Лекция 12
Остовные деревья

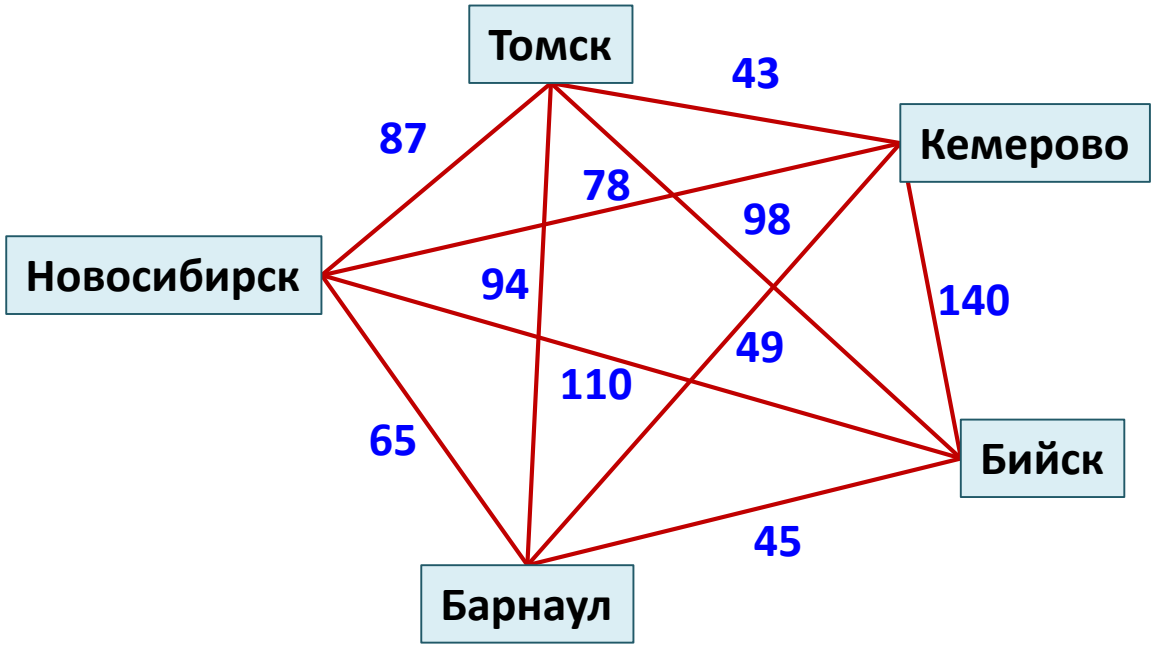
Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com
WWW: www.mkurnosov.net

Курс «Структуры и алгоритмы обработки данных»
Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)
Весенний семестр, 2016

Задача

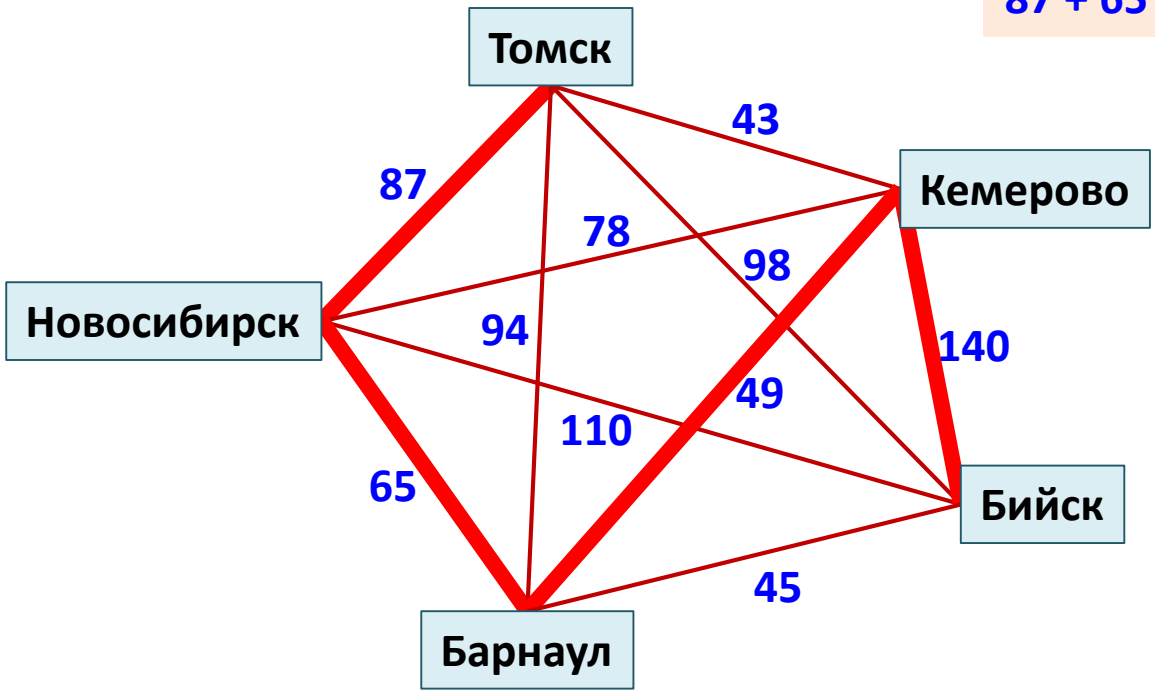
- **Имеется** *n* городов, которые необходимо соединить дорогами, так, чтобы можно было добраться из любого города в любой другой (напрямую или через другие города)
- Известна стоимость строительства дороги между любой парой городов (граф взвешенный)
- **Между какими городами строить дороги?**



Задача

- **Имеется** *n* городов, которые необходимо соединить дорогами, так, чтобы можно было добраться из любого города в любой другой (напрямую или через другие города)
- Известна стоимость строительства дороги между любой парой городов (задан взвешенный граф)
- **Между какими городами строить дороги?**

