COMP251 Assignment 1

Adam Hooper
260055737

February 1, 2005

1. Exercise 2.3-7

We are asked to describe a $\Theta(n \lg n)$ algorithm which takes a set S of n integers and another integer x and determines whether or not there exist two elements in S whose sum is exactly x.

```
EXIST-TWO-ELEMS-WITH-SUM-X($S, x$)
1  MERGE-SORT($S$)
2  for $i \leftarrow 1$ to LENGTH($S$) $- 1$
3       do if BINARY-SEARCH($S, x - S[i]$)
4             do return TRUE
5  return FALSE
```

The running time analysis is simple: there are n elements in S; for each element, an $O(\lg n)$ binary search is performed. Therefore, iterating over all elements will take $O(n \lg n)$ time. The initial sorting is also done in $O(n \lg n)$ time. The sum is $O(n \lg n)$ and so this algorithm solves the problem.

2. Problem 4-2

We are given an array $A[1..n]$, which we can only access a single bit at a time.

This implies we should look at the array differently: instead of being a single-dimensional array of integers, look at it as a two-dimensional array of bits. The first index (row) specifies the integer, and the second (column) specifies the bit of that integer. The array size is $n \times \lceil \lg n \rceil$. Retrieving an element from A takes $O(1)$ time.

Obviously, since $n \times \lceil \lg n \rceil \geq n$, we cannot look at every bit of the two-dimensional array in our solution.

To solve the problem, create a second array, $B[1..n]$: each element of B is an index into an element of A. Assume that indexing B also takes $O(1)$ time. Initially, populate B with $1, 2, 3, ..., n$.

For the first pass of our solution algorithm, we must first decide whether n is even or odd. If n is even, then there "ought" to be an equal number of even and odd numbers in A, but one is missing: if there are more even numbers than odd numbers in A, then an odd one is missing; otherwise, an even is missing. If n is odd, then there "ought" to be one more odd number than even number in A, but one is missing: if there are as many even as odd numbers in A, then an odd number is missing; otherwise, an even number is missing.

The pass is simple, but takes two steps. In the first step, simply iterate over the least significant bit of each integer in A, counting the number of 1's. This will determine whether the missing number is even or odd. Next, iterate over B, checking the least significant bit of the numbers it refers to in A: if the number in A ends in 1, it is odd; if we've decided an odd number is missing, remove this entry from B. If the number in A ends in 0, it is even; if an even number is missing, remove the associated entry from B. The array B will then only index into the elements of A whose even-ness is the same as that of the missing value.

This pass will read $2n$ entries from A.

The second (and subsequent) passes are modeled after the first but look at increasingly more significant bits. For each pass, we must first calculate how many 1s and 0s are required in the specified column. This can be deduced solely by n's value and the bits we have already inspected. For example, in the second pass we will already have eliminated half the numbers in A (by removing their indices from B). Through simple algebra, we can use this fact coupled with the value of n to calculate the desired number of 1s and 0s in the second-least significant digit of each number in A.

After making this calculation, we can then iterate over B, using its values to index into A (looking only at the second-least significant bit). After calculating the total number of 1s, we know whether a 1 or 0 is missing; we then iterate through B again, deleting its entry if we have determined its associated entry in A to be invalid.

The second such pass will read under $\frac{2n}{2} = n$ entries from A. The third pass will read under $\frac{2n}{4} = \frac{n}{2}$.

When 1 element remains in B, we can read the entire number it indexes in A. Using algebra involving n, we can determine whether to flip its most significant bit or two. The resulting number (which was read in $O(\lg n)$ time from A) is our result.

We can use the master method to analyze the running time of the recursive part of our algorithm. $T(n) = T(\frac{n}{2}) + \Theta(n)$: the recursion always runs on an array half the original's size, and each pass completes in $O(n)$ time. In the generalized master method, $T(n) = aT(\frac{n}{b}) + f(n)$, $a = 1$, $b = 2$, and $f(n) = \Theta(n)$. $\log_b a = \log_2 1 = 0$, so case 3 of the master method applies: $T(n) = \Theta(f(n)) = \Theta(n)$.

