# IN3130 Exercise set 11b

## Exercise 1

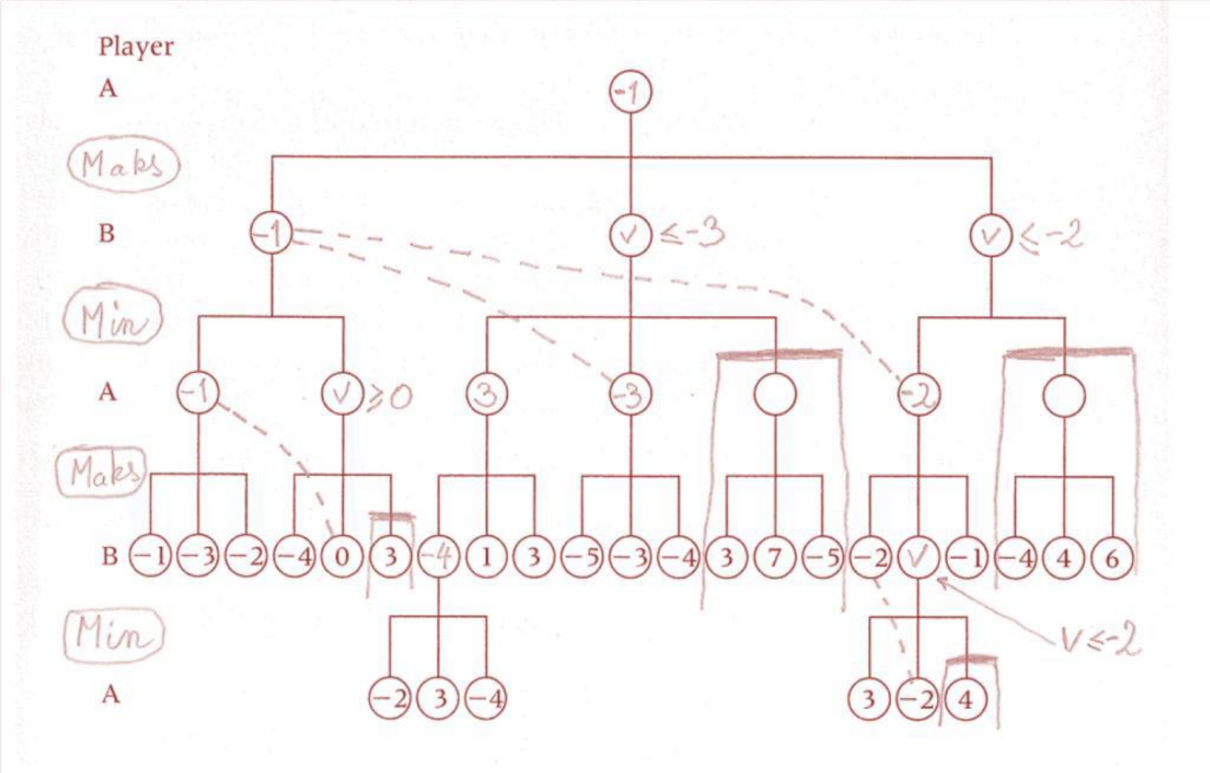
Study figures 23.13 and 23.14 (pages 735 and 736 in the textbook) where -1 and +1 is used to indicate win and loss, respectively. Look at all nodes and make sure you understand how values for the internal nodes are calculated with the min/max-algorithm. Always keep in mind that values indicate the situation for the player with the opening move – player A. For B smaller values are better. (Note also that in exercise 3 below we negate the values on every level so that we can always maximize!).

Left to the group.

## Exercise 2

Study figures 23.16 and 23.17 (pages 738 and 740 in the textbook) and check that your understanding of alphabeta-pruning is correct; then solve exercise 23.22 in the textbook.

See figure below. Alpha and beta values are not written inside the node, the real node value is indicated instead, so that it is easy to see where we get cutoff. A dotted line is drawn between the values that cause the cutoff.



## Exercise 3

Go through the program on page 741 and discuss the solution chosen there, where values are negated on every level. Note that there are some misprints in the program. First, in both lines following an "else" the name of the called procedure should be "ABNodeValue". Second, a right parenthesis is missing at the end of the last of these lines. Finally, it should be "-∞" in the outermost call. See slides!

