Algorithms & Datastructures
Laboratory Exercise Sheet 10

Wolfgang Pausch <wolfgang.pausch@uibk.ac.at>
Heiko Studt <heiko.studt@uibk.ac.at>
René Thiemann <rene.thiemann@uibk.ac.at>
Tomas Vitvar <tomas.vitvar@uibk.ac.at>

June 1st, to be discussed on June 8th

Exercise 1) Dynamic Programming You have to cut a cable of length n into pieces, where each piece has a length $\ell \in \mathbb{N}$ with $1 \leq \ell \leq n$. You can sell each piece of length $1 \leq \ell \leq n$ for a price p_ℓ. You have to determine the maximum revenue for a cable of length n for a given table of prices. For example, consider a cable of length 4 with prices $p_1 = 1$, $p_2 = 5$, $p_3 = 6$, $p_4 = 9$. Then one can cut the cable into pieces of length 1 and 3 (resulting in 7) or in two pieces of length 2 (resulting in 10), or in one piece of length 4 (resulting in 9), or ....

1. Implement a recursion algorithm that computes the maximum revenue for a cable of length n. Use the following code skeleton.

```
public class CutCable {

    public static int cutCable(byte[] p, int n) {
        // implement me!
    }

    public static void main(String[] args) {
        byte[] p = {1,5,6,9};
        System.out.println(cutCable(p, p.length));
    }

}
```

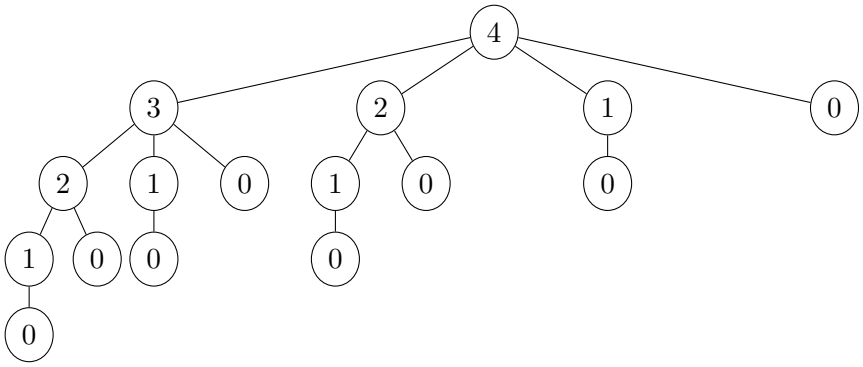
2. Draw the recursion tree for the cable of length 4.
3. Determine the running time of the algorithm. Why is the algorithm inefficient?
4. Use dynamic programming for optimal cable cutting. Implement the improved algorithm.

Exercise 1) Solution

1. A recursion algorithm that computes the maximum revenue for a cable of length n.

```
public class CutCable {

    public static int cutCable( byte[] p, int n) {
        if (n == 0)
            return 0;
        int q = Integer.MIN_VALUE;

        for (int i = 0; i < Math.min(p.length, n); i++)
            q = Math.max(q, p[i] + cutCable(p, n - (i + 1)));
        return q;
    }

    public static void main(String[] args) {
        byte[] p = {1,5,6,9};
        System.out.println(cutCable(p, p.length));
    }

}
```

2. The recursion tree for the cable of length 4 is:



Each node's label is the size n of the subproblem. An edge from a parent node with label s to a child node with label t corresponds to cutting off a piece of size $s - t$ and leaving a remaining subproblem of size t.

3. The running time of the algorithm can be determined as follows: $T(n)$ is the total number of calls made to `cutCable` when called with the second parameter equal to n. It equals the number of nodes in a subtree whose root is labeled with n in the recursion tree. This also include the initial call at its root, thus $T(0) = 1$, $T(n) = 1 + \sum_{j=0}^{n-1}(T(j))$. For example, we have $T(4) = 1 + T(0) + T(1) + T(2) + T(3)$ and $T(3) = 1 + T(0) + T(1) + T(2)$, etc. In general, it holds that $T(n) = 2 * T(n-1)$.

   Using the recursion tree method for solving recurrences, we get $T(n) = c * 2^{n-1}$. For $T(0) = 1$ we get $c = 2$, and the final running time $T(n) = 2^n$. So the running time of `cutCable` is exponential in n.

   The algorithm calls itself recursively often with the same parameter values, thus it solves the same subproblems repeatedly.

4. Dynamic programming algorithm.

```
public class CutCableDP {

    public static int cutCable(byte[] p, int n) {
        int[] r = new int[n+1];
        for (int i = 0; i < n+1; i++)
            r[i] = Integer.MIN_VALUE;
        return cutCableDP(p,n,r);
    }


    public static int cutCableDP(byte[] p, int n, int[] r) {
        if (r[n] > 0)
            return r[n];
        int q = 0;
        if (n > 0) {
            q = Integer.MIN_VALUE;
            for (int i = 0; i < Math.min(p.length, n); i++)
                q = Math.max(q, p[i] + cutCableDP(p, n - (i + 1), r));
        }
        r[n] = q;
        return q;
    }

    public static void main(String[] args) {
        byte[] p = {1,5,6,9};
        System.out.println(cutCable(p, p.length));
    }

}
```

You save the results of each subproblems so that the algorithm does not need to compute the same results several times. The algorithm first checks whether it has previously solved the subproblem. If yes, it returns the saved value, otherwise it computes the value and saves the value for further use.

