Practice Set 4: Solutions

.

Problem 1:

1. We saw in class that the run time of Counting sort on input of size n is $O(n+m)$ where the numbers in the array are all at most m. If the numbers are less than n^2, the overall runtime is $O(n^2+n) = O(n^2)$.
2. If there are n^2 numbers in the range 0 to n^2 then the run time of Counting sort is $O(n^2+n^2) = O(n^2)$.
3. If each number is in binary and less than 2^n, then the number requires at most n digits. We saw that Radix sort runs in time $O(d(n+r))$ where d is the number of digits and r is the radix. Therefore Radix sort in this case runs in time $O(n(n+2)) = O(n^2)$.
4. In this case, each binary number requires at most $\log_2(n^2)$ digits, which is $2\log_2(n)$. Note that $r = 2$. Therefore Radix sort runs in time $O(2\log_2(n)(n+2)) = O(n \log n)$.
5. In this case, each decimal number requires at most $\log_{10}(n)$ digits and $r = 10$. Therefore Radix sort runs in time $O((\log_{10} n)(n+10)) = O(n \log n)$.
6. In this case, $r = 10$ and $d = 2$. Therefore Radix sort runs in time $O(2(n+10)) = O(n)$.
7. For 100 numbers in the range 0 to n, Counting sort runs in time $O(n+m) = O(100+n) = O(n)$.
8. Insertion sort on 100 numbers runs in time $O(100^2) = O(1)$, which is constant time. The range of the input is not relevant.
9. For n numbers in the range 0 to 100, Counting sort runs in time $O(n+m) = O(n+100) = O(n)$.
10. Insertion sort on n numbers runs in time $O(n^2)$ in the worst-case, regardless of the range of the input.
11. Radix sort on n real numbers, where each number is of the form $xxx.xx$ can be implemented as the usual radix sort, where $d = 5$. In this case, the runtime is $O(d(n+r)) = O(5(n+10)) = O(n)$.

Problem 2:

- When an algorithm performs steps that are based on some random decision, then the actual runtime of the algorithm is **random**. This means that the runtime may be fast or slow, depending on the random choices that are made. The runtime is in fact a **random variable**, which has an **expected value**. The expected runtime is similar to a weighted average over all possible runtimes and how likely they are to occur. The worst-case runtime of an algorithm is the maximum number of steps that algorithm can take, over all possible random choices and over all possible inputs.
- Design bucket sort with n buckets distributed over the range $-10 \ldots 10$. Each bucket covers a range of length $20/n$. The first bucket goes from $-10 \ldots -10 + 20/n$. Since the numbers are uniformly distributed, there is an equal chance that a number falls in any one bucket. Therefore the *expected* number of elements in each bucket is $E(n_i) = n/n = 1$. The expected time to sort each bucket is constant, and therefore the time to sort all n buckets is $O(n)$. Bucket sort then performs a final pass through all n buckets, outputting the elements in sorted order. This takes time $O(n)$. Therefore the overall **expected** runtime is $O(n)$.

  The worst-case runtime is when all numbers fall into one bucket, which could happen because the numbers are random. If this happens, insertion sort will take $O(n^2)$ just to sort that one bucket. The overall runtime is then $O(n^2)$. MergeSort and Heapsort both have a worst-case runtime of $O(n \log n)$, which is a big improvement in the worst-case. Note that we could **not** use Counting sort or Radix sort here, because the numbers are not integers.
- If the input is not uniformly distributed, then we *cannot conclude* that each bucket is expected to hold *one* number. Therefore, we *cannot* conclude that the expected runtime of Bucket sort is $O(n)$.

Problem 3:

The expected runtime of bucket sort applies when the input is uniformly distributed over a range. It is very unlikely that class grades are uniform. There are probably a large number of grades around the average, and very few grades near 100 and 0. Therefore using Bucket sort is not a good idea, since we can't reasonably conclude that the expected runtime will be $O(n)$, and instead it is reasonable to assume that many numbers would end up in just a few buckets.