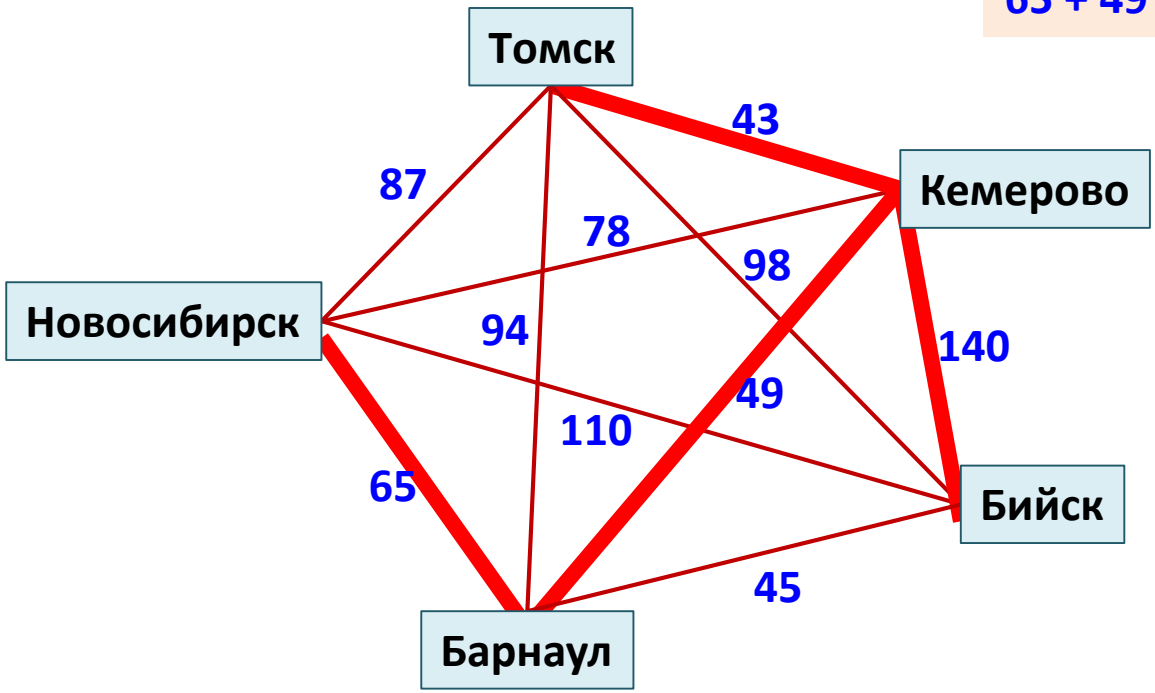
Стоимость проекта
87 + 65 + 49 + 140 = 341



Задача

- **Имеется** *n* городов, которые необходимо соединить дорогами, так, чтобы можно было добраться из любого города в любой другой (напрямую или через другие города)
- Известна стоимость строительства дороги между любой парой городов (задан взвешенный граф)
- Между какими городами строить дороги?

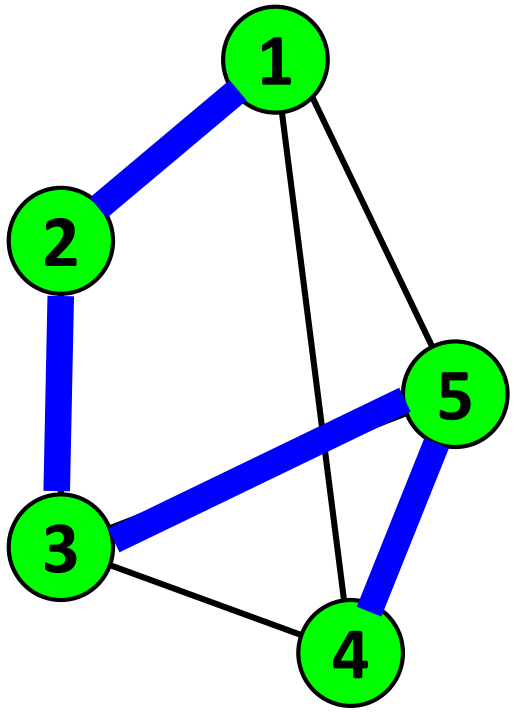
Стоимость проекта
65 + 49 + 140 + 43 = 297

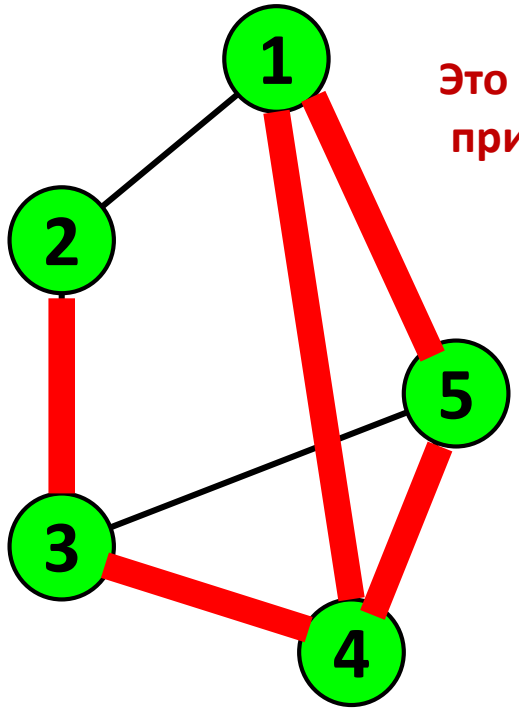


Остовные деревья

- **Ó**стовное дерево связного графа (spanning tree) – это ациклический связный подграф (дерево), в который входят все вершины данного графа