Exercise 2) Greedy Algorithm You have 8 cities connected with roads as shown in Figure 1 where each road has its length. In winter you want to maintain only a set of roads such that still all cities are connected and that the length of the opened roads is minimal.

Design a greedy algorithm to find the minimal connections among all cities and show each iteration step (hint: the result is a tree, i.e. there must not exist a circle). Give an intuitive idea why your algorithm returns an optimal solution.

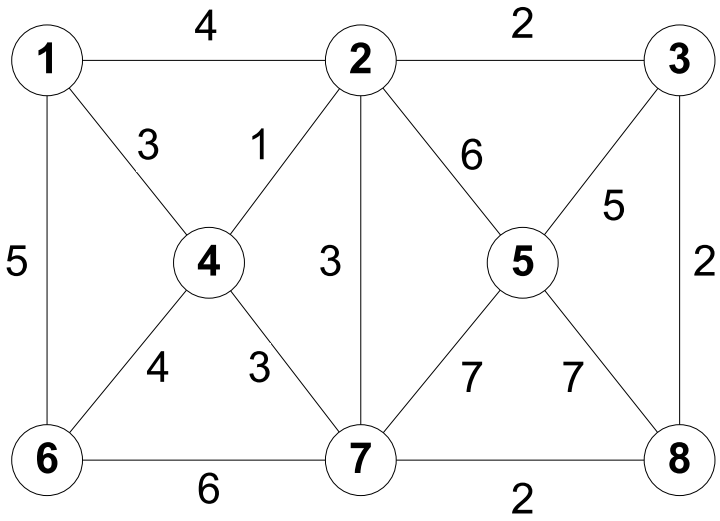


Figure 1: Cities and their connections with lengths

Exercise 2) Solution

1. Greedy algotithm:

   (a) Sort all roads by their length, you have:
   $(4,2) = 1$, $(2,3) = 2$, $(3,8) = 2$, $(8,7) = 2$, $(7,2) = 3$, $(7,4) = 3$, $(4,1) = 3$, $(1,2) = 4$, $(6,4) = 4$, $(5,3) = 5$, $(6,1) = 5$, $(2,5) = 6$, $(6,7) = 6$, $(7,5) = 7$, $(5,8) = 7$

   (b) Create a new graph with the first road from the sorted list.

   (c) Get the next road from the sorted list and check whether it creates a circle in the graph. If not, add it to the graph and contiune with the step (c). If yes, do nothing and continue with the step (c).

   (d) When you go through all the roads from the list, the resulting graph is the minimal connections among roads (so called the minimal spanning tree of the original graph).

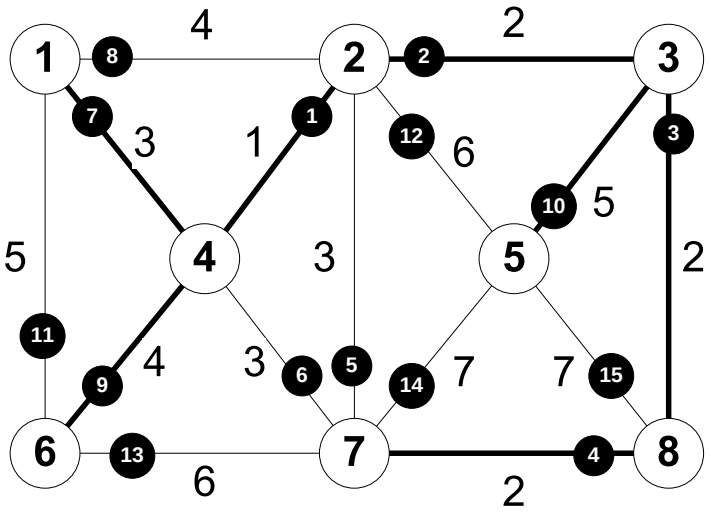


Figure 2: Minimal spanning tree of the graph

2. Algorithm runs as follows (black circles in Figure 2 indicate a step of the algorithm):

   Step 1: add $(4, 2)$
   Step 2: add $(2, 3)$
   Step 3: add $(3, 8)$
   Step 4: add $(8, 7)$
   Step 5: cannot add $(7, 2)$ as it creates a circle
   Step 6: cannot add $(7, 4)$ as it creates a circle
   Step 7: add $(4, 1)$
   Step 8: cannot add $(1, 2)$ as it creates a circle
   Step 9: add $(6, 4)$
   Step 10: add $(5, 3)$
   Step 11: cannot add $(6, 1)$ as it creates a circle
   Step 12: cannot add $(2, 5)$ as it creates a circle
   Step 13: cannot add $(6, 7)$ as it creates a circle
   Step 14: cannot add $(7, 5)$ as it creates a circle
   Step 15: cannot add $(5, 8)$ as it creates a circle

The algorithm always returns an optimal solution as at each step it adds an edge with the minimal costs (note that the list of edges is the sorted list by the costs of edges). The algorithm cannot get to a worse case at a later step than the one it has already found.