Kyle Daling

CSCI 305

Assignment 4

May 31, 2017

# Question 1

The indicator random variable, lets call it $I$, will be: for a given $i$, $j$ pair, $i < j$ and $A[i] > A[j]$. This would make this $i$, $j$ pair an inversion.

There are $\binom{n}{2}$ possible pairs in the entire array. According to the definition of the combination operator this is equivalent to $\frac{(n-1)n}{2}$.

The probability of occurring an inversion $P(I) = \frac{1}{2}$ since each pair chosen is either an inversion or not an inversion.

Calculating the expected value of the number of inversions would be

$$\begin{aligned} E\left(\sum_{i=1}^{n}\sum_{j=1}^{i-1} P(I)\right) &= \sum_{i=1}^{n}\sum_{j=1}^{i-1} E(P(I)) = \sum_{i=1}^{n}\sum_{j=1}^{i-1} \frac{1}{2} \\ &= \sum_{i=1}^{n} \frac{i-1}{2} = \frac{1}{2}\sum_{i=1}^{n} i - 1 \\ &= \frac{1}{2} \times \frac{n(n-1)}{2} = \frac{n(n-1)}{4} \end{aligned}$$

The outer loop runs over all elements in the array, and the inversions may happen at any index $j$ that is lower than $i$.

The answer we got from the summation matches the expected answer by using n choose k and multiplying it by the probability of a single inversion.

# Question 2

```
// Preparation
// Takes array A as input
prepare(A, k)
  let C[0..k] be a new array
  for i = 0 to k:
    C[i] = 0
  for j = 1 to A.length:
    C[A[j]] = C[A[j]] + 1
  for i = 1 to k:
```

```
    C[i] = C[i] + C[i - 1]
  return C

// Return the number of integers that are between a and b
// Assume that (b + 1) and (a - 1) are valid elements in array C
in_range(C, a, b):
  return C[b + 1] - C[a - 1]
```

The inner part of each for loop in prepare runs in constant time since each operation is just a simple array operation. Therefore the body of each for loop runs with $\Theta(1)$ runtime. The body of the first for loop will execute $k$ giving a total runtime of $\Theta(k)$. The body of the second for loop will execute $A.\text{length} = n$ times giving a total runtime of $\Theta(n)$. The body of the last for loop will execute $k - 1$ times giving total runtime of $\Theta(k)$. Therefore the total runtime of the preparation phase of the algorithm will be $\Theta(n + k)$.

The in_range portion of the algorithm will run in $\Theta(1)$ since it only needs to do simple array accesses and subtraction.

Therefore the algorithm runs in $\Theta(n + k)$.

# Question 3

a. Counting sort would be a poor choice. Counting sort has runtime $\Theta(n + k)$ where $k$ is the number of possible elements. Since the elements are in the range 0 to $n^2 - 1$ this would make $k = n^2 - 1$ giving counting sort a total runtime of $\Theta(n^2)$. By asymptotic runtime this is the slowest algorithm.

   Radix sort has runtime of $\Theta(d(n + k))$ where $d$ is the number of digits being sorted, and $n + k$ is the assumed runtime of the linear runtime stable sorting algorithm. However, the only sorting algorithm we know that is stable and can have linear runtime is counting sort, which we just proved has $\Theta(n^2)$ runtime above. Therefore radix sort will also run in $\Theta(n^2)$.

   Of the remaining algorithms insertion sort and quick sort have the worst case of $\mathcal{O}(n^2)$. However, variants of quick sort exist that make the worse case extremely unlikely. Further I am assuming that the sequence of elements we are being given is in random order further reducing the likelihood of quick sort's worst case. Therefore we will only get rid of insertion sort at this step.

   Heapsort and merge sort both run in $\Theta(n \lg n)$. Since we are precluding quick sort's worst case, we will instead use the average case of $\Theta(n \lg n)$. Therefore we have 3 algorithms within the same runtime class. Quick sort and merge sort both require $\mathcal{O}(n)$ additional space. Heap sort only requires $\Theta(1)$ additional space. Therefore by the metrics that we have learned in class and the constraints imposed by the problem, heap sort will be the most efficient algorithm for sorting $n$ elements.

b. I would still choose heapsort if the range of elements was from 0 to $2^n - 1$. Heapsort will still have the maximally efficient $\Theta(n \lg n)$ runtime. Counting sort and radix sort would have

runtime $\Theta(2^n)$. The runtime for the remaining algorithms will not change since they are operating on an array of the same size.

# Question 4

```
is_bst(node):
  a = in_order_traverse(node)
  for i=2 to n:
    if a[n] < a[n-1]:
      return False
  return True
```

The algorithm starts be completing an in_order traversal of the tree. This takes $\Theta(n)$ and uses $\Theta(n)$ space where $n$ is the number of elements in the tree. Next the algorithm iterates over the array returned by the in_order traversal. If the tree is a valid binary search tree, the array returned should be in ascending order. In the best case the second for loop terminates early and the runtime of the for loop is constant giving a lower bound of $\Omega(n)$. In the worse case the second for loop runs to completion and the runtime incurs an additional $\Theta(n)$ bringing the total up to $\mathcal{O}(2n) = \mathcal{O}(n)$. Since the algorithm is $\Omega(n) = \Theta(n)$, this makes the total runtime of the algorithm $\Theta(n)$ and the space complexity an additional $\Theta(n)$.