- Синонимы: *остов*, *покрывающее дерево*, *скелет графа*

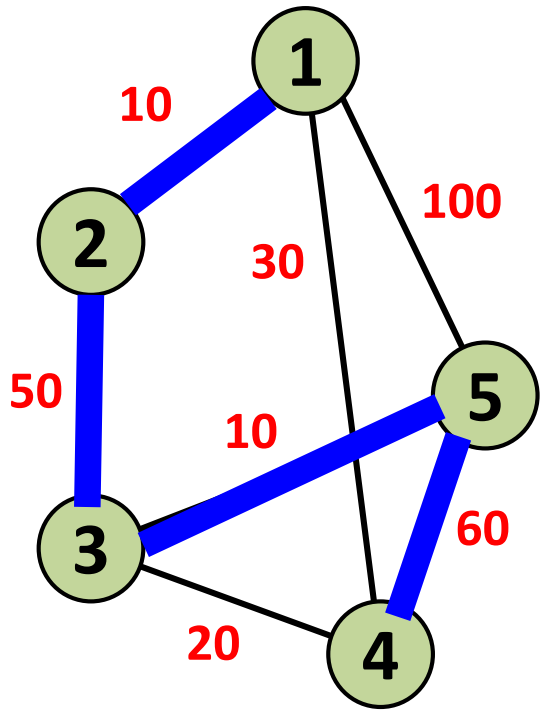




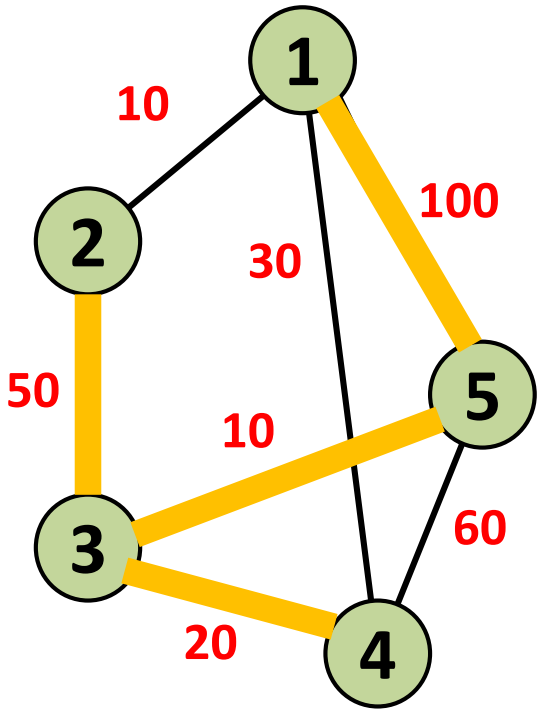
Это не остов – присутствует цикл

Остовные деревья минимальной стоимости

- Если граф взвешенный, то рассматривается задача о нахождении **остовного дерева с минимальной суммой весов входящих в него рёбер**



Cost = 10 + 50 + 10 + 60 = **130**



Cost = 100 + 10 + 50 + 20 = **180**

Применение остовных деревьев

- **Остовное дерево минимальной стоимости (minimum spanning tree, MST)** – это остовное дерево с минимальной суммой весов его ребер

- **Практическое применение MST**
  - Формирование дерева для широковещательной рассылки информации в сети (tree for broadcasting)
  - прокладка TV-кабеля между домами (вес ребер – стоимость прокладки кабеля между парой домов)
  - Spanning Tree Protocol в телекоммуникационных сетях стандарта Ethernet для предотвращения образования циклов в сети
  - …

Алгоритмы построения MST

Алгоритм	Вычислительная сложность		
	Матрица смежности	**Списки смежности + двоичная куча**	**Списки смежности + фибоначчиева куча**
Крускала (J. Kruskal, 1956)	$O(\|E\| \log \|V\|)$		
Прима (V. Jarník, 1930; R. Prim, 1957; E. Dijkstra, 1959)	$O(\|V\|^2)$	$O((\|V\| + \|E\|) \log \|V\|)$	$O(\|E\| + \|V\| \log \|V\|)$
Борувки (O. Borůvka, 1926)	$O(\|E\| \log \|V\|)$		
(B. Chazelle, 2000)	$O(\|E\| \cdot \alpha(\|E\|, \|V\|))$, где α(*m*, *n*) – обратная функция Аккермана		

Система непересекающихся множеств

- **Система непересекающихся множеств (disjoint-set data structure)** – структура данных для представления непересекающихся множеств

- Поддерживает следующие операции:
  - **MakeSet(*i*)** – создает множество из одного элемента *i*
  - **FindSet(*i*)** – возвращает номер множества, которому принадлежит элемент *i*
  - **UnionSets(*i*, *j*)** – объединяет множества, содержащие элементы *i* и *j*

- **Подробное описание**
  - **(Aho, C. 169, MFSET)**
  - **(Levitin, C. 381)**
  - **(CLRS, C. 597)**

Система непересекающихся множеств

- MakeSet(1)
- MakeSet(4)
- MakeSet(6)
- MakeSet(3)

4 множества: **{1}, {4}, {6}, {3}**

- UnionSets(1, 3)

{1, 3}, {4}, {6}

- UnionSets(4, 3)

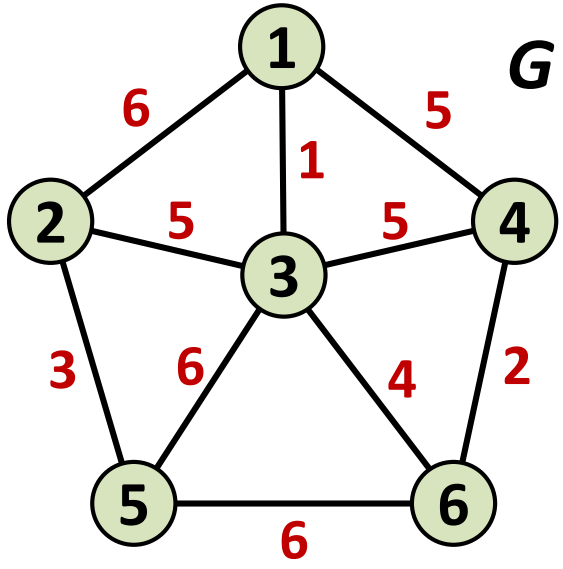
{1, 3, 4}, {6}

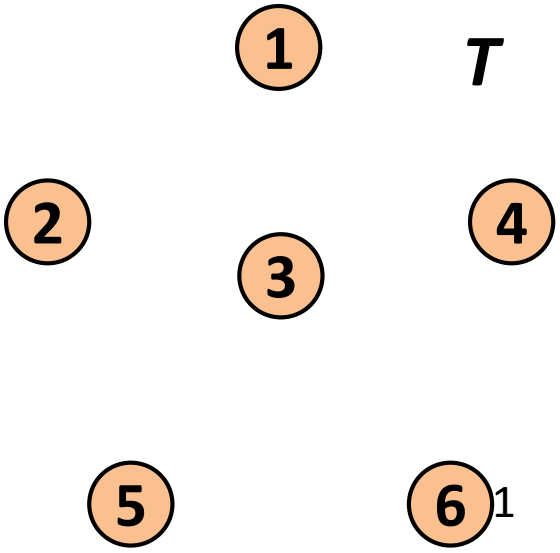
Алгоритм Крускала (Kruskal)

1. Создается пустой граф *T* из *n* вершин, несвязанных ребрами

2. Все ребра исходного графа *G* помещают в очередь с приоритетом

   Приоритет – вес ребра w_{ij}
   (ребра упорядочиваются
   по не убыванию весов – min heap)





