Algorithms - Solution 4

1. The solution of the question is purely technical and therefore not interesting. The answer is 18250 multiplications with the optimal input $((A_1A_2)(A_3A_4))$.

2. Assume that we have n activities with start and finish times $(s_i, f_i), i \in \{1 \dots n\}$ respectively. First, sort the activities by their start times. Build an array of length n, where for each start time s_i we store $act[n_i]$- number of activities that can be placed before the start time of n_i **plus 1 (n_i itself)** and $last[n_i]$- index of the activity we place before n_i or null. Fill the array recursively with the recurrence equation $act[n_i] = max\{act[n_j] | f_j \leq s_i\} + 1$. After all the entries are filled, scan an array and find the entry with maximum value- this is the number of activities that can be selected. In order to get the list of activities- use the values of $last$ for maximum value, and then $last[last]$ an so on. It is easy to see that we get an optimal optimal- every optimal solution has the *last* activity that has been selected, and without this activity- we solve the sub-problem, where all the activities *end* before this activity *starts*. The run-time is $O(n^2)$, since, roughly, for each activity we scan an array once.

3. Define W- the capacity of the thief's pack, n- number of items, w_i- weight of item i and p_i- value of item i. Build the matrix $P[n \times W]$ where each entry $[i, j]$ stores the maximal price of items $1 \dots i$ with the pack capacity j and some sign that says whether we took item i or not. When we inspect item i with capacity j we can either take it (if it's price adds to the optimal price for $i-1$ items with the capacity $j - w_i$) or not. Fill the matrix recursively with the recurrence equation

$$P[i,j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ P[i-1,j] & w_j > j \\ max\{p_i + P[i-1, j-w_i], P[i-1,j]\} & \text{else} \end{cases}$$

   and the optimal solution for the whole problem is, of course, at the entry $P[n, W]$ and in order to get the list of items we just scan the matrix "backwards" checking the sign of the item in the solution of sub-problem. Since we examine each entry once and do the constant number of operations on it, the run-time is $O(nW)$.

4. We have a sequence of n numbers. Build an array where for each number i we store $len[i]$- the maximal length of a monotonically increasing subsequence of the first i numbers, **necessarily** including item i and $prev[i]$- the previous number in this subsequence or null. Fill the array recursively with the recurrence equation $len[i] = max\{len[j] | j < i, n_j \leq n_i\} + 1$. After all the entries are filled, scan an array and find the entry with maximum value- this is the length of the monotonically increasing subsequence. In order to get the list of numbers- use the values of $prev$ for maximum value, and then $prev[prev]$ an so on. It is easy to see that we get an optimal solution- every optimal solution has the *last* number that has been selected, and without this number- we solve the sub-problem. The run-time is $O(n^2)$, since, roughly, for each activity we scan an array once.

5. We have a tree (regular, not binary) with n nodes and $w[i]$- weight of each node and assume that the nodes are stored in array, such that we know for each node it's parent and children. For each node we can either take it and it's grandchildren or it's children- this is how we ensure that no two adjacent nodes will be taken. We will fill an array $A[n]$ where at each entry $A[i]$ we will store the maximal weight of sub-tree rooted at i and some sign that says whether we took node i or not. We will scan the tree from the leaves upwards and fill the matrix recursively with the recurrence equation
$A[i] = max\{w[i] + \sum A[j] | j \text{ grandchild of } i, \quad \sum A[k] | k \text{ child of } i\}$, where we add 0 for the null nodes. It is easy to see that we will get an optimal solution at $A[root]$ and, again, we can get the list of nodes scanning the nodes signs. The run-time is $O(n^2)$.
Note: With more careful analysis we can prove that run-time is actually $O(n)$ because every node is has only one father and only one grandfather. Everybody who's grade was deducted is welcome to appeal!

6. (a) The exact definition is at Cormen p.302.
   (b) The recurrence equation is $T(n) = \sum_{k=1}^{n-1} T(k)T(n-k)$ and $T(1) = 1$. Assume that $T(n-1) \geq c * 2^{n-1}$. Now, $T(n) = \sum_{k=1}^{n-1} T(k)T(n-k) = \sum_{k=1}^{n-2} T(k)T(n-1-k) + T(n-1)T(1) \geq \sum_{k=1}^{n-2} T(k)T(n-k) + T(n-1) \geq c * 2^{n-1} + c * 2^{n-1} = c * 2^n$.