Analysis of Algorithms - Homework II (Solutions)

K. Subramani
LCSEE,
West Virginia University,
Morgantown, WV
{ksmani@csee.wvu.edu}

1 Problems

1. Professor Stankowski proposes the following algorithm for sorting an array **A** of n numbers:

   (i) If there is only one number, return.

   (ii) If there are two numbers, perform a single comparison to determine the order.

   (iii) If there are more than two numbers, then first sort the top two-thirds of the elements recursively. Follow this by sorting the bottom two-thirds of the elements recursively and then sorting the top two-thirds of the elements again.

   Write a recursive algorithm to implement the above strategy and argue the correctness of Professor Stankowski's algorithm.

   Solution:

```
Function STAN-SORT(A, low, high)
 1: if (low = high) then
 2:    return
 3: end if
 4: if (low + 1) = high) then
 5:    if (A[low] > A[high]) then
 6:       Swap A[low] and A[high].
 7:    end if
 8:    return
 9: end if
10: t_1 = low + (high-low+1)/3
11: t_2 = low + 2 · (high-low+1)/3
12: STAN-SORT(A, low, t_2)
13: STAN-SORT(A, t_1, high)
14: STAN-SORT(A, low, t_2)
```

Algorithm 1.1: Stankowski's Sorting Algorithm

STAN-SORT() should be called as STAN-SORT(**A**, 1, n) from the main program.

1.1 Proof of Correctness

Let $n = high - low + 1$ denote the number of elements in **A**. The algorithm clearly works correctly when $n = 1$ and $n = 2$. Assume that STAN-SORT() works correctly, whenever $n \leq k$, for some $k \geq 2$. Now consider the case in which $n = k + 1$. As per the mechanics of STAN-SORT(), there is a recursive call on the first $(\frac{2}{3})^{rd}$ of the elements of **A**, which are correctly sorted as per the inductive hypothesis. This is followed by a recursive call to the bottom $(\frac{2}{3})^{rd}$ of **A**'s elements, which are once again sorted correctly, as per the inductive hypothesis.

Claim 1.1 *After Line* $(13 :)$ *of Algorithm (1.1) is executed, the largest* $(\frac{1}{3})^{rd}$ *of the elements in* **A** *are in their correct places.*

Proof: Let a_i denote an element which is in the largest $(\frac{1}{3})^{rd}$ of the elements in **A**. At the commencement of Algorithm (1.1), a_i exists in precisely one of the partitions, $p_1 : \mathbf{A}[low \cdot\cdot t_1]$, $p_2 : \mathbf{A}[t_1{+}1 \cdot\cdot t_2]$ and $p_3 : \mathbf{A}[t_2{+}1 \cdot\cdot high]$. If $a_i \in p_3$, then a_i stays in p_3 as per the inductive hypothesis. If $a_i \in p_1$, then after Line $(12 :)$ of Algorithm (1.1) is executed, it is moved to parition p_2, as per the inductive hypothesis, and then to parition p_3, after Line $(13 :)$ is executed. Likewise, if $a_i \in p_2$, then it stays in p_2, after Line $(12 :)$ is executed and moves to parition p_3 after Line $(13 :)$ is executed. As per the inductive hypothesis, when Line $(13 :)$ is executed, the elements in $\mathbf{A}[t_1 \cdot \cdot high]$ are sorted. The claim follows. □

Now when Line $(14 :)$ of Algorithm (1.1) is executed, as per the inductive hypothesis, the smallest $(\frac{2}{3})^{rd}$ of the elements in **A** are also in their correct places. Thus, the entire array is sorted, as per Algorithm (1.1). □

2. What is the comparison complexity of Professor Stankowski's algorithm? Formulate a recurrence relation and solve the same to justify your answer.