***Prio. Q*: (1, 3), (4, 6), (2, 5), (3, 6), (2, 3), …**

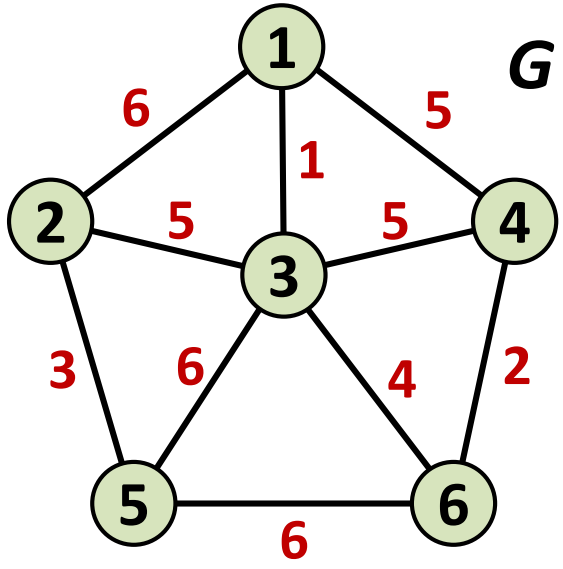
В графе *T* 6 компонент связности

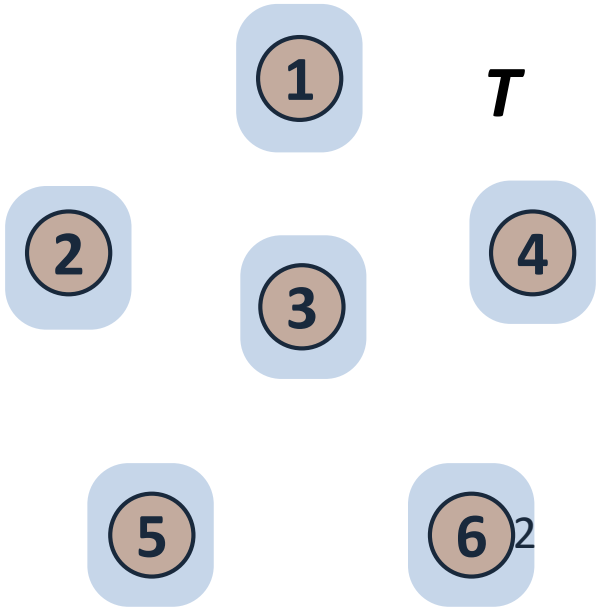
Алгоритм Крускала (Kruskal)

3. Цикл из $n - 1$ итерации
   (по количеству ребер в MST)

   a) Из очереди извлекается ребро (i, j) с минимальным весом (HeapDeleteMin)

   b) Если ребро (i, j) связывает вершины из разных компонент связности графа T, то ребро добавляется в граф T





***Prio. Q*: (1, 3), (4, 6), (2, 5), (3, 6), (2, 3), …**

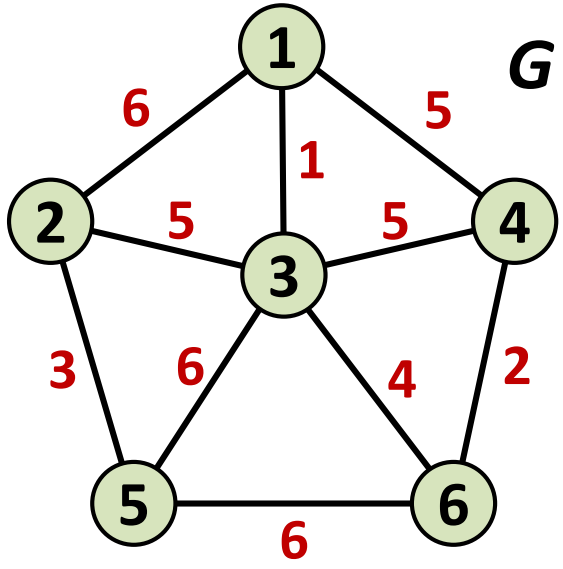
Алгоритм Крускала (Kruskal)

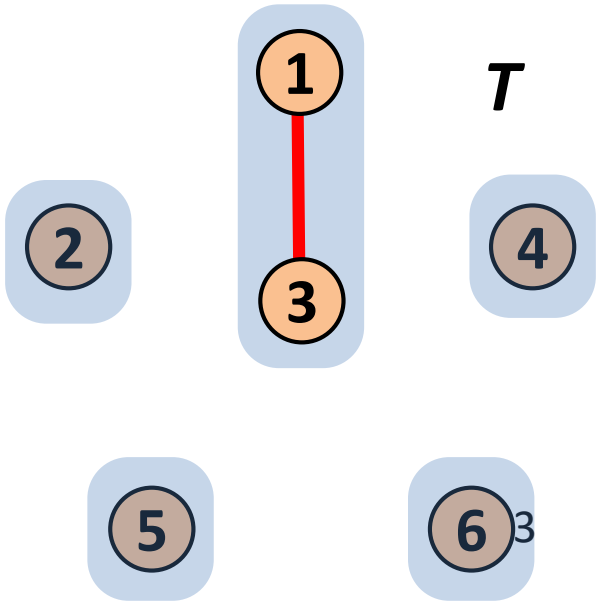
3. Цикл из $n - 1$ итерации
   (по количеству ребер в MST)

   a) Из очереди извлекается ребро (i, j) с минимальным весом (HeapDeleteMin)

   b) Если ребро (i, j) связывает вершины из разных компонент связности графа T, то ребро добавляется в граф T

Q: (1, 3), (4, 6), (2, 5), (3, 6), (2, 3), …





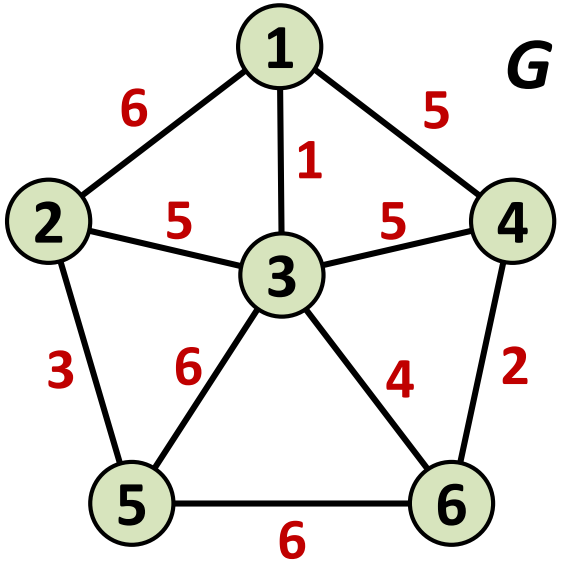
Алгоритм Крускала (Kruskal)

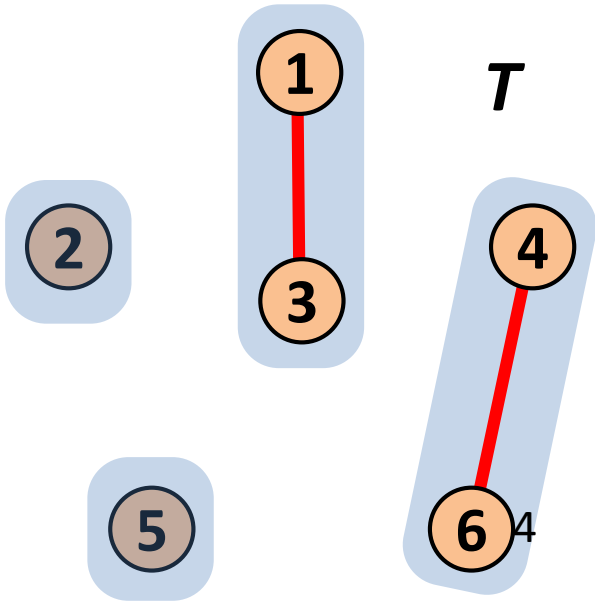
3. Цикл из $n - 1$ итерации
   (по количеству ребер в MST)

   a) Из очереди извлекается ребро (i, j) с минимальным весом (HeapDeleteMin)

   b) Если ребро (i, j) связывает вершины из разных компонент связности графа T, то ребро добавляется в граф T