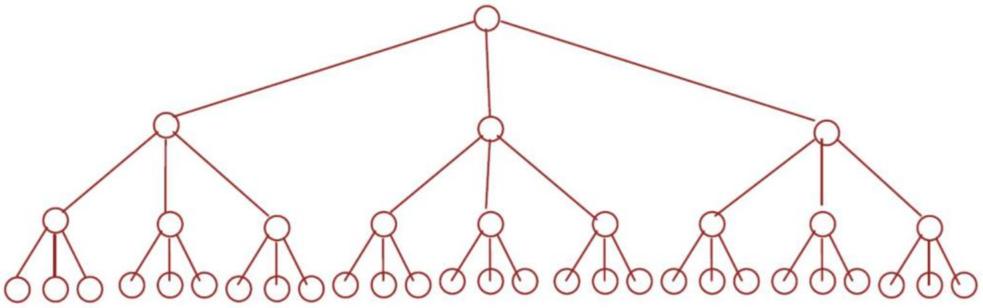
The first part is left to the group session. The second part of the question is a bit misleading, since we in an alpha-beta-cutoff do not find the best move to play for every node. The analysis is done on behalf of the present root in the tree, and if the result is of no interest for the root, we get a cutoff. The variable should have been named `bestMoveSeen`, and that is what we use below. It is then important that seen from the root it really is the best move (for the player with the move) we get if we follow `bestMoveSeen`. This is because we stop iterating through the children of node X when we realize that the parent of X would not choose X's move as its best. Assigning alpha and beta values is straight forward.

```
real function ABNodeValue(
X, // The node we calculate alpha/beta-values for, children: C[1],C[2],…,C[k]
numLev,     // Number of levels left
parentVal) // Alpha/beta-values from parent (-LB from parent)
// returns: Final alpha/beta-values for node X
{
  real LB; // LowerBound for alpha/beta-values for this node.
  real lastLB;
  if <X er terminal-state> then return <value of X for player with X-move>;
  else if numLev = 0 then return <estimated value of X for palyer with X-move>;
  else {
    LB := - ABNodeValue(C[1], NumLev-1, ∞);
    X.bestMoveSeen := C[1]; X.value = LB; // NEW!
    for i := 2 to k do {
      if LB >= parentValue then return LB
      else {
        lastLB := LB; LB := max(LB, -ABNodeVal(C[i], Numlev-1, -LB) );
        if lastLB < LB then {X.bestMoveSeen := C[i]; X.value := LB;} // NEW!
      }
    }
  }
  return LB;
}
```

We invoke by calling `ABNodeValue(root, 10, - ∞)`

## Exercise 4

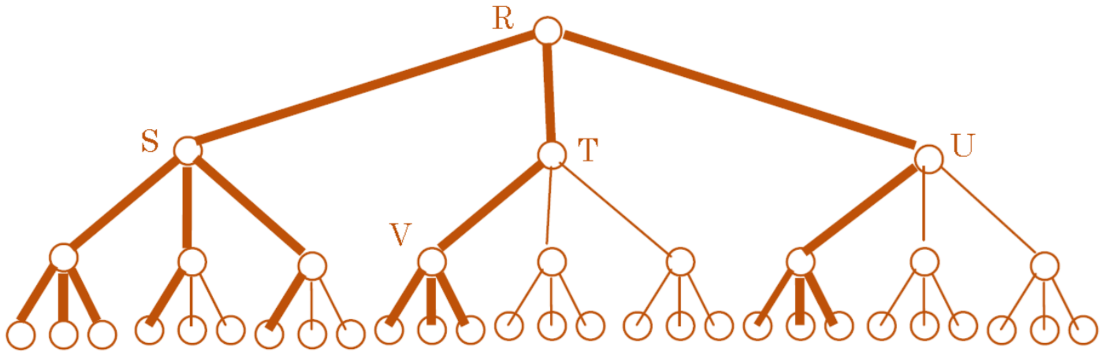
If we, in each situation in an alpha-beta-search, are lucky enough to always look at the best move (for the player to make the move) first we will get good pruning. One can even prove that if we go down to depth *d*, with branching factor of *b*, the search time with alpha/beta-pruning will be $O\left(\underbrace{b \cdot 1 \cdot b \cdot 1 \cdot b \ldots}_{d \text{ factors}}\right) = b^{\frac{d}{2}}$ instead of $O\left(\underbrace{b \cdot b \cdot b \ldots}_{d \text{ factors}}\right) = b^{d}$ . This means that if you without pruning could reach to depth n in a certain time period, you can in the same period reach to about depth 2**n* using alpha/beta pruning. We shall not attempt to prove this, but instead look at a concrete example. We let *d* = 3, and *b* = 3, and get the tree below. Assume that the best move is always the one drawn to the left in the figure below and that we look at the subtrees from left to right for each node. Mark the branches you will have to evaluate (and thereby the ones you can avoid). The tree has 39 edges, how many do you avoid looking at?



