COMP251 Assignment 2

Adam Hooper
260055737

February 15, 2005

1. Exercise 8.3-4

We are asked to show how to sort n integers in the range $[0\,..\,n^2-1]$ in $O(n)$ time.

This is a the perfect place to use RADIX-SORT. Integers under n^2 take at most $\lg n^2 = 2\lg n$ bits of storage. Using a radix of $r = \lg n$, we can guarantee linear sort time by Lemma 8.4:

> Given n b-bit numbers and any positive integer $r \leq b$, RADIX-SORT correctly sorts these numbers in $\Theta\left((b/r)(n+2^r)\right)$ time.

In this particular problem, $r = \lg n$, and $b = 2\lg n$. The running time thus becomes $\Theta\left(\frac{2\lg n}{\lg n}\left(n + 2^{\lg n}\right)\right) = \Theta\left(2\left(n+n\right)\right) = \Theta(n)$.

2. Problem 8.4

a. $\Theta(n^2)$ comparisons

To pair up the jugs in $\Theta(n^2)$ time, we simply have to compare all red jugs with all blue jugs. Assuming red jugs are in the array $R[1\,..\,n]$ and blue jugs are in the array $B[1\,..\,n]$, the following algorithm will work. (The algorithm will reorder B to contain jugs in the same order as in A.)

Note that this algorithm is almost identical to INSERTION-SORT.

```
PAIR-JUGS-SIMPLE($R, B$)
1  for $i \leftarrow 1$ to $length[R]$
2      do for $j \leftarrow i$ to $length[B]$
3             do if COMPARE($R[i], B[j]$) = 0
4                    do exchange $B[i] \leftrightarrow B[j]$
```

b. Prove lower bound of $\Omega(n \lg n)$ comparisons

This proof is identical to the proof we performed in class that any sorting algorithm cannot have a running time better than $\Omega(n \lg n)$:

- Any sorting algorithm must be able to permute the input values in all possible ways.
- Two distinct input sequences ("sequence" meaning ordering of B from smallest to largest) transformed by the same permutation will result in two distinct output sequences. Therefore each distinct input sequence must have its own associated permutation.
- Any sorting algorithm must be able to sort any of the $n!$ possible input sequences. This will require at least $n!$ leaves on its decision tree.
- As proven in class, any binary tree of height h has at most 2^h leaves. So if l is the number of leaves on the decision tree, $\lg l \leq h$.
- $n! > \left(\frac{n}{e}\right)^n$ (Stirling's Formula). So $h \geq \lg n! > \lg \left(\frac{n}{e}\right)^n = n \lg n - n \lg e = \Omega(n \lg n)$.

c. Randomized algorithm with expected $O(n \lg n)$ comparisons

Just as the earlier algorithm emulated INSERTION-SORT, the following algorithm emulates RANDOMIZED-QUICKSORT. As a side-effect, it reorders R at the same time as reordering B.

```
PAIR-JUGS-PARTITION($R, B, p, r$)
1  $i \leftarrow p - 1$
2  for $j \leftarrow p$ to $r - 1$
3      do if COMPARE($B[j], R[r]$) $\leq 0$
4            then $i \leftarrow i + 1$
5                 exchange $B[i] \leftrightarrow B[j]$
6  exchange $R[i+1] \leftrightarrow R[r]$
7  return $i + 1$
```

```
PAIR-JUGS-PARTITION-RANDOM($R, B, p, r$)
1  $i \leftarrow$ RANDOM($p, r$)
2  exchange $R[r] \leftrightarrow R[i]$
3  for $i \leftarrow p$ to $r - 1$
4      do if COMPARE($B[i], R[r]$) $= 0$
5              do exchange $B[i] \leftrightarrow B[r]$       ▷ Make $B$'s partition the same as $R$'s
6  PAIR-JUGS-PARTITION($R, B, p, r$)
7  return PAIR-JUGS-PARTITION($B, R, p, r$)     ▷ Will return same value as previous line
```

```
PAIR-JUGS-QUICK($R, B, p, r$)
1  if $p < r$
2      then $q \leftarrow$ PAIR-JUGS-PARTITION-RANDOM($R, B, p, r$)
3           PAIR-JUGS-QUICK($R, B, p, q - 1$)
4           PAIR-JUGS-QUICK($R, B, q + 1, r$)
```