Q: **(4, 6), (2, 5), (3, 6), (2, 3), …**





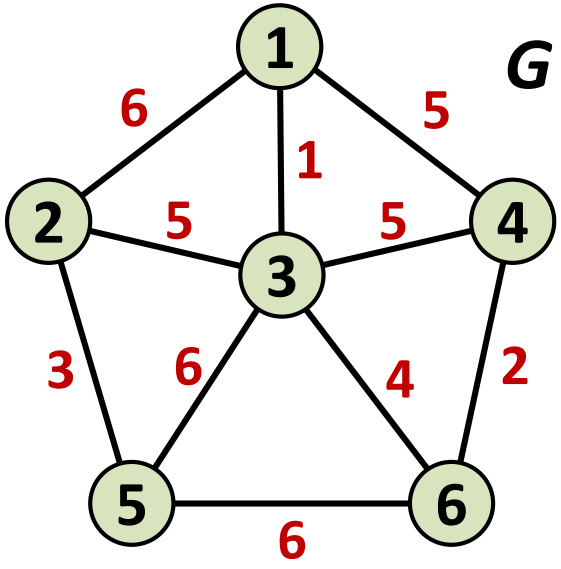
Алгоритм Крускала (Kruskal)

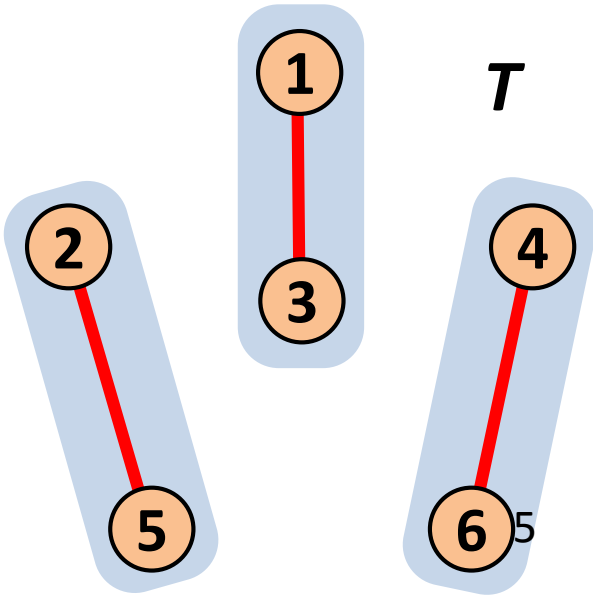
3. Цикл из $n - 1$ итерации
   (по количеству ребер в MST)

   a) Из очереди извлекается ребро (i, j) с минимальным весом (HeapDeleteMin)

   b) Если ребро (i, j) связывает вершины из разных компонент связности графа T, то ребро добавляется в граф T

Q: (2, 5), (3, 6), (2, 3), …





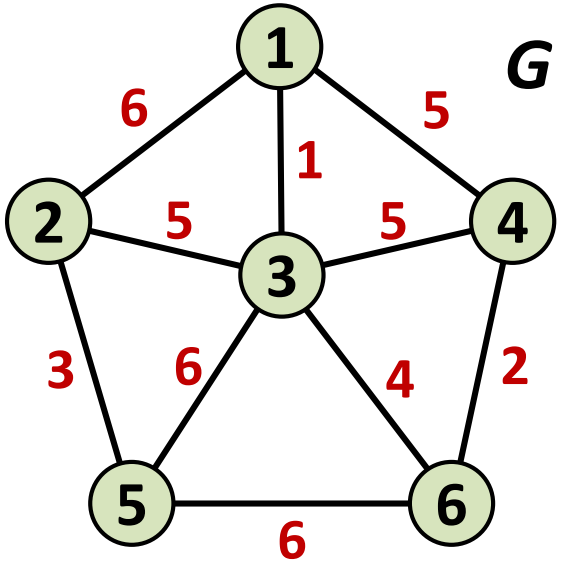
Алгоритм Крускала (Kruskal)

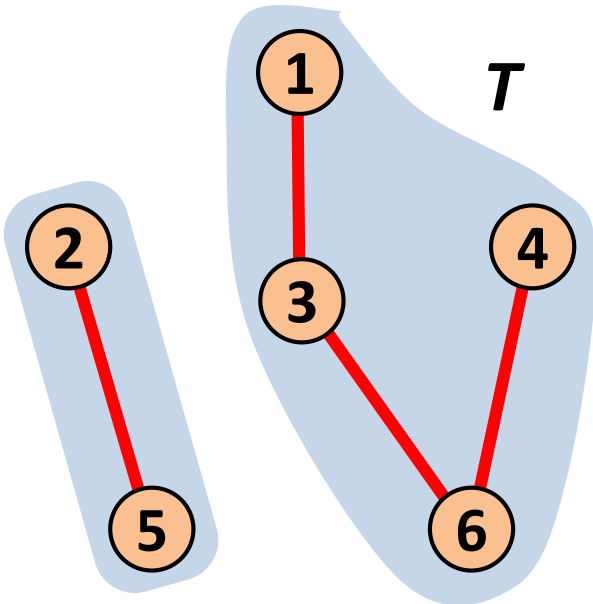
3. Цикл из $n - 1$ итерации
   (по количеству ребер в MST)

   a) Из очереди извлекается ребро (i, j) с минимальным весом (HeapDeleteMin)

   b) Если ребро (i, j) связывает вершины из разных компонент связности графа T, то ребро добавляется в граф T

