Teoretical part.

Task 3.1

1. $T(n) = aT(n/b) + f(n)$

For expected time complexity of quick sort let's consider $a=2$, $b=2$ & $f(n)=n-1$ $\Rightarrow$ So, array is divided on two approximatley equal parts $\rightarrow$ Linear to divide & combine

$\Theta(n \log n)$

$$T(n) = 2T(n/2) + n - 1 = 4T(n/4) + 2n - 3 = 8T(n/8) + 3n - 7 = \ldots =$$

$$= 2^i T(n/2^i) + in - c = \ldots$$

According to improvement, recursive step will stop, when $n/2^i = k \Rightarrow i = \log \frac{n}{k}$

$$T(n) = \frac{n}{k} T(k) + n \cdot \log \frac{n}{k} - c \quad (\Rightarrow)$$

To sort k object with insertion sort expected time complexity: $T(k) = \Theta(k^2)$

$$(\Rightarrow)\ T(n) = \frac{n}{k}\Theta(k^2) + n \log \frac{n}{k} - c < C_1(nk + n \log \frac{n}{k})$$

$$\Downarrow \qquad n = n_0$$

$$T(n) = O(nk + n \log \frac{n}{k}) \quad \text{Q.E.D.}$$

2. In theory, if constant factors for $nk + n \log \frac{n}{k} < n \log n$ are ignored

$k + \log n < \log k + \log n$

$\log k > k \rightarrow$ impossible

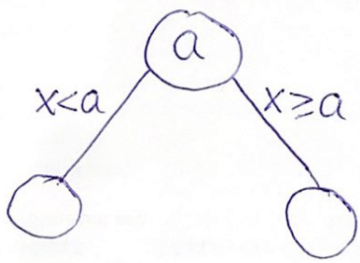
If $C_1 nk + C_2 n \log \frac{n}{k} < C_3 n \log n$

Let $C_2 = C_3 \rightarrow C_1 k < C_2 \log k \rightarrow \frac{\log 2^k}{\log k} < \frac{C_2}{C_1}$

In practice, should be chosen by experiment.

Task 3.2

1. According to the rule, that every object should be added to the BST to the left, if it's $<$, than current object, ~~else~~ or to the right otherwise.

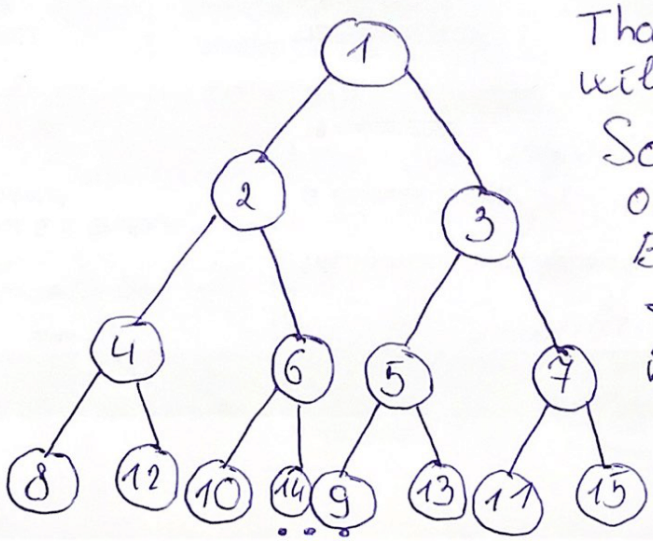


If $x = a$, than it will always be added to the right. (Some kind of linked-list)

$$\underbrace{1 + 2 + 3 + 4 + \ldots + n}_{\text{steps to add } n \text{ object in initially empty BST}} = \sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \Theta(n^2)$$

$$T(n) = \Theta(n^2)$$

2. Objects will be added ~~by~~ such a schema:



That means that BST will be balanced.

