CS624: Analysis of Algorithms

Assignment 4 – solution[10pt]

1. Exercise 15.4-1 in the text (page 396). Of course you could probably solve this problem easily enough just by looking at it. What I want you to do is to explicitly use the algorithm presented in class. Write out your derivation neatly, including the table.

Here is a table with the path marked. Notice that more than one solution exists. If both left and up arrow were equivalent I chose up, but other paths are possible.

<table>
<thead>
<tr>
<th></th>
<th>y_j</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_i</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>↑0</td>
<td>↘1</td>
<td>↘1</td>
<td>↑0</td>
<td>↘1</td>
<td>↑0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>↘1</td>
<td>↑1</td>
<td>↑1</td>
<td>↘2</td>
<td>↑1</td>
<td>↘2</td>
<td>↑2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>↑1</td>
<td>↘2</td>
<td>↘2</td>
<td>↑2</td>
<td>↘3</td>
<td>←3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>↘1</td>
<td>↑2</td>
<td>↑2</td>
<td>↘3</td>
<td>↑3</td>
<td>↘4</td>
<td>←4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>↘1</td>
<td>↑2</td>
<td>↑2</td>
<td>↘3</td>
<td>↑3</td>
<td>↑4</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>↑1</td>
<td>↘2</td>
<td>↘3</td>
<td>↑3</td>
<td>↘4</td>
<td>↑4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>↘1</td>
<td>↑2</td>
<td>↑2</td>
<td>↘3</td>
<td>↑3</td>
<td>↑5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>↘1</td>
<td>↑2</td>
<td>↑2</td>
<td>↘3</td>
<td>↑3</td>
<td>↑6</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>↑1</td>
<td>↘2</td>
<td>↘3</td>
<td>↑4</td>
<td>↘5</td>
<td>↑5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>↘1</td>
<td>↑2</td>
<td>↑2</td>
<td>↘3</td>
<td>↑3</td>
<td>↑6</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>↑1</td>
<td>↘2</td>
<td>↘3</td>
<td>↑4</td>
<td>↘5</td>
<td>↑5</td>
</tr>
</tbody>
</table>

2. Exercise 3.1 in the Lecture 8 handout (on page 8):

The tree is a BST, so the keys are sorted in dictionary order. The fact that every subtree is a consecutive set of keys follows from the properties of BST and it is not necessarily only particular to this problem.

The LCA of i and j, the first and last leaves in the subtree is the root of the subtree (by definition), so it is an ancestor of any other key in the subtree. Now, look at three keys x, y, z – say that y is x’s successor and z’s predecessor. We’ll show that if x and z are in the tree, y must be too. By HW3, y is either ancestor or descendant of both x and z. Either y is x and z’s ancestor, in which case it’s their LCA (if it weren’t LCA, it wouldn’t be both the successor and predecessor of both keys) or the three lie on one path, in which case the uppermost key in the path is the LCA of y and z or x and y, so they all must be in the subtree rooted at the LCA of the entire subtree.

3. The “cut and paste” argument says the following: Since we have independent sub-problems in the sense that the subtree containing nodes i...j where these are consecutive numbers is its own tree, then if this subtree is part of the optimal tree it must be optimal itself. A simple proof by contradiction says that if this weren’t the case, we could have cut it out of the tree and plant a better tree instead of it, not affecting the rest of the tree, and get a better result which contradicts our initial assumption that it was optimal.

4. Exercise 15.5-2 (page 404). Here is the tree:
5. Let T be a rooted tree. T is not necessarily a binary tree. That is, each node may have any finite number of children. We are going to consider a method of visiting each node of the tree. The method works like this:

At Step 1, we visit the root. The root is then marked as “visited”.

At Step n, all nodes that have already been marked “visited” can visit at most one of their children. Of course if all the children of a node have already been visited, there is nothing for that node to do. But other nodes may still do something at that step. (And so in particular, more than one node might be visited in a single step of the process.)

The question is this: what is the minimum number of steps necessary to visit every node in the tree?