Q: (3, 6), (2, 3), …





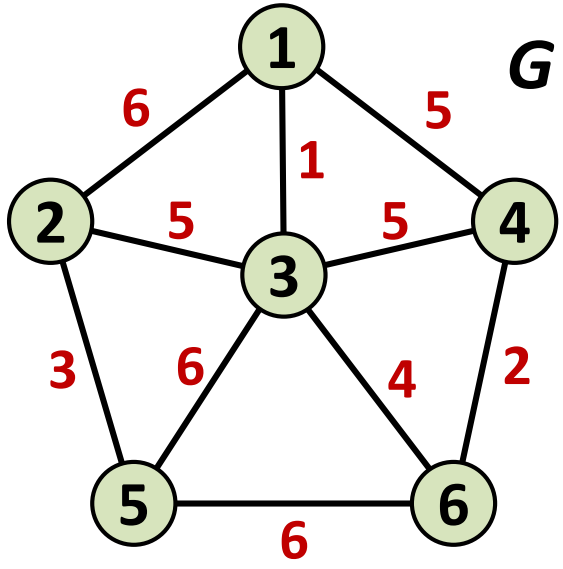
Алгоритм Крускала (Kruskal)

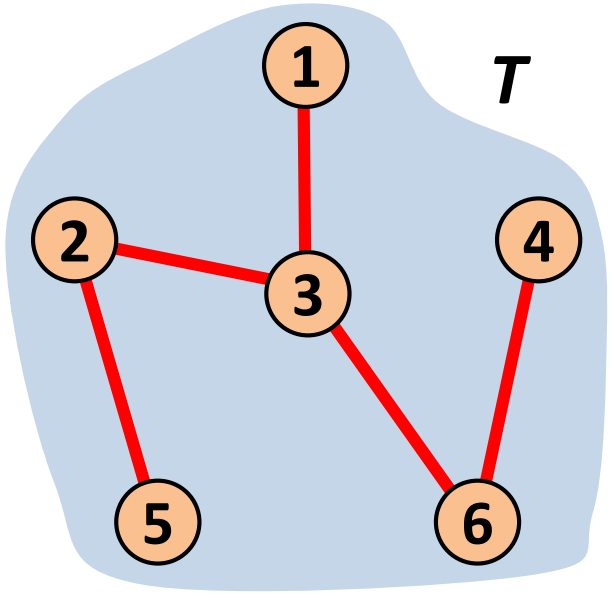
3. Цикл из $n - 1$ итерации
   (по количеству ребер в MST)

   a) Из очереди извлекается ребро (i, j) с минимальным весом (HeapDeleteMin)

   b) Если ребро (i, j) связывает вершины из разных компонент связности графа T, то ребро добавляется в граф T

***Q*: (2, 3), …**



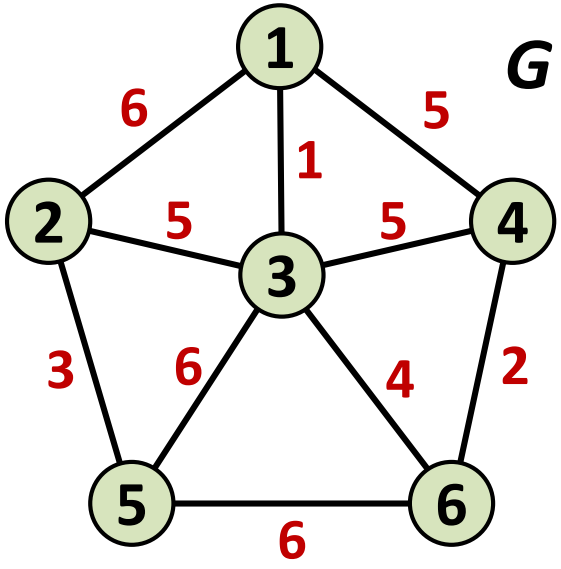


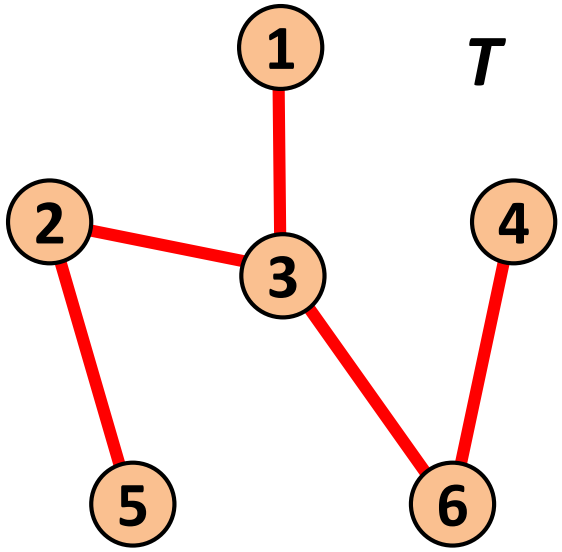
Алгоритм Крускала (Kruskal)

3. Цикл из $n - 1$ итерации
   (по количеству ребер в MST)

   a) Из очереди извлекается ребро (i, j) с минимальным весом (HeapDeleteMin)

   b) Если ребро (i, j) связывает вершины из разных компонент связности графа T, то ребро добавляется в граф T





- Построили остовное дерево T минимальной стоимости (MST)
- Стоимость 1 + 5 + 3 + 4 + 2 = 15

Алгоритм Крускала (Kruskal)

```
function MST_Kruskal([in] G, [out] T)
    // Input: G = (V, E)
    // Output: T = (V, E')
    T = CreateGraph(|V|)

    // Помещаем вершину i в отдельное множество
    // (компоненту связности графа T)
    for each i in V do
        MakeSet(i)
    end for

    // Помещаем ребра в очередь с приоритетом
    for each (i, j) in E do
        PriorityQueueInsert(w[i][j], (i, j))
    end for

    C = |V|   // Количество компонент связности
```

Алгоритм Крускала (Kruskal)

```
    while C > 1 do
        // Извлекаем ребро с минимальным весом
        (i, j) = PriorityQueueRemoveMin()
        seti = FindSet(i)
        setj = FindSet(j)
        if seti != setj then
            // Концы ребра из разных множеств
            GraphAddEdge(T, (i, j))
            UnionSets(i, j)
            C = C - 1
        end if
    end for
end function
```

Алгоритм Крускала (Kruskal)

```
function MST_Kruskal([in] G, [out] T)
    // Input: G = (V, E)
    // Output: T = (V, E')
    T = CreateGraph(|V|)

    // Помещаем вершину i в отдельное множество
    // (компоненту связности графа T)
    for each i in V do
        MakeSet(i)
    end for

    // Помещаем ребра в очередь с приоритетом
    for each (i, j) in E do
        PriorityQueueInsert(w[i][j], (i, j))
    end for

    C = |V|   // Количество компонент связности
```

MFSET (Aho)
$O(|V|)$

Binary heap
$O(|E|\log|E|)$

Алгоритм Крускала (Kruskal)

```
    while C > 1 do
        // Извлекаем ребро с минимальным весом
        (i, j) = PriorityQueueRemoveMin()    O(log|E|)
        seti = FindSet(i)                    O(1)
        setj = FindSet(j)                    O(1)
        if seti != setj then
            // Вершины ребра из разных множеств
            GraphAddEdge(T, (i, j))
            UnionSets(i, j)                  O(|V|)
            C = C - 1
        end if
    end for
end function
```