Each number is of the form $xxx.xx$ where each x represents a digit $0, 1, \ldots 9$. If we multiply each number by 100, then the new grades are in the range 0 to $10,000$, and they are now *integers*. Therefore we can use Counting sort, where $m = 10,000$. The runtime to sort the grades using Counting sort is $O(n+m) = O(n+10,000) = O(n)$. Once the grades are sorted, we simply divide each number by 100 in order to reproduce the original grades. This does not affect the sorted order, and therefore the final result is the original grades in sorted order. Therefore it is possible to use Counting Sort to get a worst-case runtime of $O(n)$. Similarly, we could use Radix sort on the grades. Once they have been multiplied by 100, each number is in the range 0 to $10,000$ and therefore $d = 5$ and $r = 10$. Therefore the worst-case runtime of Radix sort is $O(5(n+10)) = O(n)$.

Problem 4:

If the numbers have any number of decimal points, then we cannot convert them to an integer, and therefore cannot use Counting Sort, Radix Sort. We did not see a sorting algorithm in class that sorts this type of input in linear time. Therefore, the best case option would be to use MergeSort or Heapsort, which have runtimes of $\Theta(n \log n)$.

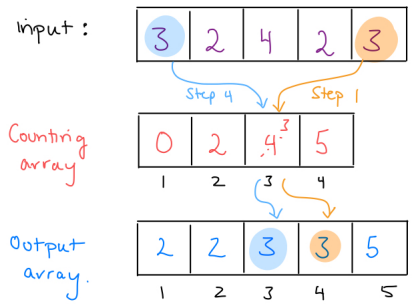
Problem 5:

Since there are n fractions, we could use any comparison-based sort, such as Mergesort or Quicksort and sort the numbers in $O(n \log n)$ time. However, since we have a bound on the denominator, we might be able to do better. If we could convert every fraction to a whole number, we could use Counting sort. Any fraction in the list has a denominator ≤ 1000. Thus by multiplying each fraction by $1000! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot 1000$, each fraction is converted to a whole number. Now we can use Counting sort to sort these new larger numbers. Note that the maximum value is 1000!. Thus the runtime will be $O(n + 1000!) = O(n)$. Although the constant of the big-oh is huge, it is nevertheless still linear. Once the numbers have been sorted, we can simply divide each number by 1000! to reproduce the original values.

Similarly, we could use Radix sort to sort the converted numbers. In this case, $r = 10$ and $d = \log_{10} 1000!$. Therefore the runtime is $O(d(n+r)) = O(\log_{10} 1000!(n+10)) = O(n)$.

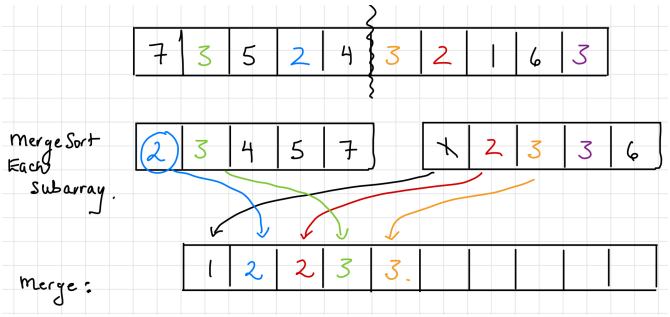
Problem 6:.

In the third step of counting sort, the elements of the array A are removed from *back* to *front* and placed in the output array in the position based on the number of elements that are smaller or equal to that element. For example, element 3 in the array below currently has rank 4 because there are 4 elements less than or equal to 3. It goes into position 4 in the output array (the orange step below). Because we process the input array from back to front, when the next 3 is encountered (in pink), it will be inserted to the left of the previous 3 in the output array.(the pink step below). The result is that counting sort is a *stable* sorting algorithm: duplicate items appear in the output array in their original order.



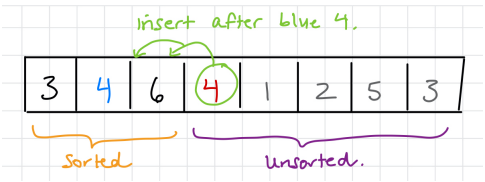
Mergesort and Insertion sort are not necessarily stable. We could update them slightly in order to make sure they produced stable output.

