CMPS 101
Homework Assignment 1
Solutions

1. p.27: 2.2-2

Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element of A and exchange it with $A[2]$. Continue in this manner for the first $n-1$ elements of A. Write pseudo-code for this algorithm, which is known as *selection sort*. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort in Θ-notation.

SelectionSort(A)

```
1.  n = length[A]
2.  for i = 1 to n-1
3.      k = i                    // find the index k of the minimum element in A[i...n]
4.      for j = i+1 to n
5.          if A[j] < A[k]
6.              k = j
7.      A[i] ↔ A[k]              // swap A[i] with A[k]
```

The loop 2-9 first locates the minimum element in the subarray $A[i \cdots n]$ (lines 3-6), then exchanges it with the element $A[i]$ (line 7). Two loop invariants are maintained: (1) each element in subarray $A[1 \cdots (i-1)]$ is less than or equal to each element in $A[i \cdots n]$, and (2) the subarray $A[1 \cdots (i-1)]$ is sorted in increasing order. The correctness of SelectionSort follows from these two invariants, for when loop 2-9 is complete, invariant (1) implies that $A[n]$ is greater than or equal to each element in the subarray $A[1 \cdots (n-1)]$, which is itself sorted by invariant (2). Hence the full array $A[1 \cdots n]$ is at that point sorted. This explains why it is unnecessary to continue loop 2-7 until $i = n$. The subarray $A[n]$ is already sorted, since it contains only one element.

Observe that on the i^{th} iteration of loop 2-7, the inner loop 4-6 executes exactly $(n-i)$ times, and on each such execution, exactly one array comparison is performed (line 5). Thus in all cases (best, worst, average) the number of comparisons done by SelectionSort is

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

Therefore the run time of SelectionSort is (in all cases) $\Theta(n^2)$.

2. p.37: 2.3-5

Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence *A* is sorted, we can check the midpoint of the sequence against *v* and eliminate half of the sequence from further consideration. *Binary search* is an algorithm that repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudo-code, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

Recursive version:

BinarySearch(*A*, *p*, *r*, *v*) (Pre: subarray $A[p \cdots r]$ is sorted)

```
1.  if p > r
2.          return NIL
3.  else
4.          q = ⌊(p + r)/2⌋
5.          if v = A[q]
6.                  return q
7.          else if v < A[q]
8.                  return BinarySearch(A, p, q − 1, v)
9.          else
10.                 return BinarySearch(A, q + 1, r, v)
```

Iterative version:

BinarySearch(*A*, *v*) (Pre: $A[1 \cdots n]$ is sorted)

```
1.  p = 1, r = length[A], found = false
2.  while p ≤ r and not found
3.          q = ⌊(p + r)/2⌋
4.          if v = A[q]
5.                  found = true
6.          else if v < A[q]
7.                  r = (q − 1)
8.          else
9.                  p = (q + 1)
8.  if found
9.          return q
10. else
11.         return NIL
```

Let t_k denote the length of the subarray being searched on the k^{th} iteration of loop 2-9 (iterative version) or on the k^{th} recursive invocation of BinarySearch (recursive version.) Since the length is halved on each iteration/invocation, we have $t_1 = n$ and $t_k = t_{k-1}/2$, and therefore $t_k = n/2^{k-1}$. In worst case the target *v* is not in the list. The process stops when $t_k = 1$, i.e. $n/2^{k-1} = 1$. Therefore $k = \lg(n) + 1$, which gives the number of iterations (or the recursion depth) in worst case. The running time of BinarySearch is therefore $\Theta(\lg n)$ in worst case.

3. p.39: 2-4abcd

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

a. List the five inversions of the array $(2, 3, 8, 6, 1)$.

The inversions are: (1, 5), (2, 5), (3, 4), (3, 5), and (4, 5). (Note that an inversion is a pair of array **indices**, *not* a pair of array **elements**.)

b. What array with elements from the set {1, 2, 3, ….., *n*} has the most inversions? How many inversions does it have?

