DSA - Seminar 5

1. Consider the following problem: Determine the sum of the largest *k* elements from a vector containing *n* distinct numbers. For example, if the array contains the following 10 elements [6, 12, 9, 91, 3, 5, 25, 81, 11, 23] and k = 3, the result should be: 91 + 81 + 25 = 197.

   I. Find the maximum k times (especially good if k is small)
   - If we just call the maximum function 3 times for our example, it will return 91 each time, so we need a solution where we also have an upper bound, and we are searching for the maximum which is less than that value.
   - First, maximum is 91. At the second call we want the maximum which is less than 91, we will get 81.
   - At the third call we want the maximum which is less than 81, we will get 25.
   - Complexity of the approach: $\Theta(k*n)$ – finding the maximum is $\Theta(n)$ and we do this k times.

   II. Similarly, we could do just k iterations of some sorting algorithms:
   - Selection sort (with descending sorting) finds the maximum and moves it to position 1. Then it finds the maximum of the remaining array and moves it to the second positions, etc. After k iterations the first k elements will be the ones we need to add together.
   - Bubble sort (with ascending sorting) – one iteration of bubble sort will move the maximum element to the end of the array. The next iteration will guarantee that the element on position n-1 is the second largest, etc. After k iterations, the last k elements will be the ones we need to sum up.
   - Complexity for both cases is $\Theta(k*n)$

   III. Sort the array in a descending order and pick the first k elements (especially good is k is large).
   - Sorting can be done in $\Theta(n*\log_2 n)$ time
   - Computing the sum of the first k elements: $\Theta(k)$
   - In total $\Theta(n*\log_2 n) + \Theta(k) \in \Theta(n*\log_2 n)$

   IV. Keep a sorted array of the k largest elements found so far. Initially this array will contain the first k elements (in sorted order). For the next elements of the input array, we compare the element with the minimum of the k selected numbers and if necessary remove the minimum and add the new element. For our example:
   - Initially the array contains [12, 9, 6] (I chose to keep them in descending order, but it would be the same with ascending ordering).
   - We process 91, it is greater than 6 (the minimum of the selected k elements), so we remove 6 and add 91. Our array of selected elements will be: [91, 12, 9]

- We process 3, it is less than 9 (the minimum of the selected k elements), so we do not modify the array
- We process 5, it is less than 9 (the minimum of the selected k elements), so we do not modify the array
- We process 25, it is greater than 9 (the minimum of the selected k elements) so we remove 9 and add 25. Our array of selected elements will be: [91, 25, 12].
- Etc.
- At the end, we need to compute the sum of the elements in k (or we can keep a variable that gets updated while we process the elements).
- Complexity:
    i. Processing one element can have a best case (element is less than the minimum, we just do a comparison: $\Theta(1)$) or a worst case (element is a new maximum, we need to move every element in the array one position to the right: $\Theta(k)$) => $O(k)$.
    ii. We need to process every element => $O(n*k)$

V. Use a binary max-heap. Add all the elements to the heap and remove the first k.

- Adding an element to a heap with n elements is $O(\log_2 n)$.
- Removing an element from a heap with n elements is $O(\log_2 n)$.
- In total we have $O(n*\log_2 n) + O(k*\log_2 n)$. Since n >= k, this is $O(n*\log_2 n)$

```
function sumOfK(elems, n, k) is:
//elems is an array of unique integer numbers
//n is the number of elements from elems
//k is the number of elements we want to sum up. Assume k <= n
       init(h, "≥") //assume we have the Heap data structure implemented. We
       initialize a heap with the relation "≥" (a max-heap)
       for i ← 1, n execute
              add(h, elems[i]) //add operation was discussed at Lecture 8
       end-for
       sum ← 0
       for i ← 1, k execute
              elem ← remove(h) //remove operation was discussed at Lecture 8
              sum ← sum + elem
       end-for
       sumOfK ← sum
end-function
```

VI. How can we reduce the complexity? Well, we do not need all the elements in the heap, we are always interested in the k largest ones (similar to solution IV, but instead of sorted array of k elements, we will have heap of k elements). If we consider the example from above, we can work in the following way, always keeping just the k maximum elements up until now:

- Initially we keep 6, 12, 9
- When we get to 91, we can drop 6, because we know for sure that it is not going to be part of the 3 maximum numbers (we already have 3 numbers greater than this). So we keep 12, 91, 9.