Mergesort: We could update the *Merge* step so that when it compares the front elements of each subarray, it breaks ties by selecting the element from the front of the left subarray first, as shown in the illustration on the left below. This only requires a small update to the pseudo-code of Merge, shown on the right below:



Merge(A, s, q, f)
- Let $L[]$ and $R[]$ be new arrays
- Copy elements from A between s and q into array $L[]$
- Set $L[q-s+2] = \infty$
- Copy elements from A between $q+1$ and f into array $R[]$
- Set $R[f-q+1] = \infty$
- $i = 1, j = 1$
- for $k = s$ to f do:
  - if $L[i] \leq R[j]$
    - $A[k] = L[i]$
    - $i++$
  - else $A[k] = R[j]$
    - $j++$

Insertion sort: If we were careful in the implementation of Insertion sort to ensure that when a duplicate element is encountered, that the new element is placed to its right, then the result will be a stable algorithm. Note that the original pseudo-code for insertion sort was already implemented in this way:



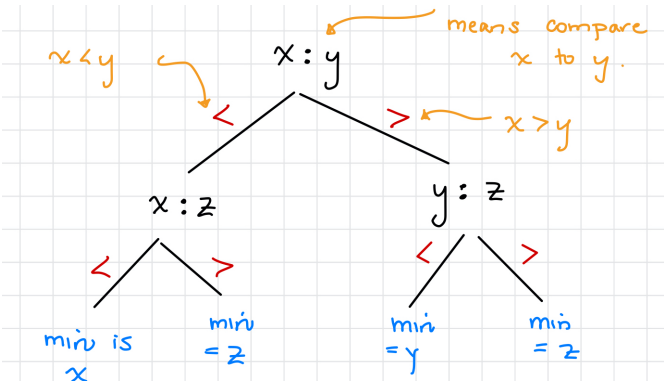
InsertionSort($A[1 \ldots n]$)
- for $i = 2$ to n :
  - $j = i$
  - for $j = i$ down to 2:
    - if $A[j] < A[j-1]$
      - Swap $A[j]$ and $A[j-1]$
    - else break

Bubble sort: The original implementation of bubble sort that we saw in class is shown below. Not that when duplicate elements are encountered, they are not swapped. Therefore a duplicate on the "right", stays on the right. Therefore Bubble sort is indeed stable.

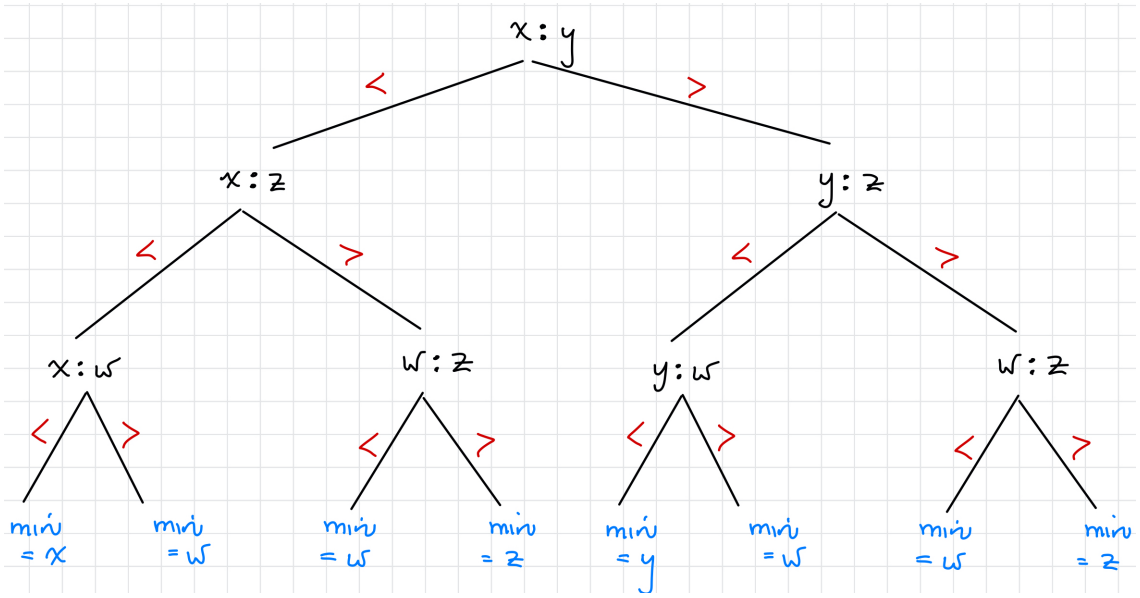
```
made-swap = true
while(made-swap)
     made-swap = false
     for i = 1 to n-1
         if A[i+1] < A[i]
             Swap A[i] and A[i+1]
             made-swap = true
```