So, to add new object to Balanced BST $\log n$.

$$\sum_{k=1}^{n} \log k = \Theta(n \log n)$$

$$T(n) = \Theta(n \log n)$$

3. List → ArrayList
   → LinkedList

3.1 Linked List.

3.1.1 Without tail-pointer

As in 1, adding n objects to Linked List will cost $\Theta(n^2)$

$$T(n) = \Theta(n^2)$$

3.1.2 With tail-pointer: $T(n) = \Theta(n)$ Costs const to add 1 element

3.2 Array List

However, add new object to array list cost const ($O(1)$). So, n object will be added in linear time.

$$T(n) = \Theta(n)$$

4.

4.1 Worst-case.

In worst-case new object each time will be added to the left or to the right, which will be the same as in 1.

So, $T(n) = \Theta(n^2)$.

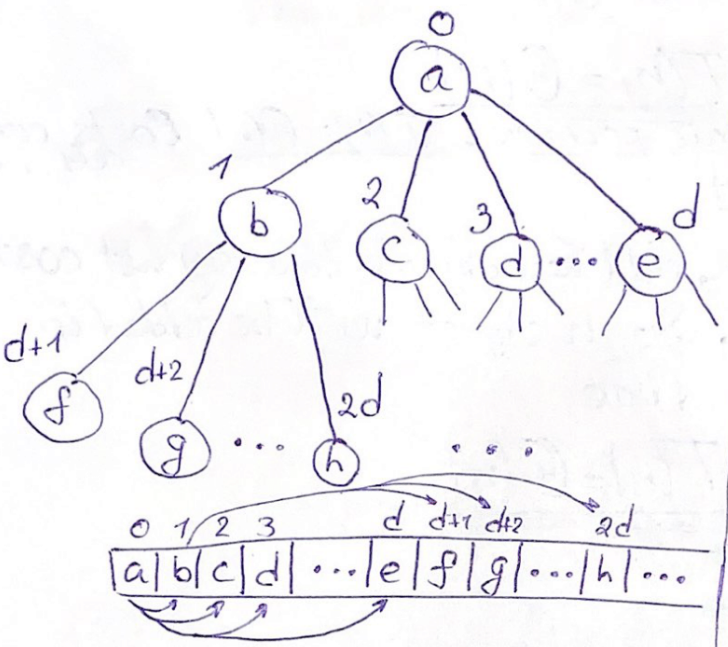
4.2 Best-case

In best-case new objects will be added by rotation to the left or right side of each branch. So, as the result, tree will be approximatley balanced. It's like in 2. So, $T(n) = \Theta(n \cdot \log n)$

Task 3.3.

1. Array representation:

$$f(p) = \begin{cases} 0, \text{ if } p\text{-root} \\ d \cdot f(q) + 1, p - 1^{st} \text{ child of position } q \\ d \cdot f(q) + d, p - d^{s} \text{ child of position } q \end{cases}$$



index of j^{th} child for node with i^{th} index

```
Child(i, j):
    return d·i + j
```

- i – index of some node n_0
- j – j^{th} child of n_0

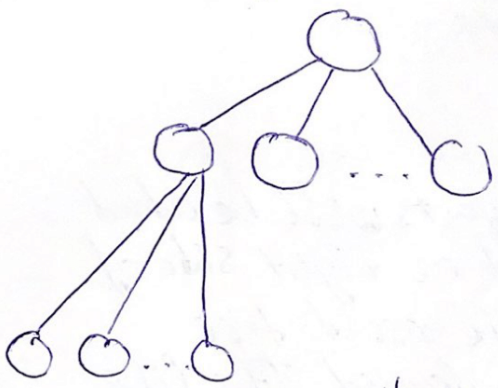
$j \in [1; d]$

```
Parent(i):
    return ⌊(i-1)/d⌋
```

- i – index of some node

returns index of its parent node

2. Height evaluation:



- On levels from 0 to h-1 should be:

$1 + d + d^2 + \ldots + d^{h-1}$ elements.

$$\sum_{k=1}^{h} d^{k-1} = \frac{d^h - 1}{d - 1}$$

- On the last (h) level could be from 1 to d^h elements.