**EXTRA question:** Assume you are less lucky than above, and always look at the best move last. Will you then get any pruning at all?

Note that what we try to do is always look at the best move for the player with the move. We do, of course, not know what move it is (this would make the analysis too easy!), instead we have to assume we have an heuristic that gives us the move, and run the algorithm. What we study is how algorithm behaves if we are lucky in our choice of heuristic.

Below is the tree with bold edges where the algorithm must descend. As an example we look at nodes S, T and U. We assume that A has the highest value (the highest value of the nodes S, T and U, since we maximize in the the root. We further assume that move V is the best possible from situation T. That is, the value returned from V to T is so low that we see that the T-value is so small (remember that we minimize in T) that is can not compete with the S-value when we maximize in R. And that we therefore need not look at the two remaining branches in T. And similarly for S.



Of the 39 edges we need only look at 19, and the number of leaves we look at is 11 out of 27. the number of leaves is important as we (at least in chess) most likely have to do a quite heavy analysis of our position to give it a score (with a suitable heuristic).

The answer to the extra question is that you *can* get some pruning, if, for instance, the next best move comes first and there are at least three children.

## Exercise 5

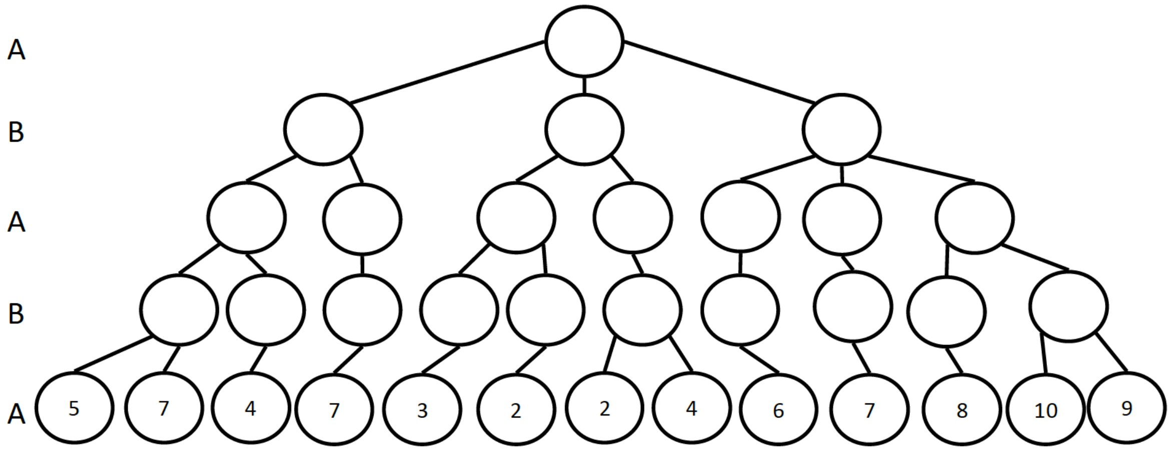
(Can be skipped, as it is not central to the course) Assume you are playing the game of NIM, with two piles, and that it is your move, that no pile is empty, and that the piles are of different sizes (number of sticks or pebbles, or whatever). Try to come up with a strategy that guarantees victory.

The idea is to make sure the opponent always finds himself in a situation where both piles are equal. You keep doing this as long as the smallest pile contains at least two pebbles. This way the opponent always has equal piles when he makes his move. If he makes a move that leaves at least two pebbles in one pile, we play another round; otherwise we are in a situation with two or more pebbles in one pile and one or zero in the other. If there is 1 in the smallest pile, you take the whole of the larger pile; if there is zero in the smallest pile, you take all but one pebble from other. In both cases the opponent must make the last move, and lose.

## Exercise 2 (From Exam 2010)

We shall look at game trees, and we assume that the root node of the tree in the figure below is representing the current situation of a game (that we do not describe further), and that it is player A's turn to move. The other player is B, and A and B alternately make moves. Player A wants to maximize the values of the nodes while B wants to minimize them.