Problem 7:

Assume the three numbers are x, y, z and they are distinct. The decision tree for finding the min is shown below. Note that at each node in the tree, we simply compare the next element to the current min, and that the height of the tree is 2.



In order to find the min of x, y, z, w we use a similar approach. The decision tree is shown below. The height in this case is 3:



Given a set of n numbers, it *not possible* that there is a path in that tree that has length less than $n-1$. The argument is by contradiction. Since there are n numbers, we cannot conclude that any one number is the max unless it has at least been compared to all the other numbers. That would take at least $n-1$ comparisons. So all paths must have length at least $n-1$. This means that the height of the tree is *at least* $n-1$. Therefore any algorithm that finds the min in a set of n numbers must run have a run time of at least $\Omega(n)$.

In fact, it is possible to create a decision tree for finding the max that has height exactly $n-1$. The examples above demonstrate this fact.

Problem 8:

Suppose we use $n/2$ buckets in bucket-sort. Since the numbers are over the range $0 \leq x \leq 1$, each bucket has "size" $\frac{1}{n/2} = 2/n$. Since the data is uniformly distributed, the *expected* number of elements in each bucket is 2 (instead of 1). Nevertheless, this is still a constant. Thus we expect n_i to be a constant, in other words $E(n_i) = constant$ As mentioned in class, this means $E(n_i^2) = constant$. So the runtime for insertion sort on *each* bucket is expected to be a constant. There are $n/2$ buckets and so the total runtime to sort them all is $O(n/2) = O(n)$. The last step of bucket sort is to loop through all the buckets and output the sorted numbers. Since there are $n/2$ buckets, this last step also runs in time $O(n)$. Therefore the entire algorithm runs in time $O(n)$.

Next, suppose we have $\sqrt{n}$ buckets. Then each bucket has "size" $\frac{1}{\sqrt{n}}$. For uniformly distributed numbers, the expected number of elements in each bucket is $\frac{n}{\sqrt{n}} = \sqrt{n}$. Insertion sort on each bucket is *not* expected to be a constant, since there are not a constant number of elements per bucket. In fact one can show that

the expected runtime of insertion sort is $O(n)$ when there are expected be be $\sqrt{n}$ elements. Since there are $\sqrt{n}$ buckets in total, sorting *all* buckets takes time $O(n^{1.5})$. The last step of bucket sorts loops through the $\sqrt{n}$ buckets, taking time $O(n^{0.5})$. Nevertheless, the overall runtime is $O(n^{1.5})$.

Suppose we have n^2 buckets. Then each bucket has "size" $\frac{1}{n^2}$. For uniformly distributed numbers, the expected number of elements in each bucket is $\frac{n}{n^2} = \frac{1}{n}$. Insertion sort on each bucket is expected to be less than 1. Therefore the expected time of insertion sort on each bucket is constant. Since there are n^2 buckets, sorting *all* buckets takes time $O(n^2)$. The last step of bucket sorts loops through the n^2 buckets, taking time $O(n^2)$. Therefore the overall runtime is $O(n^2)$.