- В худшем случае цикл выполняется $|E|$ раз

Алгоритм Крускала (Binary heap + MFSET)

$$T_{Kruskal} = \underbrace{O(|V|)}_{\text{Создание множеств}} + \underbrace{O(|E|\log|E|)}_{\text{Вставка ребер в очередь}} + \underbrace{O(|E|\log|E|)}_{\text{Извлечение ребер из очереди}} + \underbrace{O(|V|^2)}_{\text{Объединение множеств}}$$

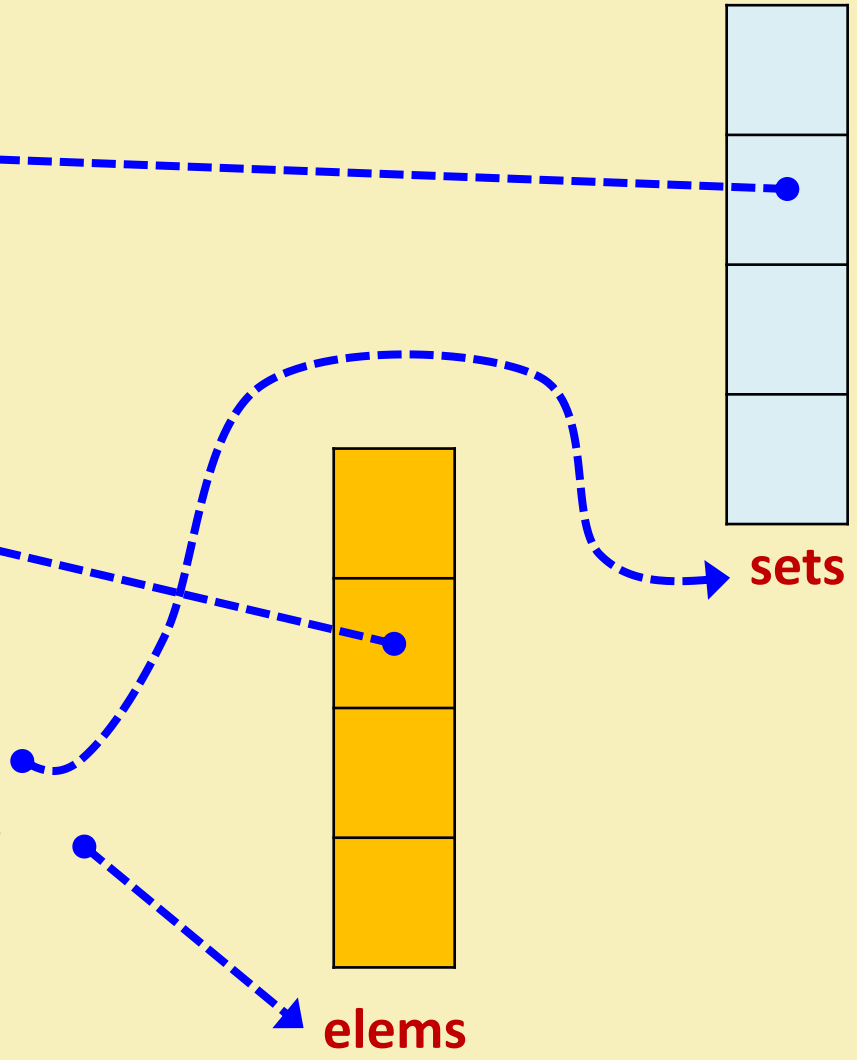
$$\log|E| \leq \log|V|^2 = 2\log|V|$$

$$T_{Kruskal} = O(|E|\log|E|) + O(|V|^2) = O(|E|\log|V|) + O(|V|^2)$$

- **Отчего зависит сложность алгоритма Крускала**
  - от реализации сортировки ребер по их весу (Sort, Binary heap, …)
  - от реализации системы непересекающихся множеств

MFSET

```
/* mfset.h: Disjoint set data structure */
struct set {
    int size;
    int first;
};

struct elem {
    int set;
    int next;
};

struct mfset {
    struct set *sets;
    struct elem *elems;
    int nelems;
    int nsets;
};
```



MFSET

```
struct mfset *mfset_create(int nelems)
{
    struct mfset *p;
    int i;

    p = malloc(sizeof(*p));
    p->nelems = nelems;
    p->nsets = 0;
    p->sets = malloc(sizeof(struct set) * nelems);
    p->elems = malloc(sizeof(struct elem) * nelems);
    for (i = 0; i < nelems; i++) {
        p->sets[i].size = 0;
        p->sets[i].first = -1;
        p->elems[i].set = -1;
        p->elems[i].next = -1;
    }
    return p;
}
```

$T_{Create} = O(nelems)$

MFSET

```
void mfset_free(struct mfset *set)
{
    free(set->sets);
    free(set->elems);
    free(set);
}

void mfset_makeset(struct mfset *set, int elem)
{
    set->sets[set->nsets].size = 1;
    set->sets[set->nsets].first = elem;
    set->elems[elem].set = set->nsets;
    set->elems[elem].next = -1;
    set->nsets++;
}
```

$T_{MakeSet} = O(1)$

MFSET

```
int mfset_findset(struct mfset *set, int elem)
{
    return set->elems[elem].set;
}
```

$T_{FindSet} = O(1)$

MFSET

```
void mfset_union(struct mfset *set,
                 int elem1, int elem2)
{
    int temp, i, set1, set2;

    set1 = mfset_findset(set, elem1);
    set2 = mfset_findset(set, elem2);

    if (set->sets[set1].size <
        set->sets[set2].size)
    {
        temp = set1;
        set1 = set2;
        set2 = temp;
    }
```

MFSET

```
    /* S1 > S2; Merge elems of S2 to S1 */
    i = set->sets[set2].first;
    while (set->elems[i].next != -1) {
        set->elems[i].set = set1;
        i = set->elems[i].next;
    }
    /* Add elems of S1 to the end of S2 */
    set->elems[i].set = set1;
    set->elems[i].next = set->sets[set1].first;
    set->sets[set1].first = set->sets[set2].first;
    set->sets[set1].size += set->sets[set2].size;

    /* Remove S2 */
    set->sets[set2].size = 0;
    set->sets[set2].first = -1;
    set->nsets--;
}
```