So, $\frac{d^h - 1}{d - 1} + 1 \leq n \leq \frac{d^h - 1}{d - 1} + d^h$

$$d^h - 1 + d - 1 \leq n(d-1) \leq d^h - 1 + d^{h+1} - d^h$$

$$(n(d-1)+1)\frac{1}{d} \leq d^h \leq n(d-1) - d + 2$$

① $\log_d(n(d-1)+1) - 1 \leq h \leq \log_d(n(d-1)+1+1-d)$ ②

As $d \geq 2$, than $\log_d(n(d-1)+1+1-d) < \log_d(n(d-1)+1)$

That means, that

$$h = \lceil \log_d(n(d-1)+1) - 1 \rceil$$

To continue evaluation:

1. $\log_d(n(d-1)+1) - 1 > \log_d(n(d-1)) - 1$

$$\log_d n + \underbrace{\log_d(d-1)}_{>-1} - 1$$

$$\Leftarrow$$

Left part ① $> \log_d n - 1$

2. $\log_d(n(d-1)+2-d) \leq \log_d(n(d-1))$

$$\log_d n + \underbrace{\log_d(d-1)}_{<1} < \log_d n + 1$$

$$\Leftarrow$$

Right part ② $< \log_d n + 1$

3. From 1. & 2.:

$$\log_d n - 1 < h < \log_d n + 1$$

$$\Leftarrow$$

$$h = \Theta(\log_d n)$$

3. extractMax

A - array, representing d-ary Heap

⊙ extractMax(A):

```
1. if A.heap-size < 1
2.       error "heap is empty"
3. max = A[0]
4. A[0] = A[A.heap_size-1]  // A[A.heap_size-1] - last element inside the heap (the most right on the last level)
5. A.heap_size = A.heap_size - 1
6. maxHeapify(A,0)
7. return max
```

⊙ maxHeapify.(A,i):

children could be found by index i, using formula from (1)

```
1. largest = i
2. for each child of node with index i:
3.     if child.value > value of node with index largest:
4.         largest = child.index
5. if largest ≠ i:
6.     swap A[i] with A[largest]
7.     maxHeapify(A, largest)
```

Worst-case: Running time of extractMax depends on maxHeapify function.

- On each step maxHeapify ~~goes~~ walk in worst-case through d children of node with index i.
- And worst-case maxHeapify will be called on each level till the height h of a heap.

$$T(n) = O(dh) = O(d \log_d n)$$

4. insert

A - array representation of d-ary heap

- insert (A, key, d):
  1. A.heap_size = A.heap_size + 1
  2. i = A.heap_size - 1 // i - index of the last element in heap
  3. A[i] = key
  4. while i > 0 and A[parent(i,d)] < A[i]:
  5. swap A[i] with A[parent(i,d)]
  6. i = parent(i,d)

- parent (i,d):
  1. return $\lfloor (i-1)/d \rfloor$

Worst-case: In worst-case element from the bottom will rise up to the root. So, it will make h steps.

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

5. increaseKey

A - array representation of d-ary heap
i - index of element, which key should be increase

- increaseKey (A, i, key, d):
  1. if key < A[i]:
  2. error "new key is smaller than current one"
  3. A[i] = key
  4. while i > 0 and A[parent(i,d)] < A[i]:
  5. swap A[i] with A[parent(i,d)]
  6. i = parent(i,d)

- parent (i,d):
  1. return $\lfloor (i-1)/d \rfloor$

Worst-case: In worst-case i = A.heap_size - 1, so it's the last element of heap. In this case element will make h steps on his way.

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

Task 3.4

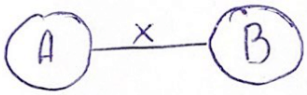
1. For graph $G=(V,E)$ let's prove that it's minimum spanning tree (MST) is a bottleneck spanning tree (BST) as well.