- When we get to 3, we know it is not going to be part of the 3 maximum elements (We already have 3 elements greater than that). Similar with 5.
- When we get to 25, we can drop 9, and go on with 12, 91, 25.
- Etc.

We can keep the k elements that we consider in a heap. Should it be a min-heap or max-heap?

When we have the k largest elements at a given point, we will be interested in the minimum of these elements (this is what we compare to the new element that is considered and this is what we remove if we find a larger one) so it should be a min-heap.

```
function sumOfK2(elems, n, k) is:
//elems is an array of unique integer numbers
//n is the number of elements from elems
//k is the number of elements we want to sum up. Assume k <= n
       init(h, "≤") //assume we have the Heap data structure implemented. We
       initialize a heap with the relation "≤" (a min-heap)
       for i ← 1, k execute //the first k elements are added "by default"
              add(h, elems[i])
       end-for
       for i ← k+1, n execute
              if elems[i] > getFirst(h) then //getFirst is an operation which returns
the first element from the heap.
                     remove(h)//it returns the removed element, but we do not need it
                     add(h, elems[i])
              end-if
       sum ← 0
       for i ← 1, k execute
              elem ← remove(h) //remove operation was discussed at Lecture 8
              sum ← sum + elem
       end-for
       sumOfK2 ← sum
end-function
```

- Complexity? Our heap has maximum k elements, so operations have a complexity of $O(\log_2 k)$. We call add at most n times (worst case, when every element is greater than the root of the heap) and remove at most n times. So in total we have $O(n*\log_2 k)$
- If you do not use an already implemented heap, but have access to the representation, you can make the previous implementation slightly more efficient (complexity will not change though):
  - the last *for* will not have to remove elements, just simply to add up the sum of the elements from the heap array (but for this you need access to the array) – You can eliminate that for even if you don't have access to the representation, if you keep a variable with the sum so far: whenever you add to the heap, you add to the sum, whenever you remove from the heap you subtract from the sum.
  - the middle for loop will not have to do a remove and an add. You can just overwrite the element from position 1 (this is what would be removed anyway) with the newly added element and do a bubble-down on it.

VII. Can we improve complexity even more? We know that if we have an array we can transform it into a heap in a complexity O(n) – it was discussed at heapsort. So we can

do something similar to version V, but instead of adding the elements one by one to the heap we could transform the array into a max-heap and then remove k elements from it. This approach has a complexity of $O(n + k*\log_2 n)$

2. Evaluate an arithmetic expression which contains single digit operands, parantheses and the +, -, *,/ operators. We assume that the expression is correct. For example:
    - 2+3*4 = 14
    - ((2+4)*7)+3*(9-5) = 54
    - 2*(4+3)-4+6/2*(1+2*7)+4 = 59

The expressions from the above example are in the so called *infix* notation. This means that operator are between the two operands (for example: 2+4). This is how we work with arithmetic expressions in general.

For a computer it is a lot easier to work in the *postfix* notation. This is a notation in which the operator comes after the two operands (for example: 2 4 +).

A few examples of expressions in the infix notation and their corresponding postfix notation:

2 + 4	2 4 +
4*3+6	4 3 * 6 +
4*(3+6)	4 3 6 + *
(5 + 6) * (4 – 1)	5 6 + 4 1 - *

Obs:

- Relative order of the operands stays the same
- Relative order of the operators might change.
- We no longer have parentheses in the postfix notation

So evaluating an arithmetic expression will have two steps:

- Transform the expression in the corresponding postfix notation
- Evaluate the expression on the postfix notation

1. Transform an infix expression in the corresponding postfix notation. Input is the infix expression, output will be the postfix notation in the form of a queue and for the transformation we use an auxiliary stack.
   Steps:
- Start parsing the expression.
- If you find an operand => push it to the queue

- If you find an open paranthesis => push it to the stack
- If you find a closed paranthesis => open paranthesis should be on the stack already, so pop eveything from the stack (and whenever you pop something, push it to the queue) until you get to the first open paranthesis. Do not push the open or closed paranthesis to the queue.
- If you find an operator:
    - As long as the stack is not empty and the top element of the stack is an operator with a priority greater than or equal to the priority of the current operator, pop from stack and push to the queue.
    - Push operator to the stack
- When the expression is over, pop whatever you have left on the stack and push it to the queue.