The array (*n*, $n-1$, …., 3, 2, 1) has the maximum number of inversions. Observe this array has $n-1$ inversions of the form $(1, x)$, $n-2$ inversions of the form $(2, x)$, $n-3$ inversions of the form $(3, x)$, …., 2 inversions of the form $(n-2, x)$, and finally 1 inversion of the form $(n-1, x)$. Thus the total number of inversions in this array is: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$.

c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

Recall that we measure the running time of an algorithm not in seconds but in the number of basic operations performed. For most sorting algorithms the basic operation is the comparison of two array elements. Let $T(n)$ denote the number of array comparisons performed by insertion sort on input of size *n*, and let $I(n)$ denote the number of inversions in the input array. Now recall the operation of InsertionSort. For each $j = 2$ to *n*, element $A[j]$ is inserted into its proper location in the sorted subarray $A[1\cdots(j-1)]$.

$$A_1 \cdots\cdots A_{j-1}, A_j \cdots\cdots A_n$$

The algorithm steps through $A[1\cdots(j-1)]$ from right to left, shifting elements to the right as it goes, making room for $A[j]$. The number of inversions of the form (i, j) is exactly the number of elements in $A[1\cdots(j-1)]$ which $A[j]$ 'passes over' in reaching it's destination, and this is exactly the number of times that the comparison $A[j] < A[i]$ is true. This process stops when the proper location for $A[j]$ is found, and this happens when either $A[j] < A[i]$ is false, or when we run out of elements in $A[1\cdots(j-1)]$ to compare to. In the first case $\#\text{comparisons} = \#\text{inversions} + 1$, and in the second case $\#\text{comparisons} = \#\text{inversions}$. Thus for each *j*:

$$\#\text{inversions} \le \#\text{comparisons} \le \#\text{inversions} + 1.$$

Summing this relation for $j = 2$ to *n* we get:

$$I(n) \le T(n) \le I(n) + (n-1).$$

In the worst and average cases $I(n) = \Theta(n^2)$, so that $(n-1)$ is a lower order term. Thus the running time of InsertionSort is asymptotically equivalent to the number of inversions in the input array, i.e. $T(n) = \Theta(I(n)) = \Theta(n^2)$.

d. Give an algorithm that determines the number of inversions in any permutation of *n* elements in $\Theta(n \lg n)$ worst-case time. (Hint: Modify merge sort.)

We alter both Merge and MergeSort to return an integer, as well as perform their previous sorting functions. Merge(*A*, *p*, *q*, *r*) returns the number of inversions between the two subarrays $A[p \cdots q]$ and $A[(q+1) \cdots r]$, i.e. it returns a count of the number of times an element in $A[(q+1) \cdots r]$ is less than an element in $A[p \cdots q]$.

Merge(*A*, *p*, *q*, *r*) (Pre: $A[p \cdots q]$ and $A[(q+1) \cdots r]$ are sorted)

1. $n_1 = (q - p + 1)$
2. $n_2 = (r - q)$
3. create arrays $L[1 \cdots (n_1 + 1)]$ and $R[1 \cdots (n_1 + 1)]$
4. for $i = 1$ to n_1
5. $\quad L[i] = A[p + i - 1]$
6. for $j = 1$ to n_2
7. $\quad R[j] = A[q + j]$
8. $L[n_1 + 1] = \infty$, $R[n_2 + 1] = \infty$
9. $i = 1$, $j = 1$, count $= 0$
10. for $k = p$ to r
11. $\quad$ if $L[i] \le R[j]$
12. $\quad\quad A[k] = L[i]$
13. $\quad\quad i = i + 1$
14. $\quad$ else
15. $\quad\quad A[k] = R[j]$
16. $\quad\quad j = j + 1$
17. $\quad\quad$ count $=$ count $+ (n_1 - i + 1)$
18. return count

Observe that each time the test in line 11 is false, count is incremented by the length of the subarray $L[i \cdots n_1]$, which is the set of elements that $R[j]$ must pass over in order to reach its proper location in subarray $A[p \cdots r]$. By the time the algorithm is complete, count is precisely the number of inversions of the form (x, y) in the subarray $A[p \cdots r]$, where $p \le x \le q$ and $q + 1 \le y \le r$. MergeSort(*A*, *p*, *r*) returns the total number of inversions in the subarray $A[p \cdots r]$.

MergeSort(A, p, r)

1. if $p < r$
2. $\quad q = \left\lfloor \frac{p+r}{2} \right\rfloor$
3. $\quad a = \text{MergeSort}(A, p, q)$
4. $\quad b = \text{MergeSort}(A, q+1, r)$
5. $\quad c = \text{Merge}(A, p, q, r)$
6. $\quad$ return $(a+b+c)$
7. else
8. $\quad$ return 0

If $p < r$, the number of inversions in $A[p \cdots r]$ is the sum of the number of inversions in $A[p \cdots q]$, plus the number of inversions in $A[(q+1) \cdots r]$, plus the number of inversions between $A[p \cdots q]$ and $A[(q+1) \cdots r]$. This is exactly what is returned on line 6. If $p \geq r$, then $A[p \cdots r]$ has length at most 1, and therefore contains no inversions. In this case 0 is returned on line 8. The asymptotic run time of this modified MergeSort is the same as the original, namely $\Theta(n \lg n)$, by the same analysis which was performed in class.