Sorting two arrays at once is obviously more complicated than sorting a single array. However, using QUICKSORT's logic we are assured that the algorithm works, because PAIR-JUGS-PARTITION works. Both arrays have the same number of jugs smaller than the partition element; running PAIR-JUGS-PARTITION with the arrays swapped will position the partition element at the same point in both arrays. And since the partition point is in the correct place, we know that all elements before $R[i]$ will be less than $B[i]$ and all elements before $B[i]$ will be less than $R[i]$. Enough symmetry is thus preserved.

Proving the algorithm's expected running time of $O(n \lg n)$ is trivial since we already know that RANDOMIZED-QUICKSORT runs in $O(n \lg n)$ time. All we need to prove is that PAIR-JUGS-PARTITION-RANDOM runs in $O(n)$ time. Since the first part (placing the partition point) iterates over each element once for $O(n)$ time, and since PAIR-JUGS-PARTITION (a direct translation of QUICKSORT's PARTITION) runs in $O(n)$ time, it is obvious that the entire PAIR-JUGS-PARTITION-RANDOM function rung in $O(n)$ time. And since all other functions in the array are directly copied from QUICKSORT, we can rest assured the expected running time is $O(n \lg n)$.

As with QUICKSORT, PAIR-JUGS-PARTITION-RANDOM's worst-case number of comparisons is $O(n^2)$; this is very unlikely to occur in practice, though.

3. Problem 8-6

a. $2n$ numbers $\leftrightarrow \binom{2n}{n}$ possible divisions

How many n-element-long sorted lists can be generated from a $2n$-element-long list? This is pretty much the definition of nCr: "from $2n$ elements, choose n of them." Since the desired list is sorted, each set of n elements can only have one order. After choosing n elements for the first list, the other n elements implicitly make up the second list. Again, since the desired second list is sorted, there can only be one ordering of its elements. So a list of $2n$ numbers can be divided into $\binom{2n}{n}$ sorted lists of n numbers.

b. Merge uses at least $2n - o(n)$ comparisons

The decision tree for MERGE must be able to merge any two sorted lists. As shown above, there are $\binom{2n}{n}$ possible methods to pick two n-element lists from a $2n$-element list; to reverse this procedure, MERGE's decision tree must have at least $l = \binom{2n}{n}$ leaves. Using the formula $h \geq \lg l$, we can determine the minimum height of the decision tree, which is also the minimum number of comparisons.

Stirling's formula, $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$, is used in these calculations.

$$\begin{aligned}
h &\geq \lg l \\
&\geq \lg\binom{2n}{n} \\
&\geq \lg\left(\frac{(2n)!}{(n!)^2}\right) \\
&\geq \lg\left(\frac{\sqrt{4\pi n}\left(\frac{2n}{e}\right)^{2n}\left(1+\Theta\left(\frac{1}{2n}\right)\right)}{\left(2\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)\right)^2}\right) \\
&\geq \lg\left(\frac{1}{2\sqrt{4\pi n}}2^{2n}\left(\frac{1+\Theta\left(\frac{1}{2n}\right)}{\left(1+\Theta\left(\frac{1}{n}\right)\right)^2}\right)\right) \\
&\geq 2n - \lg\left(4\sqrt{\pi n}\right) + \lg\left(\frac{1+\Theta\left(\frac{1}{2n}\right)}{1+2\Theta\left(\frac{1}{n}\right)+\Theta\left(\frac{1}{n}\right)^2}\right) \\
&\geq 2n - \lg\left(4\sqrt{\pi n}\right) + 0 \text{ as } n \to \infty
\end{aligned}$$

The large last lg was removed to abbreviate the following step, in which it is easy to see it would go to 0 anyway.