The final part of the algorithm (that is, the reading of the final integer) only takes $O(\lg n)$ time; thus, the entire algorithm runs in $O(n)$ time.

3. Exercise 7.4-5

We are asked to evaluate the running time of Quicksort when we only call Partition on sections of the array larger than k elements and use Insertion-Sort on the "nearly"-sorted resulting array.

As with calculating classic QUICKSORT's running time, we will start by creating an indicator random variable:

$$X_{ij} = \mathrm{I}\{z_i \text{ is compared to } z_j\}$$

The total number of comparisons within the PARTITION algorithm is:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

Using linearity of expectation (still following the regular QUICKSORT running time calculation):

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\} \end{aligned}$$

Here is where this combination of algorithms differs from straightforward QUICKSORT: the probability that z_i is compared to z_j. In regular QUICKSORT, this would be the probability that either z_i or z_j is chosen as pivot ($\frac{2}{j-i+1}$). In our modified version, we also have to take into account that if $j - i \leq k$ the items will not be compared and so the probability of their being compared is 0.

$$\begin{aligned} \mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\} \\ &= \sum_{i=1}^{n-1} \left[\sum_{j=i+1}^{k+i-2} 0 + \sum_{j=k+i-1}^{n} \frac{2}{j-i+1}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} - \sum_{i=1}^{n-1} \sum_{j=i+1}^{k+i-2} \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n-1} \sum_{m=1}^{n-i} \frac{2}{m+1} - \sum_{i=1}^{n-1} \sum_{m=1}^{k-2} \frac{2}{m+1} \end{aligned}$$

As in the original QUICKSORT proof, the minuend is $\Theta(n \lg n)$. And the subtrahend, it is easy to see, is $\Theta(n \lg k)$. This is enough to get the total running time of the QUICKSORT half of our algorithm:

$$\begin{aligned}\text{running time} &= \Theta(n \lg n) - \Theta(n \lg k) \\ &= \Theta(n \lg n - n \lg k) \\ &= \Theta(n \times (\lg n - \lg k)) \\ &= \Theta(n \lg \frac{n}{k})\end{aligned}$$

The INSERTION-SORT part of the algorithm is much easier to analyze. There are at most $\frac{n}{k}$ sorts being done, each of maximum size k^2. So the total running time will be $\Theta(k^2 \times \frac{n}{k}) = \Theta(nk)$.

Combining the two:

$$\begin{aligned}\text{total running time} &= \Theta(n \lg \frac{n}{k}) + \Theta(nk) \\ &= \Theta(nk + n \lg \frac{n}{k})\end{aligned}$$

...and so that is the running time of our new algorithm.

The next part of the question asks how to pick k. In theory, we would want our running time to grow as slowly as possible. So we can rearrange the equation:

$$\begin{aligned}nk + n \lg \frac{n}{k} &= n\,(k + \lg n - \lg k) \\ &= n\,(\lg n + (k - \lg k))\end{aligned}$$

We can see that we need a k which grows more slowly than $\lg n$, that is, $k = O(\lg n)$.

In practice, we know that k must be a constant: for low values of k, INSERTION-SORT is faster than QUICKSORT; for higher values of k, QUICKSORT is faster. This value of k will be dependent on each particular implementation of the algorithm.

For example, on average for all possible orderings of a nine-element array, INSERTION-SORT will perform 45 comparisons and 18 exchanges, while QUICKSORT will perform only 34.95 comparisons but 33.98 exchanges. One could assume that for most programming languages and hardware architectures an exchange would take significantly longer than a comparison (hence $k > 9$), but how much longer? Also, running an algorithm on all possible permutations of nine-element arrays takes a matter of seconds, but since the number of permutations grows factorially, how long would it take to actually run all possible permutations of, say, fifteen-element arrays? Such testing is not practical.

Moreover, there is a false assumption made in the above reasoning. Even if INSERTION-SORT is faster than QUICKSORT for an array of k elements, which would be better: Using

QUICKSORT to produce many arrays with an absolute maximum of k elements (most of these arrays being much smaller, i.e., QUICKSORT did too much work), or using QUICKSORT to produce fewer arrays with an absolute maximum of more than k elements but with an average of k (or maybe fewer) elements?

