (например, битов). Допустимое решение должно обладать свойством:
  - Если мы возьмем часть значений из одного допустимого решения, а остальные значения из другого, то получим снова допустимое решение
- Далее создается популяция допустимых решений
- Шаги алгоритма:
  - Мутации (в некоторых членах популяции случайным образом меняются некоторые значения)
  - Скрещивание (кроссинговер). В результате появляются новые члены популяции
  - Отбор - самые слабые (плохие в смысле оптимизируемой функции) особи вымирают
- Эволюция популяции приводит к появлению оптимальных особей

Задача

Придумать схему генетического алгоритма для задачи коммивояжера

Подсказка: надо использовать другое определение для допустимого решения, допускающее даже отсутствие цикла.