Assignment 3

Algorithms: Correctness and Complexity

2014 Draft Solution

Q0 [10]. We can model a data network as directed graph where switches, routers, computers etc. are nodes and data links are edges. Each edge is associated with a currently available bandwidth. We wish to set up a virtual circuit from node s to node f that has a bandwidth of 1Gbps. I.e., we need a path from s to f whose weakest link (i.e. minimal bandwidth link) is at least 1Gbps. In fact, to avoid creating bottle necks, we would like to find a route whose weakest link is as large as possible. Design an algorithm that takes as input a simple directed graph $G = (V, E)$, two nodes $s, f \in V$ and a bandwidth function $w : E \to \mathbb{R}^+$, and that prints out every simple path from s to f and the weight of its weakest link.

My solution:

```
procedure $paths(G, w, s, f)$
pre: $G$ is a graph, $s$ and $f$ are vertices in $G$
post: result is the set of all simple paths from $s$ to $f$ and have $p$ as a prefix.
    val $P := paths(G, s, f, \{s\}, [s])$
    for $p \leftarrow P$ do
        // The minimum can be computed in the obvious way. I won't go into details
        val $m := \min_{i \in \{0,..p.\text{length}-1\}} w((p(i), p(i+1)))$
        print $p$, $m$
    end for
end $paths$
procedure $paths(G, s, f, V, p)$
pre: $G$ is a graph, $s$ and $f$ are vertices in $G$, $p$ is a path of vertices in $G$, which ends at $s$, $V$ is
  the set of vertices on $p$
post: result is the set of all simple paths that end with $f$ and have $p$ as a prefix.
    if $s = f$ then
        return $\{p\}$
    else
        var $P := \emptyset$
        for each $v$ such that $G$ has an edge from $s$ to $v$
            if $v \notin V$ then $P := P \cup paths(G, v, f, V \cup \{v\}, p\hat{}[v])$ end if
        end for
        return $P$
    end if
end $paths$
```

Q1 [5] A tree over a set A is either

- Leaf()
- Branch(t, a, u) where a is in A and t and u are trees over A.

The fringe of a tree is given by

- fringe(Leaf()) = [ ]
- fringe(Branch(t, a, u)) = fringe(t)ˆ[a]ˆfringe(u)

Write a procedure that takes a finite sequence s over A and prints all trees with s as its fringe.

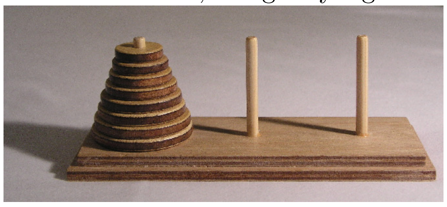
My solution:

```
procedure $allTrees(s)$
    if $s = [\,]$ then return{Leaf()}
    else
        var $P := \emptyset$
        for $i \leftarrow \{0, ..s.\text{length}\}$
            val $T := allTrees(s[0, ..i])$
            val $U := allTrees(s[i+1, ..s.\text{length}])$
            for $t \leftarrow T, u \leftarrow U$ do $P := P \cup \{\text{Branch}(t, s(i), u)\}$ end for
        end for
        return $P$
    end if
end $allTrees$

procedure $toString(x)$
    if $x =$ Leaf() then return "Leaf()"
    else val Branch($t, a, u$) := $x$
        return "Branch(" ˆ$toString(t)$ˆ","ˆ$a$ˆ","ˆ$toString(u)$ˆ")"
    end if
end

procedure $printAllTrees(s)$
    val $P := allTrees(s)$
    for $t \leftarrow T$ do print $toString(t)$ end for
end
```

Q2 **[5]** The towers of Hanoi consists of 3 stacks of disks. There are n disks in total, each a different size from the others; disk 0 is the smallest; disk 1 is larger than disk 0; and so on. A **legal** configuration is one where each disk is on some stack and, on each stack, each disk is only above larger disks. A **legal** move moves the top disk of one stack to the top of another stack that is either empty or is topped by a larger disk. Design an algorithm that starts in *any legal configuration* and moves all disks to one tower, using only legal moves.



Solution:

I'll assume that the spindles are numbered 0, 1,and 2 and that the disks are numbered from 0 up to, but not including, n. I'll assume that the state of the game is represented by an object *game* with accessor *game*.find(i), which returns the number of the spindle disk i is on, and *game*.move(a, b) which moves the top disk of spindle a to spindle b. The precondition of move is that the move is legal, i.e. that a is not empty and that either b is empty or the top-most disk on b is larger than the top-most disk on a.(It follows that a must not equal b.). I'll assume that the game object is only ever in legal configurations, i.e., that it represents only legal configurations.[1]

