Algorithms Homework 1

PROBLEM 1: Asymptotic Growth

1, $n^{1/\log(n)}$
log(log(n))
log(n), $\log(n^2)$
$(\sqrt{2})^{\log(n)}$
n
nlog(n), log(n!)
n^2
n^3
$(3/2)^n$
2^n
$n*2^n$
n^n
n!

NOTES:
$n^{1/\log(n)}$ simplifies to e (assuming the base is e) which is a constant like 1. log(n!) simplifies to $\log\sqrt{(2(pi)n}*(n/e)^n$ which becomes $\log\sqrt{(2(pi)n} + n\log(n/e)$ which has O(n(logn)).
$\log(n^2)$ is in the same group as logn because $\log(n^2) = 2\log n$.
$(\sqrt{2})^{\log(n)}$ simplifies to $2^{(½)*\log(n)} = 2^{\log(\sqrt{n})} = \sqrt{n}$ given the property that $\log_a b^x = x$.

PROBLEM 2: The Party Planning Algorithm

1. **Pseudocode**

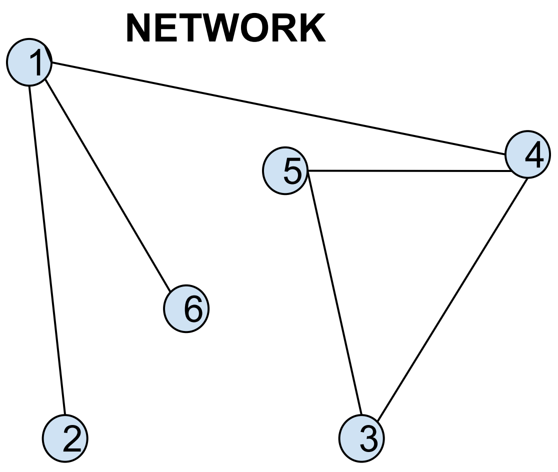
```
C = []
while network is not empty:
    max_friends = 0
    for person in network:
        friends = []
        num_friends = 0
        for friend in network: #finding number of friends
            if person is friends with friend:
                num_friends++
                        friends += friend
                if num_friends > max_friends:
                        p = person
        C = C + p # adding p to the invite list
        network = network - (network - friends) - p # removing p and all of p's non-friends
return C
```

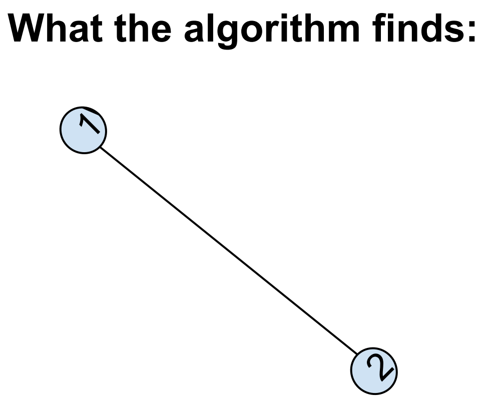
2. **Running Time**

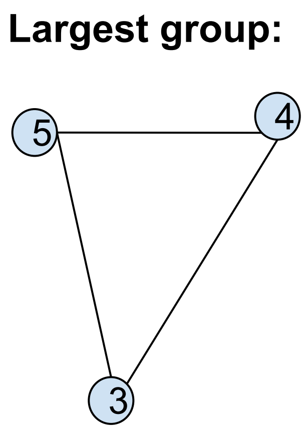
   n^3

   This is the running time is $O(n^2m)$. For the worst case scenario, the outer while-loop has $O(n)$ for each friend in the network. Inside the while loop, there is a double for loop for all people in the network because the first while loop loops for each friendship (m times), and inside of that is a double for loop with a complexity of n^2. Altogether, this is $O(n*n*n) = O(n^3)$ for the worst-case time complexity.

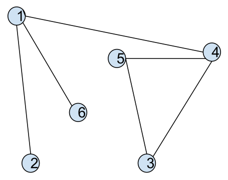
3. **Counterexample**



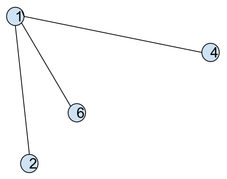




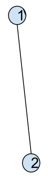
The algorithm steps:



C = []
N = [123456]



C = [1]
N = [246]



C = [12]
N = []

PROBLEM 3: The Hermit Crabs

a) This algorithm sorts n hermit crabs by their current size, and it sorts m empty shells by size in increasing order. Then, it uses two indices with one pointing to the smallest unassigned crab and the other pointing to the smallest empty shell. While there are still crabs to assign and empty shells remaining, it compares the current crab being pointed to (the smallest unassigned crab) and compares it to the current shell being pointed to. If the current shell is larger than the crab, then the crab is assigned, and the algorithm updates to point to the next crab and next shell. If the current shell is not larger, then the algorithm points to the next shell and repeats. This continues until there are either no more empty shells or no more unassigned crabs. If every crab has been assigned, then the crab index being pointed to is equal to the number of crabs. Then, the algorithm will return that it is possible. Otherwise, there are still unassigned crabs left over but no more available shells, so the algorithm returns that it is not possible.