By contradiction:

Let's consider graph $G=(V,E)$ with MST T.

T consists of two connected components A & B. $V_A \cup V_B = V$. V_A & V_B - disjoint sets.

T:



Let A & B be connected by edge $x \in T$.
x has max weight from all edges in T

- Let's say that T is not a BST. So, there exists another edge x' with weight less than x ($\omega_{x'} < \omega_x$), which $\notin T$ now.
- x' should $\in A$ or B, but it can't connect A with B. Because $\omega_{x'} < \omega_x$, if it has been connecting A with B, than it should be instead of x in T.
- x' connects to vertices u and v, which were already connected in T. So, adding x' created a cycle, which should be deleted by deleting one of edges in path between u and v (except x')
- Let's call new tree - T'.
- Now to convert T' into a BST x should be changed to some other

edge to connect A with B.
But x was a part of T, so it has minimum possible weight from all edges connecting A with B.

- If we will change it, than x'- won't be edge with max weight.
- If x should stay, than it becomes a bottleneck edge.

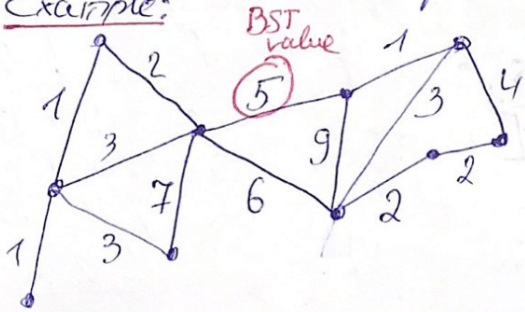
x was a part of T., than T was a BST too.
Got a contradiction.
That means that MST is BST.

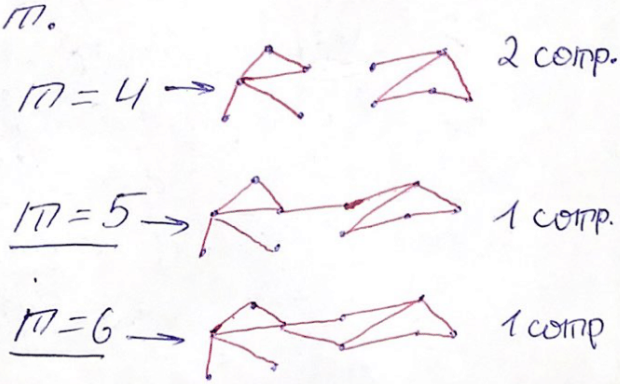
Q.E.D.

(2.) Algorithm:

1. Walk through all edges and store only those with value of weight less than or equal to m.
2. Check new graph, consisting of saved edges and same vertices, for connectivity using DFS (depth-first search).
3. If graph has only 1 connected component, than bottleneck spanning tree value is less than or equal to m.

Example:





- checkBottle (E, m):

```
1.  A = new array list of size=number of vertices
2.  visited = boolean array of size = number of vertices
3.  for each edge in E:
4.      if (edge.weight <=m):
5.          A[edge.v1.index].add(edge.v2)
6.          A[edge.v2.index].add(edge.v1)

7.  DFS(0, A, visited)
8.  for each element in visited:
9.      if (!element):
10.         return "Bottleneck Spanning Tree value
                    larger than m"
11. return "Bottleneck Spanning Tree value lesser than
            or equal to m"
```

A - ~~test~~ adjacency list for new subgraph (array list of array lists)

E - edge list of graph

m - given number

- DFS(start, A, v):

```
1.  v[start] = true
2.  for i from 0 to A[start].size-1:
3.      neighbour = A[start][i].index
4.      if (v[neighbour]=false):
5.          DFS(neighbour, A, v)
```

In terms of number of edges |E|, DFS algorithm and checkBottle have linear-time complexity