A solution is moveStack(n, a), where a is the desired destination and moveStack is

```
proc moveStack( $m$, $a$ )
// Move the $m$ smallest disks to spindle $a$.
post: Disks numbered $\{0, ..m\}$ are all on spindle $a$. I.e. $\forall k \in \{0, ..m\} \cdot game.\text{find}(k) = a$
   if $m > 0$ then
       val $b := game.\text{find}(m-1)$
       if $b \neq a$ then
           val $c := 3 - a - b$
           // Move all disks smaller than disk $m-1$ to $c$
           moveStack( $m-1, c$)
           // Now the topmost disk on $b$ must be disk $m-1$
           // and the topmost disk of $a$ (if any) must be larger than $m-1$,
           // since every disk smaller than $m-1$ is on $c$.
           $game$.move($b, a$)
       end if
       {Disk $m-1$ is now on spindle $a$.}
       moveStack( $m-1, a$ ) end if
end moveStack
```

[1]If we want to be very formal, we could specify *game*'s class by using a (possibly abstract) field f that is a total function from $\{0, ..n\}$ to $\{0, ..3\}$. The precondition of find(i) is $i \in \{0, ..n\}$ and the postcondition is *result* $= f(i)$. Suppose f^{-1} is the "inverse image" of f defined by

$$f^{-1}(a) = \{i \mid f(i) = a\}$$

The precondition of move(a, b) is $a, b \in \{0, ..3\}$ and $f^{-1}(a) \neq \emptyset$ and $\min(f^{-1}(a)) = \min(f^{-1}(a) \cup f^{-1}(b))$, while its postcondition is $f(\min(f_0^{-1}(a))) = b$ and $f(i) = f_0(i)$ for all $i \neq \min(f_0^{-1}(a))$.

Applying tail-call removal, we get.

```
proc moveStack( $m$, $a$ )
// Move the $m$ smallest disks to spindle $a$.
post: Disks numbered $\{0, ..m\}$ are all on spindle $a$. I.e. $\forall k \in \{0, ..m\} \cdot game.\text{find}(k) = a$
    var $i := m$
    $\{0 \le i \le m$ and all disks with numbers in $\{i, ..m\}$ are on spindle $a.\}$
    while $i > 0$ do
        $i := i - 1$
        val $b := game.\text{find}(i)$
        if $b \ne a$ then
            val $c := 3 - a - b$
            // Move all disks smaller than disk $i$ to $c$
            moveStack( $i, c$)
            // Now the topmost disk on $b$ must be disk $i$
            // and the topmost disk of $a$ (if any) must be larger,
            // since every disk smaller than $i$ is on $c$.
            $game.\text{move}(b, a)$
        end if
        {Disk $i$ is now on spindle $a$.}
    end while
end moveStack
```

Q3 [10]. A priority queue is a data structure that holds a collection of values (say strings). There is an operation to add a value and an operation to report and remove the smallest value currently in the collection. Show that for any implementation of a priority queue which only accesses its data by copying it and comparing it, the worst-case time complexity of at least one of these operations is $\Omega(\log n)$.

Hint: You might want to use the result we will look at in Tuesday's class that sorting (under the same restrictions) requires $\Omega(n \log n)$ time.

Solution: We proceed by proof by contradiction. I'll assume that the priority queue class has two operations q.add(x) adds a value while q.removeSmallest() returns and removes the smallest value (if there is one) and returns a special value, null, when the queue is empty. I'll also assume that new queues are initially empty.

We'll assume that both these operations have worst case time complexity not in (i.e., better than) $\Omega(\log n)$ where n is the number of items on the queue. Suppose the times for add and removeSmallest are in fact in $\Theta(f(n))$ and $\Theta(g(n))$ respectively, where $f, g \notin \Omega(\log n)$.

Consider the following algorithm to sort a list s of m items.

```
PQueue $q$ := new PQueue() ;
for $x \leftarrow s$ do $q$.add($x$) end for
Loop: var $y$ := $q$.removeSmallest()
    if $y \neq$ null then
        $s := s\hat{}[y]$
        goto Loop
    end if
```

The first loop takes time in

$$\Theta(f(0)) + \Theta(f(1)) + \cdots + \Theta(f(m-1))$$

which will be in $\Theta(mf(m))$. Similarly the second loop takes time $\Theta(mg(m))$. The total time will be $\Theta(m(f(m)+g(m)))$. Since $f, g \notin \Omega(\log n)$, $\Theta(m(f(m)+g(m))) \notin \Omega(m \log m)$. This contradicts the result proved in class that the sort must take time in $\Omega(m \log m)$.