Example: 2*(4+3)-4+6/2*(1+2*7)+4

Current element	What to do	Stack (top is on the right)	Queue
2	Push to queue		2
*	Push to stack	*	
(	Push to stack	* (	
4	Push to queue		2 4
+	Push to stack	* (+	
3	Push to queue		2 4 3
)	Pop from stack and push to queue until you find the open paranthesis	*	2 4 3 +
-	Pop from stack as long as top has greater or equal priority.Push - to the stack	-	2 4 3 + *
4	Push to queue		2 4 3 + * 4
+	Pop from stack as long as top has greater or equal priority. Push + to stack	+	2 4 3 + * 4 -
6	Push to queue		2 4 3 + * 4 – 6
/	Push to stack	+ /	
2	Push to queue		2 4 3 + * 4 – 6 2
*	Pop from stack as long as top has greater or equal priority. Push * to stack	+ *	2 4 3 + * 4 – 6 2 /
(	Push to stack	+ * (	
1	Push to queue		2 4 3 + * 4 – 6 2 / 1
+	Push to stack	+ * (+	
2	Push to queue		2 4 3 + * 4 – 6 2 / 1 2
*	Push to stack	+ * (+ *	
7	Push to queue		2 4 3 + * 4 – 6 2 / 1 2 7
)	Pop from stack and push to queue until you find the open parenthesis	+ *	2 4 3 + * 4 – 6 2 / 1 2 7 * +
+	Pop from stack as long as top has greater or equal priority. Push + to the stack	+	2 4 3 + * 4 – 6 2 / 1 2 7 * + * +
4	Push to queue		2 4 3 + * 4 – 6 2 / 1 2 7 * + * + 4

	Pop whatever we have in the stack and push to queue		2 4 3 + * 4 – 6 2 / 1 2 7 * + * + 4 +

Pseudocode implementation:

Assume stack and queue is already imeplemented and they have the standard operations: init, push, pop, top, isEmpty.

```
function transform(expression) is:
  //create stack and queue
  init(st)
  init(q)
  for e in expression execute:
    if e is operand then
      push(q, e)
    else if e is '(' then
      push(st, e)
    else if e is ')' then
      while top(st) != '(' execute
        elem <- pop(st)
        push(q, elem)
      end-while
      pop(st) //to remove the (
    else //e is operand
      while (not isEmpty(st)) AND (top(st) != '(') AND
            (top(st) has higher priority than e) execute
        elem <- pop(st)
        push(q, elem)
      end-while
      push(st, e)
    end-if
  end-while
  while not isEmpty(st) execute:
    elem <- pop(st)
    push(q, elem)
  end-while
  transform <- q
end-function
```

2. Evaluate the postfix expression. Input is the postfix notation in the form of a queue (actually the output of step 1) and the output will be a value: the result of the evaluation. In the process we use an auxiliary stack.

- Start parsing the expression from the queue.
- If you find an operand => push to the stack
- If you find an operator (this is postfix notation, operators are after operands, so the operands have to be in the stack already) => pop two elements from the stack (these are the operands for this operator), perform the operation and push the result back to the stack
- When the queue is empty, the stack contains one single element, this is the result

