ECE368 Homework #2

Solutions

1. Does the bidirectional bubble sorting algorithm reduce the number of swaps when compared to the original bubble sort? Does it reduce the total number of comparisons required to sort an array? Justify your answers.

Solution: The number of swaps would remain the same, because both bubble sort and bidirectional bubble sort allow only swapping of adjacent elements. The total number of comparisons required to sort an array would be reduced for the following reasons:

- In bi-directional bubble sort, small elements can be bubbled to the correct position in a few (backward) passes. In contrast, small elements can move only one position at a time in the forward passes of a regular bubble sort. Hence, bidirectional bubble sort cuts down the total number of iterations and hence the total number of comparisons.

- For a group of contiguous elements that are in the correct order but not in the right positions, forward passes of a bubble sort over these elements result in only comparisons and no swaps. Consider for example an array that has the first *n-1* elements in the correct order and the smallest element occupying the last position in the array. In each forward pass, only one swap occurs. In contrast, a backward pass results in a swap for every comparison. Consequently, the number of comparisons that result in data swapping increases. As the number of swaps remain the same, the total number of comparisons reduces.

2 Consider the following procedure acting on an array $A[1..n]$.

```
INSERTION_SORT($A[1..n]$)                    Cost     Times
1.     for $j \leftarrow 2$ to $n$                  $C_1$      $n$
2.         key $\leftarrow A[j]$                    $C_2$      $n-1$
3.         $i \leftarrow j-1$                       $C_3$      $n-1$
4.         while $i > 0$ and $A[i] > key$           $C_4$      $\sum_{j=2}^{n} t_j$
5.             $A[i+1] \leftarrow A[i]$             $C_5$      $\sum_{j=2}^{n}(t_j-1)$
6.             $i \leftarrow i-1$                   $C_6$      $\sum_{j=2}^{n}(t_j-1)$
7.         $A[i+1] \leftarrow$ key                  $C_7$      $n-1$
```

(a) Let t_j denote the number of times the while loop test in line 4 is executed for that value of j. Fill in for each line of instruction, the number of times the instruction is executed.

(b) Derive the expression for the running time of INSERTION_SORT in terms of n, C_i, and t_j.

Let $T(n) =$ running time of INSERTION_SORT.

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4\sum_{j=2}^{n} t_j + C_5\sum_{j=2}^{n}(t_j-1) + C_6\sum_{j=2}^{n}(t_j-1) + C_7(n-1)$$

(c) What is t_j for the best-case scenario, i.e., when the running time of the algorithm is the smallest. Use that to derive the expression for the best-case running time of INSERTION_SORT in terms of n and C_i. What is the best-case time complexity of INSERTION_SORT using the big-O notation?

The array is already sorted. All t_j are 1. Therefore,

$$\begin{aligned} T(n) &= C_1 n + C_2(n-1) + C_3(n-1) + C_4(n-1) + C_7(n-1) \\ &= (C_1 + C_2 + C_3 + C_4 + C_7)n - (C_2 + C_3 + C_4 + C_7). \end{aligned}$$

$T(n) = O(n)$. (In fact, $T(n) = \Theta(n)$.)

(d) What is t_j for the worst-case scenario, i.e., when the running time of the algorithm is the largest. Use that to derive the expression for the worst-case running time of INSERTION_SORT in terms of n and C_i. What is the worst-case time complexity of INSERTION_SORT using the big-O notation?

The array is in reverse sorted order. We have to compare the j-th element with the previous $(j-1)$ elements. We also need an additional test to get out of the while-loop. Therefore, $t_j = j$. Hence,

$$\begin{aligned} T(n) &= C_1 n + C_2(n-1) + C_3(n-1) + C_4\sum_{j=2}^{n} j + C_5\sum_{j=2}^{n}(j-1) + C_6\sum_{j=2}^{n}(j-1) + C_7(n-1) \\ &= C_1 n + C_2(n-1) + C_3(n-1) + C_4\left(\frac{n(n+1)}{2} - 1\right) + C_5\frac{n(n-1)}{2} + C_6\frac{n(n-1)}{2} + C_7(n-1) \\ &= \left(\frac{C_4 + C_5 + C_6}{2}\right)n^2 + \left(C_1 + C_2 + C_3 + \frac{C_4 - C_5 - C_6}{2} + C_7\right)n - (C_2 + C_3 + C_4 + C_7). \end{aligned}$$

$T(n) = O(n^2)$. (In fact, $T(n) = \Theta(n^2)$.)