Next, we need to prove that $\lg(4\sqrt{\pi n}) = o(n)$:

$$\begin{aligned}
\lim_{n\to\infty}\frac{\lg(4\sqrt{\pi n})}{n} &= \lim_{n\to\infty}\frac{2+\frac{1}{2}\lg\pi+\frac{1}{2}\lg n}{n} \\
&= 0
\end{aligned}$$

Thus, we have proven that $h = 2n - o(n)$, and so at most $2n - o(n)$ comparisons must be made in the MERGE algorithm.

c. Requirements for two elements to be compared

If two elements are in consecutive order from opposite lists, they must be compared. This can be explained by cases. In the following examples, A is the first sorted half-list and B is the second; a is MERGE's index into A and b is MERGE's index into B. C is the target list.

- If $A[a]$ comes before $B[b]$ but not directly before it, then elements $A[a..i-1]$ will be moved into list C, for some value of i. This will leave $A[i]$ in list A; $A[i]$ comes directly before $B[b]$ so the consecutive elements $A[i]$ and $B[b]$ will be compared.
- If $A[a]$ comes directly before $B[b]$, the two consecutive elements will be compared.

- If $A[a]$ comes after $B[b]$, the two above arguments still hold, but with A and a reversed with B and b.

Since a and b end up taking all possible values from 1 to n within MERGE, the above situations will arise for all possible indices into both arrays. Thus, any two consecutive elements from opposite lists will be compared.

d. Lower bound of $2n - 1$

We have just proven that any two consecutive terms from opposite lists will be sorted. In other words, *all* consecutive terms from opposite lists will be sorted. There are $2n$ terms in total, and thus $2n - 1$ sets of consecutive terms. Thus $2n - 1$ comparisons must be made.

4. Exercise 9.3-7

We are asked for an $O(n)$ algorithm to find the k numbers closest to the median of a set S of n distinct integers.

```
SELECT-MIDDLE-NUMBERS($S, k$)
1  $i \leftarrow \lceil \frac{length[S]-k+1}{2} \rceil$
2  SELECT($S, i$)
3  $T \leftarrow S[i \,..\, length[S]]$
4  SELECT($T, k$)
5  return $T[1 \,..\, k]$
```

Each of the lines in SELECT-MIDDLE-NUMBERS run in at most $O(n)$ time, so the entire algorithm runs in $O(n)$ time. The first call to SELECT will partition around the lowest desired number; we don't yet care what that number is. We then place the entire array into T, thus discarding all elements smaller than any of the elements we want. We repeat the procedure to discard all elements greater than any of the elements we want. (The temporary array T is necessary: the second call to SELECT could reorder elements positioned in the first call since the groups of 5 elements are different in both calls to SELECT.)

5. Problem 9-2

a. Regular median is a weighted median

Suppose we look for the weighted median set of $2n$ distinct elements $x_1, x_2, \ldots, x_{2n}$, each with weight $w_i = \frac{1}{2n}$, and we are looking for the (lower) median, x_n. Then the two given equations are satisfied:

$$\sum_{x_i < x_n} w_i = \frac{n-1}{2n} < \frac{1}{2}$$

$$\sum_{x_i > x_n} w_i = \frac{n}{2n} \leq \frac{1}{2}$$

Similarly, if there are $2n - 1$ distinct elements (i.e., an odd number of elements), each has weight $w_i = \frac{1}{2n-1}$ and we are looking for x_n. The equations are still satisfied:

$$\sum_{x_i < x_n} w_i = \frac{n-1}{2n-1} < \frac{1}{2}$$

$$\sum_{x_i > x_n} w_i = \frac{n-1}{2n-1} \leq \frac{1}{2}$$

b. Compute weighted median using sorting

This algorithm follows straight from the definition of weighted median. It sorts the list and applies a summation.

```
WEIGHTED-MEDIAN-SORTED($A$)
1  HEAPSORT($A$)
2  $w \leftarrow 0$
3  $i \leftarrow 0$
4  while $w < \frac{1}{2}$
5        do $i \leftarrow i + 1$
6           $w \leftarrow w + weight[A[i]]$
7  return $A[i]$
```