(a) Draw a picture of a tree in which more than one node is visited at some step of this process, and show why that happens.

See figure:

(b) Show that this problem contains optimal substructure. Please be careful when you do this. You have to state precisely and clearly what the substructure is and show why it is optimal.

The optimal substructure is that the optimal visitation of a tree contains the optimal visitations of its sub-trees. First you visit a node, then you visit its entire subtree. Therefore, the optimal sub-tree visitation doesn’t depend on the root and the optimal visit time is that of the root + the optimal of the subtrees (by standard cut-paste algorithm).

Since you can visit more than one node at the same time we should visit the children of the root in descending order of their own optimal visitation times.

(c) Use that result to write a recursive algorithm to solve this problem.

i. Visit the node.
ii. Boundary – if the node is a leaf, finish.
iii. For each child, calculate its optimal visitation time. Sort the children by the optimal visit time of their sub-trees.
iv. Visit them in this order.
v. Obviously, do the visit of children simultaneously while continuing the parent.

(d) Turn that algorithm into a dynamic programming algorithm.

Since there is a lot of overlapping calculations of sub-tree visit time and since there is optimal sub-structure (since the optimal visit time for a sub-tree remains the same
whether it is a part of a bigger tree or stand alone), we can think of the visit time of a
tree as follows:

i. If it's a leaf, the visit time is 1.

ii. Otherwise, since we visit the children in descending order of their visit time, the
visiting time is bound by the finish time of the child that finishes last. This would
be the child with the largest visiting time of all the children. So the visiting time of
a tree is $1 + \text{the time we finished visiting the last child}$. If we put the visiting time
of the children in an array $A$ sorted by descending order of their visit time, then the
visiting time of a tree is $\max A[i] + i$

(e) What is the cost of the algorithm? (By that, I mean, “What is the cost of the algo-

rithm that computes the minimum cost?”, not “What is the minimum cost?”. In doing
this problem, please write as little pseudo-code as possible. I would much rather read
something that is clearly explained using ordinary language.

If we have to sort all the children it takes $O(k \log k)$, where $k$ is the number of children.

6. (a) Read Section 10.1. This should be very easy, since I assume you already know about stacks
and queues. The text has an implementation of a queue in terms of an array of fixed size,
in which the operations Enqueue and Dequeue each have cost $O(1)$. (Remember that
this is just a fancy way of saying that the cost is uniformly bounded by some constant.)
You don't have to write anything – just read this and understand it. Let me know if you
find anything confusing.

(b) The only problem with that implementation of queues is that the size of a queue is
bounded by the size of the array. It would be nicer to have a queue which, like a stack,
had no upper bound on its size. (Now of course in practice, a stack does have an upper
bound on its size – that’s why we talk about "stack overflow". But also in practice we
have ways of dealing with this so that except in very unusual circumstances, stacks act
as if their size was not constrained.)

So assuming that we had an implementation for stacks in which their size was not con-
strained, and in which the operations Push and Pop each have cost 1, I first want you to
do Exercise 10.1-6 (page 236). The idea of this exercise is to provide an implementation
of a queue in which the size of the queue is not constrained. In particular, you should
derive the worst-case costs of Enqueue and Dequeue, and it should be clear that the
cost of Dequeue is not $O(1)$. Presumably, that’s the price we pay for having a queue of
unlimited size.

(c) Then finally, do Exercise 17.3-6 (page 463).

We can have two stacks, one "in" and one "out". That is – when push an item into the
stack "in" (it takes $O(1)$). When popping, pop from stack "out" if not empty. If "out"
is empty, pop every item in stack "in" and push it into stack "out". This process takes
$O(n)$ (n is the size of stack "in"). However, it is done once per item. Since we push out
of stack "in" and into stack "out", the items reverse their order so that they are in a
"first in first out" order. The amortized run time is $O(1)$ and it’s easy to see that every
item goes through at most one push (into stack "in") one pop (out of stack "in"), one
push (into stack "out") and one pop (out of stack "out"). Hence, a total of 4 operations
per item.