Current elem from queue	What to do	Stack (top is on the right)
2	Push to stack	2
4	Push to stack	2 4
3	Push to stack	2 4 3
+	Pop 2 elements, perform +, push result to stack	2 7
*	Pop 2 elements, perform *, push result to stack	14
4	Push to stack	14 4
-	Pop 2 elements, perform -, push result to stack	10
6	Push to stack	10 6
2	Push to stack	10 6 2
/	Pop 2 elements, perform /, push result to stack	10 3
1	Push to stack	10 3 1
2	Push to stack	10 3 1 2
7	Push to stack	10 3 1 2 7
*	Pop 2 elements, perform *, push result to stack	10 3 1 14
+	Pop 2 elements, perform +. push result to stack	10 3 15
*	Pop 2 elements, perform *, push result to stack	10 45
+	Pop 2 elements, perform +, push result to stack	55
4	Push to stack	55 4
+	Pop 2 elements, perform +, push result to stack	59

Pseudocode implementation

```
function evaluate(postfix) is:
  init(st)
  while not isEmpty(postfix) execute:
    elem <- pop(postfix)
    if elem is operand then
      push (st, elem)
    else
      op1 <- pop(st)
      op2 <- pop(st)
      res<- @compute the result of op2 elem op1
      push(st, res)
    end-if
  end-while
  result <- pop(st)
  evaluate <- result
end-function
```

3. Problem taken from: https://open.kattis.com/problems/chewbacca

You are given a tree of out-degree K with N nodes or, in other words, each node can have at most K children. The tree is constructed so it is of the "lowest energy": the nodes are placed in a new depth of the tree only when all the places (from left to right) in the previous depth have been filled. This is also the order of enumerating the nodes, starting with 1. Figure 1 depicts an example of a tree of order 3 with 9 nodes.

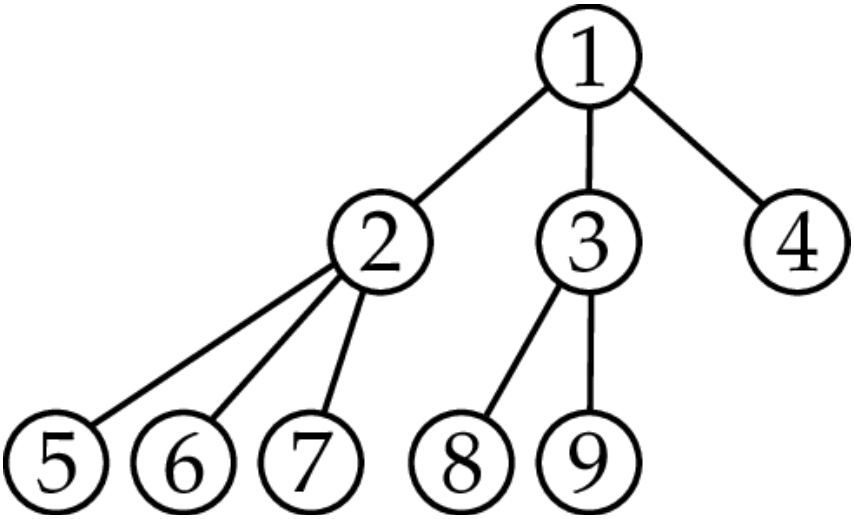


Figure 1: Sample Input 2

Given two nodes, x and y (x != y, both valid numbers in the tree) determine the minimum number of steps to get from node x to y. For the above example:
minDist(5, 9) = 4
minDist(5, 3) = 3
minDist(5, 7) = 2
minDist(7, 2) = 1

If we look at how the tree is built, we can see that it is the same principle that we use for a binary heap (filling nodes from left to right and going to the next level only when the current one is full), but here, instead of having 2 children, every node will have K children (so it is like a K-ary heap, instead of binary heap).

All those *rules* that we use to navigate in a binary heap (going from a node to its children and going from a child to its parent) can be adapted to this case as well.
For this problem it is actually enough to go from a node to its parent. Having the two nodes, we can go up in the tree (and count how many steps we make) until we find the first common parent (technically called Least Common Ancestor).

So how can we go from a node to its parent?
For a binary heap, parent of node p was on position p/2.
If we look at the heap from the figure, we can see that parent of node p is on position (p+1)/3.
If we draw a heap with K = 4, and look at the nodes, we will see that the parent of node p is on position (p+2)/4

Let's see how we can find a general expression. Intuitively we know that we will have to divide by K.

