COMP 252 Assignment 3

Christopher Hundt (110220945)

March 12, 2003

Question 1

This algorithm uses an array, `A`. The array's indexing starts at 1. It is a assumed that the array is implemented in some way such that `parent[A[i]]`, an integer, can be set and retrieved, as can `bit[A[i]]`, a bit. It is also assumed that the array's size can grow as needed. Finally, the algorithm expects to be given n, the number of the elements in the input sequence.

```
T <-- new tree
A <-- new array
i, j <-- 1
root <-- T
tempBit, tempNode, tempParent <-- null
key[root] <-- null

while j < or = n do
  tempNode <-- root
  while tempNode is not null and j < or = n do
    tempParent <-- tempNode
    tempBit <-- x_j
    if tempBit = 0 then
      tempNode <-- left[tempNode]
    else
      tempNode <-- right[tempNode]
    j <-- j + 1
  tempNode <-- new tree/node
  key[tempNode] <-- i
  if tempBit = 0 then
    left[tempParent] <-- tempNode
    bit[A[i]] <-- 0
  else
    right[tempParent] <-- tempNode
    bit[A[i]] <-- 1
  parent[A[i]] <-- key[tempParent]
  i <-- i + 1

output A
```

At the end of the loop, `T` can be discarded, because all its information has been encoded in `A`. As for the running time, every individual line in this algorithm runs in time $\Theta(1)$. There is a nested loop, but in the deepest level of that loop, `j` is increased by one in every iteration, and the algorithm halts when `j` reaches n. Thus, the total running time must be $O(n)$.

Question 2

This algorithm uses an array, `kTimes`. The array's indexing starts at 1. The algorithm expects `k`, the number of elements desired, and `T`, a pointer to the root of the tree being considered. It also assumes the existence of some data type for its tree `auxTree` that has the operations `insert`, `deletemin`, and `minimum`, and which would store pointers but sort by timestamp.

```
auxTree <-- new tree
kTimes <-- new array of size k
i <-- 1

while i < or = k
  if T is null then T <-- deletemin(auxTree)
  if auxTree is empty then
    kTimes[i] <-- T
    i <-- i + 1
    if T has two children then
      if t[left[T]] < t[right[T]] then
        insert(right[T], auxTree)
        T <-- left[T]
      else
        insert(left[T], auxTree)
        T <-- right[T]
    if T has one child then
      T <-- left[T] or right[T] (whichever is not null)
    if T has no children then T <-- null
  else (auxTree is not empty)
    if t[T] < or = t[minimum(auxTree)] then
      kTimes[i] <-- T
      i <-- i + 1
      if t has two children then
        if t[left[T]] < t[right[T]] then
          insert(right[T], auxTree)
          T <-- left[T]
        else
          insert(left[T], auxTree)
          T <-- right[T]
      if T has one child then
        T <-- left[T] or right[T] (whichever is not null)
      if T has no children then T <-- null
    else (t[T] > t[minimum(auxTree)])
      insert(T, auxTree)
      T <-- deletemin(auxTree)

output kTimes
```

The first three lines of this algorithm clearly have running time $\Theta(1)$. Before we begin the analysis of the rest of the algorithm, we can assume that we are using some implementation for `auxTree` that allow the required operations to be performed in time $O(\log m)$, where m is the number of nodes in `auxTree`. For instance, red-black trees would meet this requirement. Let us now consider the running time of each line in the `while` loop:

1. If the current node is empty then make it the node with minimum timestamp in `auxTree`. Time is $O(\log m)$.

2. If `auxTree` is empty (Time $\Theta(1)$ for check):
   (a) Set the next element of `kTimes` and increment the counter. Time is $\Theta(1)$.
   (b) Insert the child with the larger timestamp in `auxTree`, and set the current node to be the smaller one. Time is $O(\log m)$, unless there is one or fewer children, in which case time is $\Theta(1)$.
3. If `auxTree` was not empty:
   (a) If the timestamp of the current node is less than (or equal to) the minimum timestamp in `auxTree` (Time $O(\log m)$ for check):
      i. Set the next element of `kTimes` and increment the counter. Time is $\Theta(1)$.
      ii. Insert the child with the larger timestamp in `auxTree`, and set the current node to be the smaller one. Time is $O(\log m)$, unless there is one or fewer children, in which case time is $\Theta(1)$.
   (b) If the timestamp of the current node was greater than the minimum timestamp in `auxTree`:
      i. Insert the current node in `auxTree`. Time is $O(\log m)$.
      ii. Set the current node to the minimum of `auxTree` (deleting it from `auxTree` in the process). Time is $O(\log m)$.

There are no nested loops, so none of the instructions will be performed more than once. Even if every instruction were performed once (which is not the case, because of the `if` statements), the time would be $O(\log m)$. Thus, the running time is clearly $O(\log m)$ for each iteration of the loop.