b)

```
# crabs is a list of the current crab sizes and shells is a list of the empty shell sizes.
# pop removes the first item from the list and returns it
crabs = mergesort(crabs) #sorting crabs in increasing order
shells = mergesort(shells) #sorting shells in increasing order

while there are still crabs to assign and available shells:
        if current_shell > current_crab_size:
                Go to next crab
                Go to next shell
        else:
                Go to next shell
if there are no more remaining crab:
        Possible
```

```
else:
    Impossible
```

c) The algorithm is correct because it sorts the crabs and shells. Once sorted, the algorithm starts with the smallest crab. When it scans for larger shells, any shell the same size or smaller than that crab will be too small for any of the other crabs, since they are all larger. In other words, all the shells that were skipped before the the shell was assigned would not work for any of the other crabs either. When the first shell that is larger than the current crab, it is assigned. This makes sure that all of the larger shells are left for the larger crabs. Because this is repeated for each crab, it will make sure that every crab is assigned a shell if it is possible.

d) O(Nlog N)

e) This code runs in O(NlogN) because it first sorts n crab shell sizes and m empty shells using mergesort. Using mergesort to sort the crabs takes O(nlogn) time, and sorting the shells takes O(mlogm) time. These are added together so that the total sorting time is O(nlogn +mlogm), which can become O(NlogN) where N = n + m. Once the crabs and shells are sorted. The algorithm scans through the crab and shell lists, which takes n + m = O(N) time. Thus, the total time is O(NlogN + N), but NlogN dominates over N, which causes the final time complexity to be O(NlogN).

PROBLEM 3: The Hermit Crabs

a) This algorithm sorts the n hermit crabs by their current size and sorts the m empty shells by size in increasing order using mergesort. It then uses two pointers at the indices, starting at the largest crab and the largest available empty shell. When the indices are not 0, there are still crabs and shells left to consider. So when there are still crabs and shells left, it checks whether the current largest shell can fit the current largest crab. If the shell is larger than the crab, the crab is assigned that shell and its old shell becomes an empty shell. So, the algorithm inserts the crab's old shell size back into the sorted shell list at the current position, then moves to the next largest crab. The shell pointer doesn't move but it is now pointing to the newly empty shell. If the current largest shell is not big enough, the algorithm moves to the next smaller shell and tries again. This continues until either all crabs have been placed into larger shells, and it will output possible, or there are no remaining shells large enough to place the remaining crabs and it will output impossible. One thing to note is that I wrote the original code in Python but the last two test cases failed. I rewrote the code in Java, and it did succeed using the same algorithm.

b)

```
# crabs is a list of the current crab sizes, and shells is a list of the empty shell sizes.
# pop removes the first item from the list and returns it
crabs = mergesort(crabs) #sorting crabs in increasing order
shells = mergesort(shells) #sorting shells in increasing order

while crab_index > 0 and shell_index > 0: #while there are still crabs to assign and available
shells:
    if current_shell > current_crab_size:
```

```
                shells.pop(current_shell) #remove the current shell
                shells += current_crab_shell
                crab_index– # got to next crab
        else:
                shell_index– #Go to next shell
    if there are no more remaining crabs:
        Possible
    else:
        Impossible
```

c) The algorithm is correct because it sorts the crabs and shells. Once sorted, the algorithm starts with the largest crab. In each step, it will look at the largest remaining shell and if that shell is not larger than the largest remaining crab, then the ….When it scans for larger shells, any shell the same size or smaller than that crab will be too small for any of the other crabs, since they are all larger. In other words, all the shells that were skipped before the the shell was assigned would not work for any of the other crabs either. When the first shell that is larger than the current crab, it is assigned. This makes sure that all of the larger shells are left for the larger crabs. Because this is repeated for each crab, it will make sure that every crab is assigned a shell if it is possible.