Because of all these complications, the only practical approach is to take a bunch of problem-domain arrays and try sorting them with different values of k. A search on Google would suggest that $k = 15$ is a decent value in a C implementation on x86 architecture.

4. Problem 7-4

a. Argue that QUICKSORT' sorts correctly

The basic QUICKSORT calls itself on the left half of the array ($A[p\,..\,q-1]$) and then the right half of the array ($A[q+1\,..\,r]$). QUICKSORT' calls itself on the left half of the array ($A[p\,..\,q-1]$) and then iterates; before iterating, it sets $p = q+1$. By setting $p = q+1$ and then iterating back to the beginning of the function (reminder: its signature is QUICKSORT'(A, p, r)), it is essentially calling QUICKSORT'$(A, q+1, r)$: calling itself on the right half of the array ($A[q+1\,..\,r]$).

b. $\Theta(n)$ scenario

If the array is already sorted, both the original QUICKSORT and this problem's QUICKSORT' would make the same mistake: call PARTITION to move the last element to the last partition, and then recursively call PARTITION to move the second-last element to the second-last position, and so on. This would create a stack of size n.

c. Worst-case stack depth $\Theta(\lg n)$

To achieve a worst-case stack depth of $\Theta(\lg n)$, we can simply always recurse on the smaller half of the array and iterate over the larger.

```
QUICKSORT'($A, p, r$)
1 while $p < r$
2       do ▷ Partition and sort smaller subarray.
3          $q \leftarrow$ PARTITION($A, p, r$)
4          if $q - p > r - q$
5             then QUICKSORT'($A, q+1, r$)
6                  $r \leftarrow q - 1$
7             else QUICKSORT'($A, p, q-1$)
8                  $p \leftarrow q + 1$
```

It is trivial to prove that the stack depth grows as slowly as $\Theta(\lg n)$. Simply notice that at each recursion, QUICKSORT' is being called on an array of at most half the original array's size.

5. Problem 9-1

a. Sort numbers, list i largest

```
SORT-THEN-LIST($A, i$)
1  $len \leftarrow$ LENGTH($A$)
2  MERGE-SORT($A, 1, len$)
3  return $A[len - i + 1 \,..\, len]$
```

The running time is trivial to analyze: MERGE-SORT's running time $O(n \lg n)$ and building the return array takes $O(i)$ time. Since $i \leq n$, the entire algorithm is $O(n \lg n)$.

b. Build max-priority queue and call EXTRACT-MAX i times

```
BUILD-QUEUE-AND-EXTRACT($A, i$)
1  BUILD-MAX-HEAP($A$)
2  for $j \leftarrow i$ to 1
3       do $R[j] \leftarrow$ HEAP-EXTRACT-MAX($A$)
4  return $R$
```

This algorithm's running time is also easy to analyze. The running time of BUILD-MAX-HEAP is $O(n)$ and the running time of HEAP-EXTRACT-MAX is $O(\lg n)$. HEAP-EXTRACT-MAX is called i times. So the total running time is $O(n + i \lg n)$.

c. Use an order-statistic algorithm, partition, and sort

```
SELECT-PARTITION-SORT($A, i$)
1  $len \leftarrow$ LENGTH($A$)
2  SELECT($A, 1, len, i$)
3  MERGE-SORT($A, len - i + 2, len$)
4  return $A[len - i + 1 \,..\, len]$
```

This algorithm is divided into distinct parts; the running time of each part can be determined separately.

Line 1 calculates the length of A. It runs either in $O(n)$ time or $O(1)$ time, depending on the array implementation. As we will see, this line's running time is insignificant.

Line 2 partitions A so that the i largest elements are at the end. As discussed in class and in the book, SELECT runs in $O(n)$ time.

Line 3 simply sorts the end of A. It sorts $i - 1$ elements, and so its running time is $O(i)$.

Finally, line 4 must iterate over i elements; its running time is $O(i)$.

Simple algebra lets us conclude that the total running time is $O(n) + O(i)$. And since $i \leq n$, the total running time is simply $O(n)$.