   Solution: Let $T(n)$ denote the comparison complexity of STAN-SORT(). From the above description, it follows that,

$$\begin{aligned} T(n) &= 0, \text{ if n} = 1 \\ &= 1, \text{ if n} = 2 \\ &= 3 \cdot T(\frac{2n}{3}), \text{ n} \geq 3 \end{aligned}$$

   Using the Master Theorem, it follows that $T(n) = \Theta(n^{2.71})$. Professor Stankowski's algorithm is worse than INSERTION-SORT()! □

3. Describe how you would implement a queue data structure using two stacks. In particular, describe algorithms for the INSERT() and DELETE() operations, assuming that PUSH() and POP() functions have been implemented.

   Solution:

   We call the two stacks A and B; assume that the elements of the queue are stored in Stack A. Recall that a stack is a LIFO structure, whereas, a queue is a FIFO structure. Implementing INSERT() operation is easy; we simply PUSH() the element onto stack A. The DELETE() operation is implemented as follows: The elements of stack A are popped out using the POP() operation and pushed into stack B, using the PUSH() operation. The top element of B is then popped out, thereby deleting the correct element of the queue structure. □

4. The path length of a binary tree is defined as the sum of the depths of all the nodes in the tree. Write a linear time algorithm to compute the path length of an arbitrary binary tree.

   Solution: We assume that a node v in the tree has the following structure: a left child, which is accessed through $v \rightarrow lchild$, a right child, which is accessed through $v \rightarrow rchild$, a key, which is accessed through $v \rightarrow key$ and a field which stores the number of nodes in the subtree rooted at v. The last field is accessed through $v \rightarrow num$.

   We first compute $v \rightarrow num$ for each node in the tree, by running Algorithm (1.2) on the root r of the input tree. Algorithm (1.2) recursively updates the num field in each node of the tree. It is important to note that each node is touched at most twice; once, when descending down the tree and the second time, when ascending up the tree. Accordingly, Algorithm (1.2) runs in linear time. The key idea is that the number of nodes in the subtree rooted at the

```
Function NODE-NUM(v)
1: if (v == NULL) then
2:    return(0)
3: else
4:    v → num = 1+NODE-NUM(v → lchild)+ NODE-NUM(v → rchild)
5:    return(v → num)
6: end if
```

Algorithm 1.2: Computing the number of nodes in the subtree rooted a node

current node is one more than the number of nodes in the subtrees rooted at its children. A formal proof of correctness can be derived in straightforward fashion, using induction on the height of the tree.

Having run Algorithm (1.2) on the root r of the input tree, we call Algorithm (1.3) on r. The key idea in Algorithm (1.3) is that the path-length at the current node is equal to the path-length of the subtrees at its children plus the number of nodes in those subtrees (Why?). A formal proof of correctness using induction on the height of the tree, is left as an exercise. The IS-LEAF(v) function checks whether the current node is a leaf and returns **true** or **false** accordingly. Note that a node is a leaf if and only if both its children are $NULL$. In this case, we return 0, since the path-length of a leaf node is 0.

```
Function PATH-LENGTH(v)
 1: if (v = NULL) then
 2:    return("error")
 3: end if
 4: {v is not a NULL node}
 5: if (IS-LEAF(v) then
 6:    return(0)
 7: else
 8:    if (v → lchild = NULL) then
 9:       {The right child is definitely not NULL}
10:       return(PATH-LENGTH(v → rchild)+((v → rchild) → num))
11:    else
12:       if (v → rchild = NULL) then
13:          {The left child is definitely not NULL}
14:          return(PATH-LENGTH(v → lchild)+((v → lchild) → num))
15:       else
16:          return(PATH-LENGTH(v → lchild)+((v → lchild) → num)+
                    PATH-LENGTH(v → rchild)+((v → rchild) → num))
17:       end if
18:    end if
19: end if
```

Algorithm 1.3: Computing path length of a binary tree

□

5. In the single machine scheduling problem, you are given a single machine and a collection of tasks $T = \{T_1, T_2, \ldots, T_n\}$ with respective start times $S = \{s_1, s_2, \ldots, s_n\}$ and respective finish times $F = \{f_1, f_2, \ldots, f_n\}$. In other words task T_i starts at time s_i and finishes at time f_i. The machine can execute precisely one task at a time; accordingly, tasks scheduled on the machine must be non-conflicting. The goal is to maximize the number of tasks scheduled on the machine. Design a greedy algorithm for this problem and argue its correctness.