3 Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, the pair (i, j) is called an *inversion* of A.

(a) List all the inversions of the array $\langle 2,3,8,6,1 \rangle$.

$(1,5)$, $(2,5)$, $(3,4)$, $(3,5)$, and $(4,5)$. Note that the inversions are specified by indices rather than by the values in the array. Also note that the array starts with index 1.

(b) What array with elements from the set $\{1,2,...,n\}$ has the most inversions? How many does it have?

The array with elements from $\{1,2,...,n\}$ with the most inversions is $\langle n,n-1,n-2,...,2,1 \rangle$. For all $1 \le i < j \le n$, there is an inversion (i, j). The number of inversions with $i = 1$ is $n-1$. The number of inversions with $i = 2$ is $n-2$. The number of inversions with $i = n$ is $n-n = 0$. Therefore the number of inversions is $(n-1)+(n-2)+...+0 = n(n-1)/2$.

(c) What is the relationship between the running time of `INSERTION_SORT` (see question 1) and the number of inversions in the input array? Justify your answer.

Every execution of the instructions (i.e., data move) within the while-loop eliminates an inversion. Therefore, the run-time of `INSERTION_SORT` is determined by the number of inversions.

(d) Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64n\log_2 n$ steps. For which values of n does insertion sort beat merge sort?

When $n = 1$, no sorting is required. So, we assume that $n \ge 2$ to begin with. Dividing both complexity expressions by $8n$, we need $n < 8\log_2 n$ for insertion sort to be faster. A linear or binary search for n should yield $n \le 43$.

4 Consider the following procedure that performs multiplication of two upper triangular matrices $A[1..n][1..n]$ and $B[1..n][1..n]$.

```
MATRIX_MULTIPLY($A[1..n][1..n]$, $B[1..n][1..n]$)   Cost     Times
1.      for $i \leftarrow 1$ to $n$                  $C_1$    $n+1$
2.          for $j \leftarrow i$ to $n$              $C_2$    $\sum_{i=1}^{n}(n-i+2)$
3.              $c_{ij} \leftarrow 0$                $C_3$    $\sum_{i=1}^{n}(n-i+1)$
4.              for $k \leftarrow i$ to $j$          $C_4$    $\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+2)$
5.                  $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$   $C_5$    $\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+1)$
6.      return $C$                                   $C_6$    $1$
```

As the product $C[1..n][1..n]$ is also upper triangular, we assume that we do not have to be concerned with the strictly lower triangular entries of the matrix.

(a) Fill in for each line of instruction, the number of times the instruction is executed.

(b) Derive the expression for the running time of `MATRIX_MULTIPLY` in terms of n and C_i. What is complexity of the algorithm using the big-O notation?

Let $T(n) =$ running time of `MATRIX_MULTIPLY`.

$$\begin{aligned}
T(n) &= C_1(n+1) + C_2\sum_{i=1}^{n}(n-i+2) + C_3\sum_{i=1}^{n}(n-i+1) + C_4\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+2) + C_5\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+1) + C_6 \\
&= C_1(n+1) + C_2\frac{n(n+3)}{2} + C_3\frac{n(n+1)}{2} + C_4\left(\frac{n(n+1)(2n+1)}{12} + 3\frac{n(n+1)}{4}\right) \\
&\quad + C_5\left(\frac{n(n+1)(2n+1)}{12} + \frac{n(n+1)}{4}\right) + C_6 \\
&= (C_4+C_5)\frac{n^3}{6} + (\frac{C_2}{2} + \frac{C_3}{2} + C_4 + \frac{C_5}{2})n^2 + (C_1 + \frac{3C_2}{2} + \frac{C_3}{2} + \frac{5C_4}{6} + \frac{C_5}{3})n + (C_1 + C_6)
\end{aligned}$$

$T(n) = O(n^3)$. (In fact, $T(n) = \Theta(n^3)$.)