We also note that there is only one case in which the counter `i` for the loop is not incremented. This is the case when `auxTree` is not empty and the node being considered has a timestamp greater than that of the minimum from `auxTree`. However, after this occurs, that minimum is made the current node, so this new current node will certainly be added to `kTimes` in the next iteration, so the counter must be incremented the next time through. Since the counter starts at one, and the loop ends after it reaches k, this means that the loop executes no more than $2k$ times. This means that the total running time for the algorithm is no more than $\Theta(1) + 2kO(\log m) \in O(k \log m)$.

This leaves one remaining question: what is m? Clearly, $m = 0$ at the beginning of the algorithm, because `auxTree` has just been created. Looking back at the algorithm, we see three cases in which m is increased (that is, an element is inserted into `auxTree`):

1. `auxTree` is empty, the current node has been added to `kTimes`, and its larger child is added to `auxTree`, adding 1 to m. `i` was just increased by 1.
2. `auxTree` is not empty, but the current node has a timestamp smaller than the minimum in `auxTree`, and the current node's larger child is added to `auxTree`, adding 1 to m. `i` was just increased by 1.
3. `auxTree` is not empty, and its minimum timestamp is smaller than that of the current node. The current node is added to `auxTree`, but then the minimum is deleted, so the net change of m is 0.

This shows that whenever m increases, `i` increases by the same amount. Since the loop halts when `i` reaches k, it is clear that $m \leq k$ at all times in the algorithm. Thus, the total running time is $O(k \log k)$.

Question 3

Before we describe how to merge $10n$ arrays into n arrays, we will consider the problem of simply merging 10 sorted arrays into 1 sorted array. We will call will the arrays $x_1, \ldots, x_{10}$, and their respective sizes $s_1, \ldots, s_{10}$, and we assume the existence of the operation `merge(A1, A2)`, which merges two arrays in time equal to the sum of the sizes of the arrays. The algorithm follows, with notes on the side:

```
PART I:
x12 <-- merge(x1, x2)                          (s12 = s1 + s2)
x34 <-- merge(x3, x4)                          (s34 = s3 + s4)
x56 <-- merge(x5, x6)                          (s56 = s5 + s6)
x78 <-- merge(x7, x8)                          (s78 = s7 + s8)
x90 <-- merge(x9, x10)                         (s90 = s9 + s10)

PART II:
x1234 <-- merge(x12, x34)                      (s1234 = s12 + s34 = s1 + s2 + s3 + s4)
x5678 <-- merge(x56, x78)                      (s5678 = s56 + s78 = s5 + s6 + s7 + s8)

PART III:
x567890 <-- merge(x5678, x90)                  (s567890 = s5678 + s90
                                                        = s5 + s6 + s7 + s8 + s9 + s10)
PART IV:
x_tot <-- merge(x1234, x567890)                (s_tot = s1234 + s567890
                                                      = s1 + s2 + s3 + s4 + s5 + s6 + s7
output x_tot                                             + s8 + s9 + s10)
```

We can now analyze the running time of this algorithm. We first note that the running time for any merge operation is equal to the size of the resulting array, or the number of elements included in it.

Now let us call S the sum of all the sizes; that is, $S = s_1 + s_2 + \ldots + s_{10}$. We see from the sizes of the resulting arrays that the time for the first part of the algorithm is S, because every element in the original group of arrays is included exactly once in the resulting arrays. The second part of the algorithm has time less than S because no elements are included twice in the results and some elements (those in x_9 and x_{10}) are not included at all. The third part of the algorithm makes an array whose size is less than S, so its time is less than S. Finally, the fourth part again has time S. So the total running time R is thus $R < S + S + S + S = 4S$.

To extend this algorithm to more than 10 arrays, we simply need to repeat it. So the algorithm for merging $10n$ arrays would be to simply repeat the above algorithm n times, each time merging 10 arrays into 1, and then going onto the next 10. For every 10 arrays, the running time would be $R < 4S$, where S is the total number of elements in those 10 arrays. Since the resulting array is then left alone and the next 10 arrays are considered, every element is in 1 group of 10, and no element is in 2 or more groups of 10. Thus, the total running time T would be

$$\begin{aligned} T &= \sum_{\text{groups of } 10} R \\ &< 4N, \end{aligned}$$

where N is the total number of elements in all lists together.

Question 4

We first consider the problem of merging two arrays of elements. The length of each array cannot be greater than k, so the running time for a normal merge is no more than $k+k=2k$, regardless of the size of n (using the RAM model, of course, because each k-value holds a number indicating the number of elements equal to that value, which may need to be added in the merge process).