   Solution:

   Algorithm (1.4) represents our greedy strategy.

Function SCHEDULE-TASK(T)

1: Order the tasks in T by their finish times.
2: Without loss of generality, assume that $f_1 \leq f_2 \leq \ldots f_n$.
3: $X_G = T_1$
4: **for** ($i = 2$ **to** n) **do**
5: **if** (T_i does not create a conflict with the tasks in X_G) **then**
6: $X_G = X_G \cup T_i$
7: **end if**
8: **end for**

Algorithm 1.4: Greedy task Scheduling

We claim that X_G is maximal in terms of the number of tasks that can be feasibly scheduled on the machine. To see this, assume that X_G is not maximal and let Y denote the output of an arbitrary algorithm for this problem, which is supposedly optimal.

Lemma 1.1 *X_G is not a proper subset of Y.*

Proof: If $X_G = Y$, then clearly Lemma (1.1) is true. Assume that $X_G \neq Y$ and that $X_G \subset Y$. Let $T_j \in Y$ denote a task which is not in X_G. Now Algorithm (1.4) considered T_j at some point and discarded it, since it created a conflict with the tasks already in X_G. But this means that T_j would create a conflict with the tasks in Y as well! Therefore, we cannot have $X_G \subset Y$. □

Lemma 1.2 *T_1 can always be made part of the optimal solution.*

Proof: Assume that $T_1 \notin Y$, where Y is an optimal solution. Insert T_1 into Y; if it does not conflict with any job in Y, then Y cannot be optimal. We now claim that at most one task in Y can conflict with T_1. To see this, assume the contrary and let two tasks T_r and T_s in Y conflict with T_1. Note that we must have $f_r \geq f_1$ and $f_s \geq f_1$, since T_1 is the task with the smallest finish time. Therefore, we must have $s_r < f_1$ and $s_s < f_1$ (since T_r and T_s conflict with T_1) and thus T_r and T_s conflict with each other! Thus, there can be at most task in Y that conficts with T_1 and this task can be replaced by T_1, without affecting the cardinality of Y. □

Let X_G and Y agree on the first $k-1$ tasks; call this set S_k. Let T_j denote the first task in X_G, such that $T_j \notin Y$. First note that $f_j \leq f_r,\ (\forall r)\ T_r \in Y - S_k$. Add T_j to Y; we claim that T_j creates a conflict with at most one task of Y. Assume the contrary, and let T_j create conflicts with two tasks T_r and T_s of Y. Clearly, neither T_r nor T_s can belong to X_G. Further, we must have $f_r, f_s \geq f_j$. To see this, assume that $f_r < f_j$. Since $T_r \in Y$, it can coexist with the tasks in S_k; but this means that it would have been considered before T_j by the greedy algorithm and added to X_G. A similar argument establishes that $f_s \geq f_j$. Likewise, $s_r < f_j$ and $s_s < f_j$, since T_r and T_s conflict with T_j. But this forces T_r and T_s to conflict with each other, thereby establishing that both cannot exist simultaneously in Y. This means that T_j can displace at most one task in Y, say T_r; this displacement does not affect the number of tasks that can be feasibly scheduled. Let Z denote the set of jobs $Y \cup T_j - T_r$; note that Z is one task closer to X_G than Y is. Working this way, we can transform Y to include all the jobs in X_G, without affecting the number of feasible jobs. At this point, we must have $X_G = Y$ or $X_G \subset Y$, but by Lemma (1.1), $X_G \subset Y$ is not possible. We have thus shown that X_G is maximal, thereby proving that the greedy strategy always outputs the optimal solution.

□