Let's see the form of the children of a given node:

- Root has the number 1
- Children of node 1 will go from 2 to K + 1 (this is how we get K children)
- Then children of node 2 will go from K+2 to 2K + 1
- Then children of node 3 will go from 2K + 1 to 3K + 1

We can see that for a node X, its children will be from (X-1)* K + 2 to X*K + 1 (Obs. You can easily check this rule for K = 2 – the regular binary heap – and K = 3 - the example from Figure 1.)
And we want a formula so that when we divide the child's number by K, the result should be X. So we should make the "smallest" node (X-1)*K + 2) to be X*K => we can do this by adding to it K – 2.
So, for the "smallest" (leftmost) child, its parent will be:
((X-1) * K + 2 + K – 2) / K => (X*K – K + 2 + K – 2)/ K = X – good
If we use the same rule, for the "greatest" (rightmost) child, its parent will be:
(X * K + 1 + K – 2) / K => (K * (X + 1) – 1) / K => this is less than K * (X+1) / K (which would be X+1), so the result (since we are doing integer division) is X.

So now we know (and we have proof that this is correct) how to go from a node Y to its parent: (Y + K – 2) / K

So now we need to figure out how to find the common parent. From the example, we can see that it might happen that one of the nodes is already the common parent (ex: 7 and 2, 2 is the common parent). We might have to go up the same number of steps for both nodes (ex: 5, 9), but we might need to make more steps with one node than with the other (ex: 5, 3).

So we cannot do something like:

```
while x != y execute:
      x <- parent of x
      y <- parent of y
      steps <- steps + 2
end-while
```

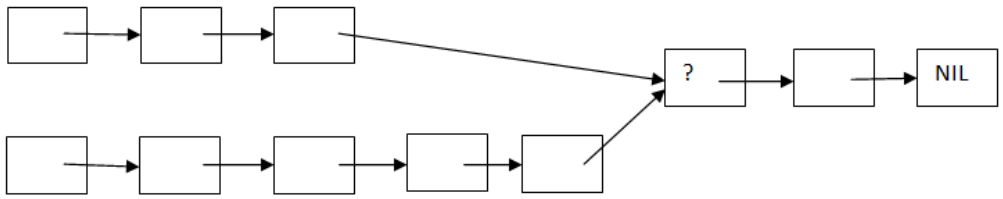
It is a lot better to change just one node at a time, but which one?
We can see that if x > y, it means that either x is at a lower level than y or they are on the same level, but x is to the right of y. So, if we just change one node, we should change the one with the greatest number.

```
function minDist(N, K, x, y) is:
      distance <- 0
      while x != y execute:
            if x > y then
                  x <- (x + K – 2) / K
                  distance <- distance + 1
```

```
            else
                  y <- (y + K - 2) / K
                  distance <- distance + 1
            end-if
      end-while
      minDist <- distance
end-function
```

The fact that the nodes are numbered allows us to know which node to change. But what if we did not know this? What if we had a function to return the parent of a node, but the nodes were not numbered ascendingly?

This is the same as the following problem: we have two singly linked lists, which at a given point merge into the same list (see the figure). You only have the first node of the two lists, nothing else. How would you find the first common node?



(If you rotate the figure with 90 degrees counterclockwise, you will have the same situation as we have in case of the heap, considering that we only have the paths from nodes x and y to the root, ignoring the remaining nodes).
One solution is to go through one list, and for every node of that list, check if that node appears in the second. The moment you find the first such node, you have the common node. But this is inefficient from a time complexity perspective.
A more efficient solution (from the time complexity perspective, because it will require some extra memory) is to take two stacks: in one we will add the nodes from the first list (from the first to the last) and in the other the nodes from the second list. Both stacks will have on top the common node(s). So we just have to remove in parallel from them until we get the last common node.
This idea can be translated to our heap problem as well:

- In a stack we will put the “path” from node x to the root. For example: 5, 2, 1.
- In another stack we will put the “path” from node y to the root: For example: 3, 1
- Top of the stack contains the common nodes, we have to eliminate those and then count how many nodes we have left in the two stacks. This is the minimum distances between nodes x and y.