$T_{Union} = O(n)$

Алгоритм Крускала (Adj. matrix + MFSET + Bin. heap)

```
int search_mst_kruskal(struct graph *g,
                       struct graph *mst)
{
    struct mfset *set;
    struct heap *pq;
    struct heapvalue edge;
    struct heapitem item`;
    int mstlen, i, j, n, w, s1, s2;

    n = g->nvertices;
    mstlen = 0;
    /* Create forest (N sets) */
    set = mfset_create(n);
    for (i = 0; i < n; i++) {
        mfset_makeset(set, i);
    }
```

Алгоритм Крускала (Adj. matrix + MFSET + Bin. heap)

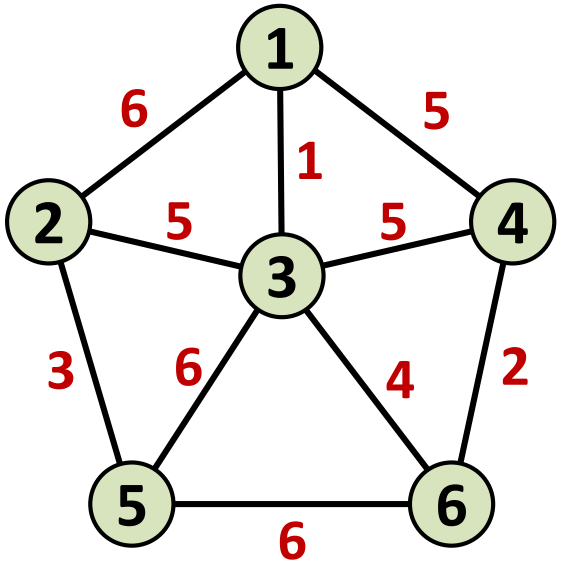
```
/* Insert edges in heap */
pq = heap_create(n * n);

/* For all edges (adj. matrix) */
for (i = 0; i < n; i++) {
    for (j = i + 1; j < n; j++) {
        w = graph_get_edge(g, i + 1,
                                j + 1);
        if (w > 0) {
            edge.i = i;
            edge.j = j;
            heap_insert(pq, w, edge);
        }
    }
}
```

Алгоритм Крускала (Adj. matrix + MFSET + Bin. heap)

```
    for (i = 0; i < n - 1; ) {
        item = heap_removemin(pq);
        s1 = mfset_findset(set, item.value.i);
        s2 = mfset_findset(set, item.value.j);

        if (s1 != s2) {
            mfset_union(set, item.value.i,
                        item.value.j);
            mstlen += item.priority;
            graph_set_edge(mst, item.value.i + 1,
                           item.value.j + 1,
                           item.priority);
            i++;
        }
    }

    heap_free(pq);
    mfset_free(set);
    return mstlen;
}
```

Пример

```
int main()
{
    struct graph *g, *mst;
    int i, j, mstlen;

    g = graph_create(6);
    graph_set_edge(g, 1, 2, 6);
    graph_set_edge(g, 1, 3, 1);
    graph_set_edge(g, 1, 4, 5);
    graph_set_edge(g, 2, 3, 5);
    graph_set_edge(g, 2, 5, 3);
    graph_set_edge(g, 3, 4, 5);
    graph_set_edge(g, 3, 5, 6);
    graph_set_edge(g, 3, 6, 4);
    graph_set_edge(g, 4, 6, 2);
    graph_set_edge(g, 5, 6, 6);
```



Пример

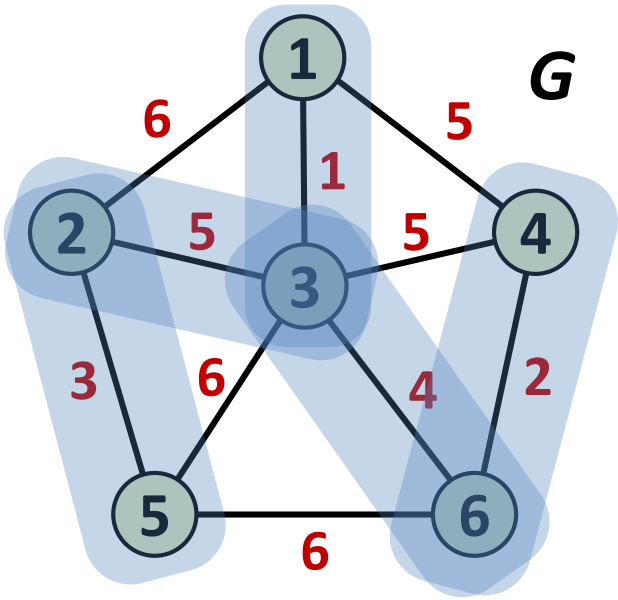
```
    mst = graph_create(6);
    mstlen = search_mst_kruskal(g, mst);

    printf("Minimum spanning tree: %d\n", mstlen);
    for (i = 0; i < 6; i++) {
        for (j = 0; j < 6; j++) {
            printf("%4d ", graph_get_edge(mst,
                    i + 1, j + 1));
        }
        printf("\n");
    }

    graph_free(mst);
    graph_free(g);
    return 0;
}
```

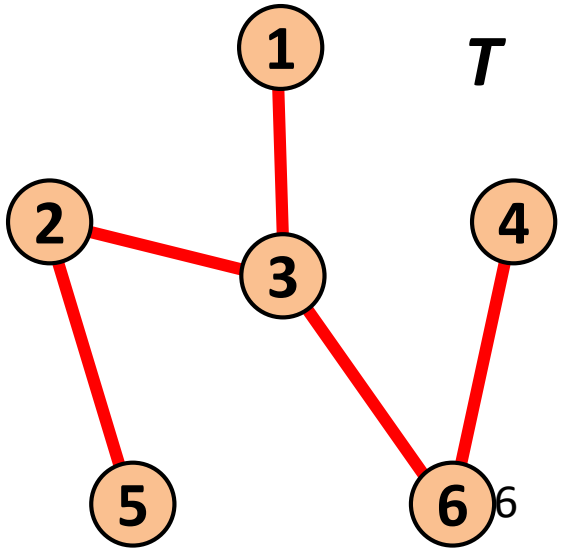
Пример

```
$ ./kruskal
Minimum spanning tree: 15
   0    0    1    0    0    0
   0    0    5    0    3    0
   1    5    0    0    0    4
   0    0    0    0    0    2
   0    3    0    0    0    0
   0    0    4    2    0    0
```

Алгоритм Прима (Prim)

1. Создается пустой граф *T*.
2. Во множество *U* помещается вершина 1, с которой начнется формирование остова.
3. Цикл пока $U \neq V$
   a) Найти ребро (*i*, *j*) с наименьшим весом такое, что $i \in U$ и $j \in V$
   b) Добавить ребро (*i*, *j*) в граф *T*
   c) Добавить вершину *j* во множество *U*





Домашнее чтение

- Прочитать о системе непересекающихся множеств в [Aho] и [CLRS]