The sorting runs in $O(n \lg n)$ time. The loop iterates over at most n elements and so runs in $O(n)$ time. Therefore the total running time is $O(n \lg n)$.

c. Use a SELECT-like algorithm

We know that SELECT works by running PARTITION multiple times. In essence, if SELECT calls PARTITION with the weighted median as input, we will be able to verify the weighted median's validity in $O(n)$ time using the same type of loop as in WEIGHTED-MEDIAN-SORTED.

The algorithm, WEIGHTED-MEDIAN, will be a variation of SELECT which does not require an order argument (since we always want to find the median). After each call to

PARTITION, a loop like the one in part b will determine whether the pivot in PARTITION was actually the weighted median. If it is the median, WEIGHTED-MEDIAN will return it. If the sum of all numbers up to and including the pivot is less than $\frac{1}{2}$, WEIGHTED-MEDIAN will recurse to the right of the pivot. Otherwise, WEIGHTED-MEDIAN will recurse to the left of the pivot.

The running time of WEIGHTED-MEDIAN is $O(n)$. This follows from the proof that SELECT is $O(n)$. We are only adding an $O(n)$ step in the algorithm, in the same step as that in which PARTITION is called. Since PARTITION is $O(n)$ and the test for the pivot being the weighted median is also $O(n)$, the entire proof of SELECT's running time being $O(n)$ remains untouched. So WEIGHTED-MEDIAN's running time is $O(n)$.

d. Post-office problem

To prove the weighted median's appropriateness for the 1-dimensional post-office problem, we need only do the math.

$$\begin{aligned}
\sum_{i=1}^{n} w_i d(p, p_i) &= \sum_{i=1}^{n} w_i \left| p - p_i \right| \\
&= \sum_{i=1}^{n} \left| p w_i - p_i w_i \right| \\
&= \sum_{i=1}^{k} (p w_i - p_i w_i) + \sum_{i=k+1}^{n} (p_i w_i - p w_i) \\
&= \sum_{i=1}^{k} p w_i - \sum_{i=k+1}^{n} p w_i + \sum_{i=k+1}^{n} p_i w_i - \sum_{i=1}^{k} p_i w_i \\
&= p \left(\sum_{i=1}^{k} w_i - \sum_{i=k+1}^{n} w_i \right) + p_i \left(\sum_{i=k+1}^{n} p_i w_i - \sum_{i=1}^{k} p_i w_i \right)
\end{aligned}$$

Both sets of sums depend on p, since k depends on p. But there is an interdependence: as the fourth sum ranges over more terms (i.e., k gets larger), the first sum also gets larger, by a larger amount (since $p \geq p_i$ for all terms). Same goes for the second and third sums. So there exists an optimal value for which changing k makes the entire sum larger.

The optimum situation will occur when both the first sums are equal (to $\frac{1}{2}$) and cancel each other out. That is, by definition, the weighted median.

e. 2-dimensional post-office

This question relies upon the same logic as the last. The easiest way to solve it is to again write out the equation:

$$\begin{aligned}\sum_{i=1}^{n} w_i d(p, p_i) &= \sum_{i=1}^{n} w_i \left(|x - x_i| + |y - y_i|\right) \\ &= \sum_{i=1}^{n} w_i \left|x - x_i\right| + \sum_{i=1}^{n} w_i \left|y - y_i\right|\end{aligned}$$

After this, the exact logic applies as in part d. So the best solution is the set of points (x, y) where x is the weighted median in the first dimension and y is the weighted median in the second.