We also know that k cannot exceed n. However, we will, for convenience, say that k can exceed n, but the "effective" value of k is bounded by n. That is, if $n < k$, then it doesn't really matter what k is; the problem is the same as that of merging two lists of unique numbers. For the base case of just merging two single-element arrays, we see that $n = 2$, and therefore the "effective" $k \leq 2$, so the running time $T(2) \leq 2 + 2 = 4$. So we say that once n reaches 2, the time is constant.

We can now consider the time to sort as a recurrence for some general values of n and k. Since we must first have two sorted lists of size $n/2$, we say

$$T(n,k) \leq 2T\left(\frac{n}{2},k\right) + 2k. \tag{1}$$

However, this is eventually misleading. As we noted before, once n becomes sufficiently small, then the effective k must shrink as well. Since the number of comparisons will not exceed n, we can say that we will only use the above recurrence if $n > k$. Otherwise, a new recurrence comes into play:

$$T(n,k) \leq 2T\left(\frac{n}{2},k\right) + n; \qquad n \leq k \tag{2}$$

Let us now look at the first few terms of recurrence 1, switching the order of the summands:

$$\begin{aligned} T(n,k) &\leq 2k + 2(2k + 2(2k + 2(2k + \dots \\ &\leq 2k + 4k + 8k + 16k + \dots \\ &\leq 2^1k + 2^2k + 2^3k + 2^4k + \dots + 2^ik + \dots \end{aligned}$$

Let us call n_0 the original total number of elements. In the first level of the recursion, which corresponds to the first term $(2k)$, $n = n_0$. In the next level, corresponding to the term 2^2k, $n = n_0/2$. In general, in the ith level of recursion, with corresponding term 2^ik, $n = n_0/2^{i-1}$. As noted above, the time at which $n/2 = k$, or $n = 2k$, is the last time we will use recurrence 1 (because for the next term, $n = k$). We can now solve for i at this time:

$$\begin{aligned} 2k &= \frac{n_0}{2^{i-1}} \\ 2^ik &= n_0 \\ 2^i &= \frac{n_0}{k} \\ i &= \log_2 \frac{n_0}{k} \end{aligned}$$

After that point, recurrence 2 would come into effect. Expanding the first few terms of recurrence 2, we see it looks like the following:

$$\begin{aligned} T(n,k) &\leq n + 2T\left(\frac{n}{2},k\right) \\ &\leq n + 2\left(\frac{n}{2} + 2\left(\frac{n}{4} + 2\left(\frac{n}{8} + 2\left(\frac{n}{16} + \dots\right.\right.\right.\right. \\ &\leq n + n + n + n + n + \dots \end{aligned}$$

Again, at each level of the recursion, the size of the n being considered is halved, so that at any level, $n = n_0/2^{i-1}$ (even though it is multiplied by 2^{i-1}). This time, however, the recursion stops when the size of the n being considered reaches 2, at which point we say the time is constant. So we solve:

$$\begin{aligned} \frac{n_0}{2^{i-1}} &= 2 \\ n_0 &= 2^i \\ i &= \log_2 n_0 \end{aligned}$$

We can now see that the total process would be recurrence 1 repeated $\log_2(n_0/k)$ times (at which point $n < k$), followed by recurrence 2 repeated until the total number of repetitions for both times is equal to $\log_2 n_0$ (at which point $n = 2$), followed by a constant number of steps. We note that this means the number of repetions of recurrence 2 would be:

$$\log_2 n_0 - \log_2 \frac{n_0}{k} = \log_2 n_0 - (\log_2 n_0 - \log_2 k) = \log_2 k$$

We can also see that, the first time recurrence 2 is used, it will be at depth i in recurrence 1, where $i = \log_2(n_0/k)$. This means that $T(n/2, k)$ (where $n/2 = k$) will be multiplied by $2^i = n_0/k$. Since $n/2 = k$, this shows us that all the n's in the $n + n + \ldots$ form of recurrence 2 will actually be k's, but they will be multiplied by n_0/k.

This allows us to express a bound for the total running time in the following way:

$$\begin{aligned}
T(n_0, k) &\leq 2k + 2^2 k + \ldots + 2^{\log_2 \frac{n_0}{k}} k + \frac{n_0}{k} \left(\underbrace{k + k + \ldots + k}_{\log_2 k} + 4 \right) \\
&\leq \sum_{i=1}^{\log_2 \frac{n_0}{k}} 2^i k + \frac{n_0}{k} (\log_2 k \cdot k + 4) \\
&\leq k \sum_{i=1}^{\log_2 \frac{n_0}{k}} 2^i + n_0 \log_2 k + 4\frac{n_0}{k} \\
&\leq k \cdot 2^{\log_2 \frac{n_0}{k} + 1} + n_0 \log_2 k + 4\frac{n_0}{k} \\
&\leq 6n_0 + n_0 \log_2 k
\end{aligned}$$

Since $6n_0 \in O(n_0 \log k)$, and n_0 is our original n, this shows that the total running time is $O(n \log k)$.