CSc 220: Algorithms  
Homework 6 Solutions

Problem 1: Suppose you have one machine and a set of \( n \) jobs \( a_1, a_2, \ldots, a_n \) to process on that machine. Each job \( a_j \) has a processing time \( t_j \), a profit \( p_j \) and a deadline \( d_j \). The machine can process only one job at a time, and job \( a_j \) must run uninterrupted for \( t_j \) consecutive time units. If job \( a_j \) is completed by its deadline \( d_j \), you receive profit \( p_j \), but if it is completed after its deadline, you receive a profit of 0. Give an algorithm to find the schedule that obtains the maximum amount of profit, assuming that all processing times are integers between 1 and \( n \). What is the running time of your algorithm?

Solution: Use dynamic programming. We first order the jobs by increasing deadline so \( a_1, a_2, \ldots, a_n \) are ordered by deadline.

Consider the matrix \( P(i, j) \), defined as the maximum profit we can make using jobs \( a_1, \ldots, a_i \) before deadline \( d_j \). We are looking for \( P(n, n) \). We also add a 0-row with

\[
P(0, j) = 0 \quad \text{for all } j
\]

Note that if \( j > i \) then clearly the optimal solution for jobs \( a_1, \ldots, a_i \) before time \( d_j \) is the same as the optimal solution before time \( d_i \), those jobs add no profit.

\[
P(i, j) = P(i, i) \quad \text{if } j > i
\]

This means that we only need to fill the top triangle of this matrix. Finally if \( j \leq i \) then we have two options: either job \( i \) is part of the optimal solution or it is not. In the latter case, clearly \( P(i, j) = P(i - 1, j) \). In the first case then the optimal solution without job \( i \) should be the optimal solution for jobs \( 1, \ldots, i - 1 \) before time \( j - t_i \), i.e. before deadline \( d_k \) where \( k \) is the maximum deadline such that \( d_k \leq d_j - t_i \). Therefore

\[
P(i, j) = \max[P(i - 1, j), P(i - 1, k) + p_i] \quad \text{if } j \geq d_i \quad \text{where } k = \max[\ell : d_\ell \leq d_j - t_i]
\]

We can fill in the matrix \( P \) starting from the top 0-row and filling in each row in order, since the recursion above only looks at the previous row. To keep track of what actual jobs are scheduled, we keep track of the choices made to compute \( P(n, n) \) at each step.

Running Time: \( P \) is of size \( O(n^2) \). Filling in each square of \( P \) requires \( O(n) \) time at worst (to find the index \( k \)), for a total running time of \( O(n^3) \). Note that the \( O(n \log n) \) sorting cost has been absorbed.

Problem 2: Given a graph \( G = (V, E) \) a subset \( U \subseteq V \) of nodes is called a node cover if each edge in \( E \) is adjacent to at least one node in \( U \). Given a graph, we do not know how to find the minimum node cover in an efficient manner. But if we restrict \( G \) to be a tree, then it is possible. Give a greedy algorithm that finds the minimum node cover for a tree. Analyze its correctness and running time.

Solution: First of all note that for every cover of a tree that contains a leaf, there is a cover that does not contain that leaf and has the same or smaller size. That’s because you can replace the leaf with its parent. Therefore there is a minimum cover that does not contain leaves, but that implies that the leaves’ parents must be in this minimum cover (in order to cover those edges).

This gives rise to the following greedy algorithm. For every leaf in the tree, place its parent in the cover and remove all its adjacent edges (remove all "singleton" disconnected nodes too). You are left with another tree. You can repeat the above process until you have covered all the edges.

The running time is linear since each edge is considered once (you can use DFS on the tree to figure out which ones are the leaves).

Problem 3: Let’s revisit the binary search trees from the previous homework. They are augmented with the following information. At each node \( x \) we also store \( m(x) \): the number of nodes in the subtree rooted at \( x \) (including \( x \)). This time we relax our balance requirement to be that for every node \( x \) in the tree

\[
m(L(x)) \leq \alpha m(R(x)) \quad \text{or} \quad m(R(x)) \leq \alpha m(L(x))
\]

for a constant \( 1/2 \leq \alpha < 1 \). We call these tree \( \alpha \)-balanced.
• Prove that $h = O(\log n)$ where $h$ is the height of a $\alpha$-balanced tree with $n$ nodes.

• Show how to make an arbitrary binary search tree into an $\alpha$-balanced one for $\alpha = 1/2$

• Start from a 1/2-balanced tree. Do insertion and deletions. When the tree is not $\alpha$-balanced anymore, bring it back to 1/2 balanced. Show that inserting and deleting cost $O(\log n)$ time in an amortized sense.

Solution:

• The worst case of unbalance happens when a fraction of $\alpha$ nodes goes always to the same side at each node, say the right side. So at level 1 the right child of the root will have $\alpha n$ nodes in its subtree. At level 2 the rightmost node will have $\alpha^2 n$ in the subtree. And at level $i$ it will be $\alpha^i n$. The height of the tree is the length of the rightmost path which is such that $\alpha^h n = 1$ (since at the point the subtree is just a leaf), which sets $h = \log_{1/\alpha} n = O(\log n)$.

• We saw this in the solution of the previous homework. It takes $O(n \log n)$ time.

• If you start from a fully balanced tree, you must add or delete $\Omega(n)$ elements to violate the balanced condition. Each insertion/deletion before the condition is violated takes $O(\log n)$ actual. If you prepay an additional $O(\log n)$ at each insertion/deletion, you will have $\Omega(n \log n)$ ”money in the bank” to pay for the actual time to rebalance the tree when the condition is violated.