3. Algorithm (Camerini's algorithm)

1. Find a median edge weight of the graph and split all edges in two sets E_1 and E_2.

   E_1 consists of all edges with weights less than or equal to median edge weight

   E_2 consists of all edges with weights greater than median edge weight

2. Check E_1 on connectivity, using DFS. If it is connected, run step 1, else run step 3.
3. Contract each connected component in to single ~~vertex~~ vertices. And run Algorithm again with those vertices and edges from E_2. The result is union of edges from E_1 and new edges found.

```
BST(V, E, A'):
1.    if(|E|=1):
2.        return E
3.    else:
4.        A = new array list of size = number of vertices
5.        visited = boolean array of size = number of vertices
6.        E1 = new array list
7.        E2 = new array list
8.        m = median(E)
9.        for each edge of E:
10.           if (edge.weight > m):
11.               E2.add(edge)
12.           else:
13.               E1.add(edge)
```

```
14.             A[edge.v1.index].add(edge.v2)
15.             A[edge.v2.index].add(edge.v1)

16.     DFS(0, A, visited)
17.     count = 0
18.     for each element of visited:
19.         if (element):
20.             count += 1
21.     if (count = visited.size and A.size = V.size)
22.         return BST(V, E1, A')
23.     else:
24.         V' = new array list
25.         for each edge from E1:
26.             V'.add(contract(edge, V, A'))
27.         return A + BST(V', E2, A')
```

- median(E):

```
1.      quickMedian(E, |E|/2)
```

- quickMedian(E, l)

```
1.      if |E| = 1
2.          return E[0]
3.      index = random(l)
4.      pivot = E[index]
5.      lows, highs, pivots = new array lists
6.      for each edge in E:
7.          if (edge.weight < pivot):
8.              lows.add(edge)
9.          elseif (edge.weight > pivot):
10.             highs.add(edge)
11.         else: pivots.add(edge)
```

```
12.     if(l < lows.size):
13.         return quickMedian(lows, l)
14.     else if (l < (lows.size + pivots.size)):
15.         return pivots[0]
16.     else:
17.         return quickMedian(highs, l - lows.size - pivots.size)
```

6. contract(edge, V, A')

```
1.  v1 = edge.v1.index
2.  v2 = edge.v2.index
3.  for each vertex A'[v2]:
4.      A'[v1].add(vertex)
5.  edge.v2.index = edge.v1.index
6.  V.size = V.size - 1
```

quickMedian expected time complexity:

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots = O(n)$$

In terms of edges

Time complexity of full algorithm:

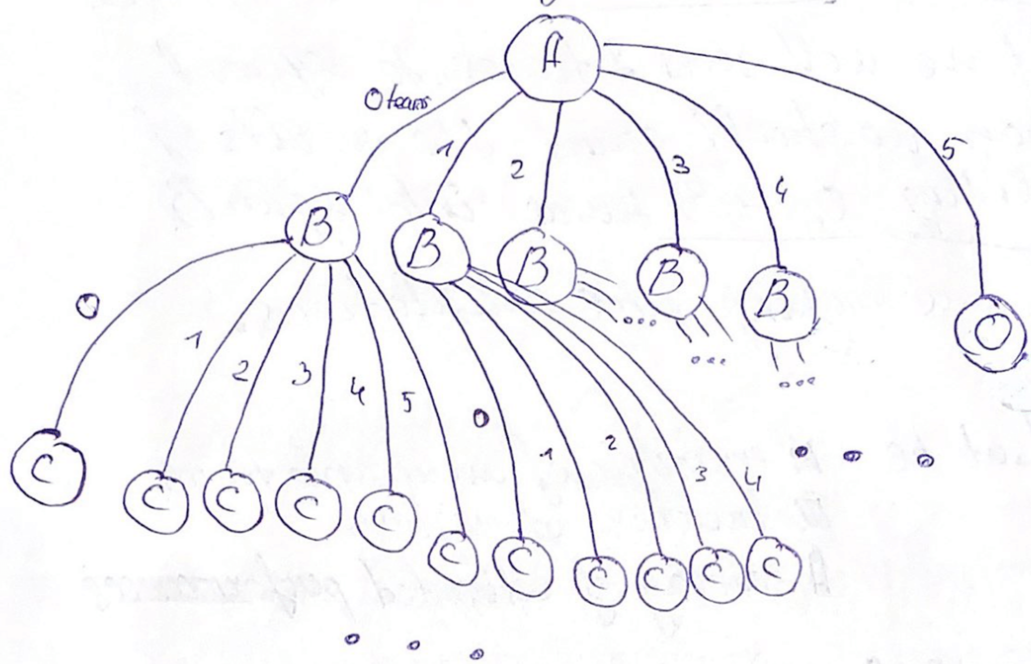
$$O(|E|)$$

Task 3.5.

1.

1.1. For each region should be considered how many teams will be send by using recursion.

In example of 5 teams and 3 regions



On each recursive call as input of algorithm number of availible teams and considerable regions will be taken.

So if from n teams on first step to first region (from m) were sent t teams, than on next step there'll be (n-t) available teams and (m-1) considerable regions.

For each possible t, should be counted value of to measure performance and added to the result of sending (n-t) teams to (m-1) regions and etc. Maximum value of all is the result of algorithm.

1.2. Overlapping subproblems

- If we will send 1 team to region A, than we should count the results of sending 0, 1, 2, 3, 4 teams into region B.
- If we will send 2 teams to region A, than we should count the results of sending 0, 1, 2, 3 teams into region B.

We are making same calculations.

1.3.

Let be M - number of available teams
R - number of regions
E - array of estimate

Times	Cost
1	C_1
$R+1$	C_2
$R(M+2)$	C_3
$R(M+1)$	C_4
$R+M$	C_5
—	0
$RM-M$	C_6
$\sum_{i=0}^{R-1}\sum_{j=0}^{M}\sum_{k=0}^{j} 1$	C_7
$\approx RM^2$	C_8
$\approx RM^2$	C_9
RM book 1.4	C_{10}

```
NHC(E,M,R):
1.  T = new array((M+1) x R)
2.  for i from 0 to R-1:
3.      for j from 0 to M:
4.          if i=0 or j=0:
5.              T[j][i] = E[j][i]
6.          else:
7.              max = T[j-1][i]
8.              from k from 0 to j:
9.                  if(T[k][i-1]+E[j-k][i] > max):
10.                     max = T[k][i-1]+E[j-k][i]
11.             T[j][i] = max
12. return T[M][R-1]
```

(1.4) Asymptotic worst-case time complexity

$$\sum_{i=0}^{R-1}\sum_{j=0}^{M}\sum_{k=0}^{j} 1 = \sum_{i=0}^{R-1}\sum_{j=0}^{M}(j+1) = \sum_{i=0}^{R-1}\frac{(1+M+1)(M+1)}{2} =$$

$$= R\left(\frac{(M+2)(M+1)}{2}\right) < c \cdot RM^2$$

From table from 1.3.

$$T(n) = C_1 + C_2(R+1) + C_3R(M+2) + C_4R(M+1) + C_5(R+M) + C_6R + C_7RM^2 + C_8RM^2 + C_9RM^2 + C_{10}RM < C_{11} \cdot RM^2$$

$$T(n) = O(RM^2)$$

Time complexity: $O(RM^2)$

(2.)

Number of teams	Region A	Region B	Region C
0	0	0	0
1 team	45(1A)	50(1B)	50(1B)
2 teams	70(2A)	95(1A1B)	95(1A1B)
3 teams	90(3A)	120(2A1B)	120(2A1B)
4 teams	105(4A)	140(2A2B)	140(2A2B)
5 teams	110(5A)	160(3A2B)	170(1A1B3C)

Answer: 170