Problem 9:

First we find the maximum value:

```
m = A[1]
for i = 2 to n
    m = max(m, A[i])
```

Next we count the number of occurrences of each element. In the solution below, we assume the input consists of numbers ≥ 1, so the array C starts at 1.

```
Initialize array C[1, ..., m] = 0
for i = 1 to n
    C[A[i]] = C[A[i]] + 1
```

Next we finalize the array C:

```
for i = 2 to m
    C[i] = C[i] + C[i-1]
```

Next we fill in an output array B:

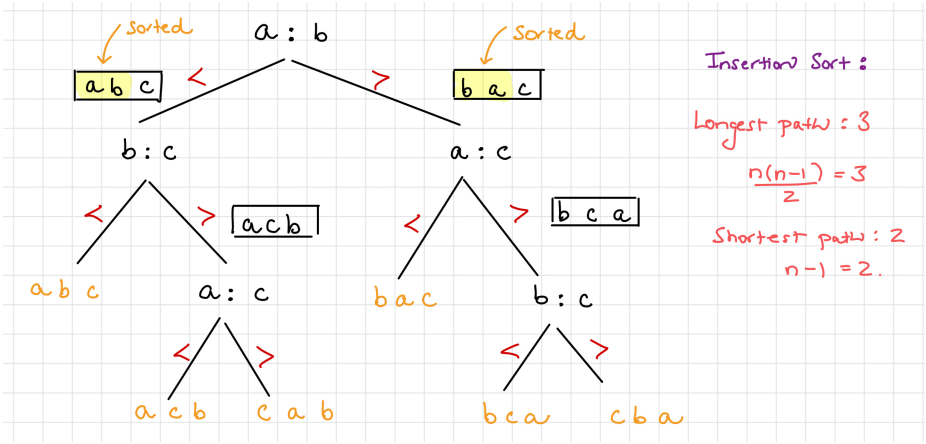
```
Initialize output array B[1, ..., n]
for i = n down to 1
    index = C[A[i]]
    B[index] = A[i]
    C[A[i]] = C[A[i]] - 1
```

Copy Result back into A:

```
ArrayCopy(A,B)
```

Problem 10

- Below is the decision tree that represents the execution of insertion sort on $\{a, b, c\}$. The shortest path in the above tree is 2 and the longest path is 3.



- If the input is already sorted, we know that insertion sort only performs $n-1$ comparisons. Therefore the decision tree for insertion sort will have a path of length $n-1$, representing the best-case scenario. We also saw that the *exact* number of comparisons in the worst case is $n(n-1)/2$. Therefore the longest path of the decision tree for insertion sort has length $n(n-1)/2$.

- Selection sort on the other hand, always performs the same number of comparisons, regardless of input. We saw in week 1 that the number of comparisons performed by selection sort is always $(n-1)+(n-2)+\ldots+1 = n(n-1)/2$. Therefore *all* paths in the decision tree for Selection sort will have length $n(n-1)/2$.
- In comparison-based sorting, we need *at least* $n-1$ comparisons to determine the sorted list. This can be argued by contradiction. If in fact we used *less* than $n-1$ comparisons, there there is at least one pair x_i, x_j that was never compared. So we cannot be certain of the final sorted order. This means that *any* sorting algorithm will never have a path that is shorter than $n-1$. But that does *not* mean that the *height* of the tree is $n-1$ (recall that we showed the height is $\Omega(n \log n)$). Instead, it just means there is *no path* in the tree that is shorter than $n-1$.
- In order to find the median of a set, we need *at least* $n-1$ comparisons to determine the sorted list. This can be argued by contradiction. If in fact we used *less* than $n-1$ comparisons, there there is at least one pair x_i, x_j that was never compared. So we cannot be sure which element is the median without knowing if x_i or x_j is larger. This means that *any* median-finding algorithm will never have a path that is shorter than $n-1$.

Problem 10

Given a set of n elements, we need to carry out at least $n-1$ comparisons in order to determine if an element x is the median. This is because we need to ensure that there are the same number of elements less than and greater than x. Therefore any comparison-based algorithm for finding the median will need at least $n-1$ comparisons.

Problem 11

Given n elements in the range n^2 elements, we could use Radix sort to sort the numbers and then select the element at the index of the median. In this case, $d = \log_{10} n^2 = 2\log_{10} n$ and $r = 10$. Therefore Radix sort runs in time $O(\log n(10+n) = O(n \log n)$. Unfortunately, this means the approach is no faster than just sorting the elements using Merge sort.

If we instead use the Select algorithm, we can find the median in time $O(n)$.