Bonus [5] Suppose s is an array of n things and p is an array of n integers representing a permutation (i.e. the value of p is a one-one onto function from $\{0, ..n\}$ to $\{0, ..n\}$). Each item of p indicates the rank of the corresponding item of s. Write a procedure to sort s according to rank.

```
procedure sortByRank( $p$ : array $\langle\mathbb{N}\rangle$ ,var $s$ : array $\langle T\rangle$)
    precondition $p$ is a permutation of $\{0, ..p.\text{length}\}$ and $p$.length = $a$.length
    changes $s$
    postcondition $\forall i \in \{0, ..p.\text{length}\} \cdot s_0(i) = s(p_0(i))$
```

For example, if p and s are initially

s	bob	doug	alice	clara
p	1	3	0	2

then the final value s should be

s	alice	bob	clara	doug

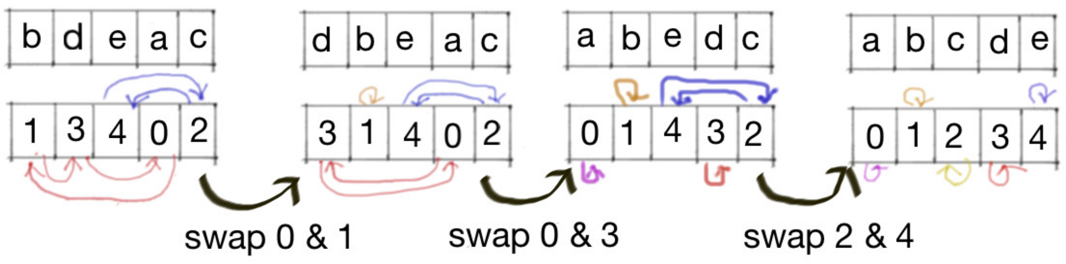
and it doesn't matter what the final value of p is.

Implement this procedure in time $\Theta(n)$, where n is the length of p. Do not use any temporary arrays. Explain why the time is linear.

Solution:

Suppose s.length $= p$.length $= n$. The rank array p is a permutation of $\{0, ..n\}$. If we can turn p into the identity permutation, $[0, ..n]$, while making corresponding changes to array s, then we'll be done. A formal way to look at it is this. Let $f \circ g$ be the function h such that $h(i) = f(g(i))$. And let $p^{-1}(i)$ be the inverse of p, i.e. $p^{-1}(i)$ gives that j such that $p(j) = i$. Note that if p gives ranks for corresponding items in s, then $s \circ p^{-1}$ is simply the function that answers the question, "Given a rank, which value in s has that rank?" Then $s_0 \circ p_0^{-1}$ (where s_0 and p_0 are the initial values of s and p) is the final value required for s. Our postcondition is $s = s_0 \circ p_0^{-1}$.[2] If we can maintain an invariant that $s \circ p^{-1} = s_0 \circ p_0^{-1}$ and change p to be the identity permutation $[0, ..n]$, then we succeed because, once $p = [0, ..n]$, so does $p^{-1}$and so $s \circ p^{-1} = s$ and by the invariant $s = s_0 \circ p_0^{-1}$.

To maintain the invariant, we need to make sure that any changes made to p are matched by corresponding changes to s. In particular, if we swap two items of p and the same two items of s, then the value of $s \circ p^{-1}$ doesn't change. Here is an example. Note that, as corresponding items of p and s are swapped, the composition $s \circ p^{-1}$ does not change.



All that is left is to find a sequence of swaps that makes p into the identity function.

A *fixed-point* of a function p is a value k such that $p(k) = k$. Suppose p is a permutation and i and j are such that $p(i) = j \neq i$; thus i is not a fixed-point. Note that j is also not a fixed-point, since, if $p(j)$ were equal to j, there would be two items of p with the same value. If we swap $p(i)$ with $p(j)$, then after the swap $p(j)$ will equal the old value of $p(i)$ which is j and so the new value of p has j as a fixed-point. A swap of $p(i)$ with $p(j)$, where $p(i) = j \neq i$, increases, by either 1 or 2, the number of fixed-points of p. If we only make this sort of swap, we can't make more than n swaps before there are n fixed-points, i.e., before $p = [0, ..n]$. In the example above, each swap is of this sort; i.e. $p(i)$ and $p(j)$ are swapped where i and j are such that $p(i) = j \neq i$.

[2]The postcondition in the question says.$s_0 = s \circ p_0$. But this is equivalent to $s_0 \circ p^{-1} = s$.

One way to sequence the swaps is to work from left to right. The code could look like the following. In addition to $s \circ p^{-1} = s_0 \circ p_0^{-1}$, the invariant says that the first i natural numbers are fixed-points of p.

```
var $i := 0$
$\{0 \le i \le n \wedge p[0, ..i] = [0, ..i] \wedge s \circ p^{-1} = s_0 \circ p_0^{-1}\}$
// Variant $(n - i) + |\{k \in \{0, ..n\} \mid p(k) = k\}|$
while $i < n$ do
    if $p(i) = i$ then
        $i := i + 1$
    else
        val $j := p(i)$
        $\begin{bmatrix} p(i) \\ p(j) \\ a(i) \\ a(j) \end{bmatrix} := \begin{bmatrix} p(j) \\ p(i) \\ a(j) \\ a(i) \end{bmatrix}$
    end if
end while
```

Each iteration either increments i or makes one swap. Obviously i will be incremented n times. As argued above, each swap increases the number of fixed-points of p, and so there can be no more than n swaps. Thus the total number of iterations is no more than $2n$ (and no less than n). Each iteration takes an amount of time independent of n. The time is thus $\Theta(n)$ in the worst case (and also the best case).