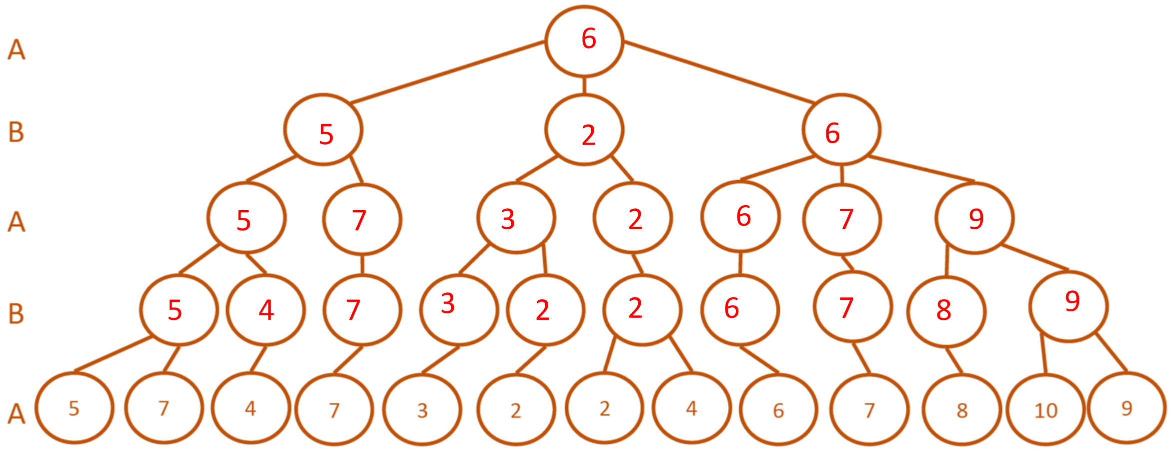
Player A shall make considerations for deciding which move to make from the root situation, and the tree in the figure below shows all situations it is possible to reach with at most four moves from the current situation. A has a heuristic function (that is, a function that for a given situation gives an integer) that he uses to evaluate how god the situation is for him. A uses this function for situations where he terminates the search towards deeper nodes. For each terminal node in the tree below this function is evaluated and the value appears in the nodes.



### 6.a

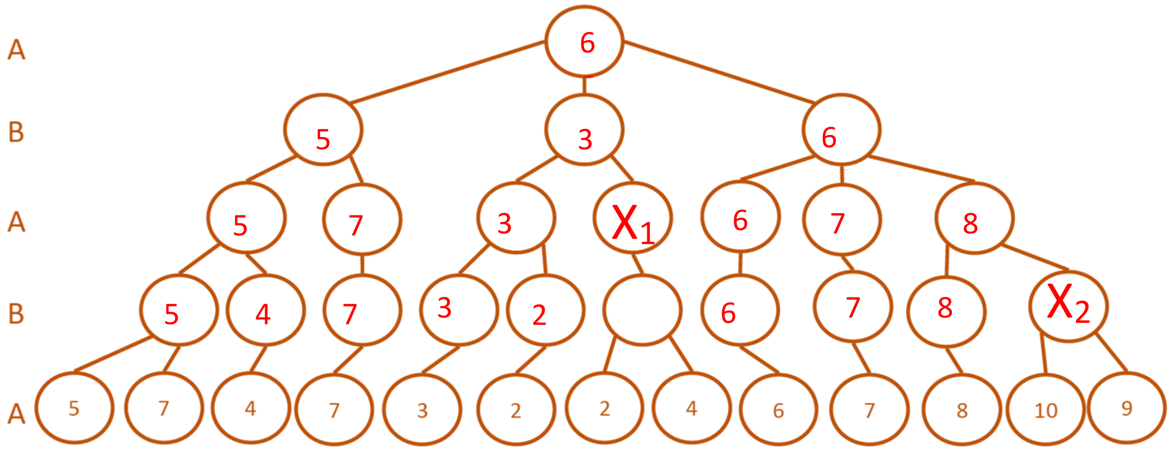
Using the heuristic values in the terminal nodes, indicate how good the situation is for A in each of the other nodes. What is the best value player A can achieve regardless of how well B plays. Don't use any alpha-beta pruning.

A simple application og min and max yields 6 for the root.



### 6.b

We now assume that we are back to the start-situation, no nodes have values. We start the algorithm again, and A will then make a depth-first search in the tree from the root node, down to the depth of the tree above. In each terminal node A computes the value of the heuristic function (and thereby gets the value given in the corresponding terminal node in the figure above). The search is done from left to right in the figure above. Indicate which alpha- and beta-cutoffs you will get during this search. In the drawing you should simply write an 'X' at the root nodes of the sub trees that will not visit because of alpha- or beta-cutoffs. Give a short but explicit explanation for each cutoff (and for this you may suitably give names to some of the nodes in the figure).



The X1 branch i pruned because a smaller value than 3 is irrelevant as A already has a 5 i the left main branch of the tree (at this point the alphsn value in the root will be 5). A larger value than 3 will never be chosen by B.

The X2 branch is pruned because a larger value than 8 is irrelevant as B already has a 6 i the left branch of the sub-tree. A smaller value than 8 will never be chosen by A.

[end]