Homework 3 Solutions

Question 1

Upper bound: O(n^2)

The outer loop has n iterations, but the inner loop has a variable number of iterations, so we can't just multiply the loop lengths; instead, we need to count the total number of times `swap()` is called:

```
i=1, inner loop has 1 iteration
i=2, inner loop has 2 iterations
i=3, inner loop has 3 iterations
...
i=n-1, inner loop has n-1 iterations
```

Therefore, the total number of iterations of the inner loop is 1 + 2 + 3 + … + n–1 = n(n–1)/2 = O(n^2). Since `swap()` is O(1), the total running time is O(n^2).

Lower bound: Omega(n^2)

The justification is the same. The algorithm *always* does n(n–1)/2 total iterations of the inner loop.

Question 2

Note: This is linear search on an array of length n.

Upper bound: $O(n)$

The code searches an n-long array for the value v. What's the *longest* this can take? Suppose v is not in the array, then the loop will have n iterations. Since the loop body is $O(1)$, the upper bound on the running time is $O(n)$.

Lower bound: Omega(1)

Suppose v is the first entry in A. The code will do a single iteration and finish in $O(1)$ time.

Question 3

Note: This is binary search on an n-long sorted array.

Upper bound: $O(\log n)$

The code starts with $r - p + 1 = n$. It then finds the midpoint, q, between r and p and restricts the search to the half of the array that could contain v. It then repeats this procedure with an updated r or q value. That is, it divides the search region in half with each iteration, stopping when $r = p$. How many times can we divide an n-long array in half before we are down to a single element? Approximately log2(n) times, where log2() is the logarithm base 2. Therefore, the loop has approximately log(*n*) iterations and the loop body is O(1), so the upper bound on the running time is O(log(*n*)).

Lower bound: Omega(log *n*)

Unlike the linear search in Question #2, this search does not exit early; it always iterates until *p* = *r*. Therefore, the number of iterations is determined by *n* alone, and the lower bound is Omega(log(*n*)).

Question 4

Upper bound: O(*n*)

The first loop always does *n* constant time iterations, so it is O(*n*). The second loop does O(log(*n*)) iterations [same reason as for binary search, above]. Since these computations are done sequentially and *n* is “bigger” than log(*n*), the O(*n*) term dominates.

Lower bound: Omega(*n*)

The first loop is Omega(*n*), so the code is *at least* Omega(*n*). Since we know it is O(*n*), and the lower bound can’t be larger than the upper bound, the code is Omega(*n*).

Question 5

Upper bound: O(*n*^(1/2)) (i.e. square root of *n*)

Suppose the inner loop performs *k* iterations. Then the value of `total` is

```
1 + 2 + 3 + ... + k = k(k+1) / 2
```

So, to determine the number of iterations, we need to find the largest value of *k* such that

```
k(k+1) / 2 <= n
```

Well, that means we want *k*(*k*+1) approximately equal to 2*n* or *k* about the square root of 2*n*. Therefore, the total number of iterations is approximately (2*n*)^(1/2), and each iterations is constant time, so the upper bound is O(*n*^(1/2)).

Lower bound: O(*n*^(1/2))

The number of iterations depends on *n* alone. Since it always peforms approximately *n*^(1/2) iterations, the lower bound is the same as the upper bound.