The algorithm is correct because it sorts the crabs and shells by size and then assigns shells in a greedy way. Starting with the smallest crab, it scans the shells until it finds the first one that is strictly larger. Any shell that is the same size or smaller than the current crab cannot fit that crab, and it also cannot fit any later crab because all remaining crabs are at least as large. So skipping those shells does not eliminate any valid solution. Assigning the first shell that works also preserves larger shells for larger crabs, preventing "wasting" a big shell on a small crab. Repeating this step for each crab ensures that if a complete assignment is possible, the algorithm will find one; otherwise, it correctly concludes that it is not possible.

f) O(Nlog N)

g) This code runs in O(NlogN) because it first sorts n crab shell sizes and m empty shells using mergesort. Using mergesort to sort the crabs takes O(nlogn) time, and sorting the shells takes O(mlogm) time. These are added together so that the total sorting time is O(nlogn +mlogm), which can become O(NlogN) where N = n + m. Once the crabs and shells are sorted. The algorithm scans through the crab and shell lists, which takes n + m = O(N) time. Thus, the total time is O(NlogN + N), but NlogN dominates over N, which causes the final time complexity to be O(NlogN).

PROBLEM 4: Drone Delivery Combinations

Verbal algo description, pseudocode, sketch or proof or correctness, tight running time estimate, reasoning behind running time estimate.

n^2logn

a) The algorithm iterates through the list A (outbound distances). For each value in A, it iterates through every return distance in list B. For each A value and B value pair (a,b), it finds the total mission distance by a+b = distance. It then adds this distance to a list of distances. Once all the sums have been calculated, the list of these sums is sorted using mergesort. Then, there is a for loop that loops through each distance and counts the first occurrence of a new number as a unique distance. Once it loops through the whole list, it will have the total number of unique distances.

b)

```
# get list of all the distances
distances = []
for a in A:
        for b in B:
                sum = a + b
                distances += sum

# sort the list of distances which is n^2 size → O(n^2log(n^2)
sorted_distances = mergesort(distances)

# loop through the list and count unique
unique_distances = 1
for i in (1, n^2):
        if sorted_distances[i] != sorted_distances [i-1]:
                unique_distances++
return unique_distances
```

c) This algorithm is correct because it makes sure to check every possible outbound and return combination pair. This is because it loops through A and for every a in A, it checks it with every b in B. This makes sure that every possible mission distance is actually in the distances list. The sorting of this list makes duplicates appear next to each other. Thus, whe the final loops scans through the sorted list, it will only increment the counter when the current value is different from the previous value. This will occur only once for each unique distance. Thus, the algorithm will count each unique mission distance.

d) $O(n^2\log(n))$

e) The algorithm runs in $O(n^2\log n)$ time because it first uses two nested loops to iterate through lists A and B, which takes $O(n^2)$ time. During this process, it finds all possible sums and adds them to a list. So, the list of these distances contains n^2 elements. The algorithm then sorts this list using mergesort, which takes $O(n^2\log(n^2))$ time which simplifies to $O(n^2*2*\log(n))= O(n^2\log(n))$. Finally, the algorithm scans through the sorted list once to count distinct distances, which takes another $O(n^2)$ time. Combining all steps gives a total time complexity of $O(n^2+n^2\log n+n^2)=O(n^2\log n)$.

PROBLEM 5: Empirical Sorting Analysis

This figure illustrates how time complexity varies with input size, input distribution, and sorting algorithm. Specifically, the sorting algorithms mergesort, bucketsort, and insertion sort were used; they each have known time complexities of $O(n \log n)$, $O(n)$, and $O(n^2)$, respectively. N values of 100, 1000, 10,000, and 100,000 were used, and both a uniform and a Gaussian distribution were generated. In the figure, blue represents mergesort, red represents bucketsort, and green represents insertion sort, while dashed lines indicate the Gaussian distribution and solid lines indicate the uniform distribution. It is important to note that the axes in this plot are on a logarithmic scale. Additionally, the program timed out after two minutes. For example, this means that insertion sort would take longer than two minutes for n = 100,000. In this plot, for both Gaussian and uniformly distributed data, insertion sort and mergesort each perform similarly within themselves, meaning distribution probably does not have a large impact on their performance. However, it is notable that mergesort performs much better than insertion sort. As n increases, the degree to which mergesort outperforms insertion sort also increases. This makes sense because insertion sort runs in O(n^2) time, which grows rapidly, while mergesort runs in O(n logn), which grows more slowly. It is not as fast as linear time, but it is better than quadratic time. Bucket sort is different because its speed depends on how the data is distributed. When elements are uniformly distributed across buckets, it can run in near-linear time. But when elements fall more heavily into one bucket, like with a Gaussian distribution, it runs closer to O(n^2) time and becomes much less efficient. When $\sigma = .01$ (Figure 1A), bucketsort appears to perform well for both Gaussian and uniform distributions, with only a minor gain with uniform data. However, when sigma is decreased to $\sigma = .00001$, it is less evenly distributed, and thus, bucket sort performs more similarly to insertion sort for Gaussian data (Figure 1B). Therefore, bucket sort performs better than mergesort with uniform data but much worse with Gaussian data. This demonstrates that not only can the size of the input data (n) impact time complexity, but also the distribution of the data itself. In general, insertion sort is always the worst.

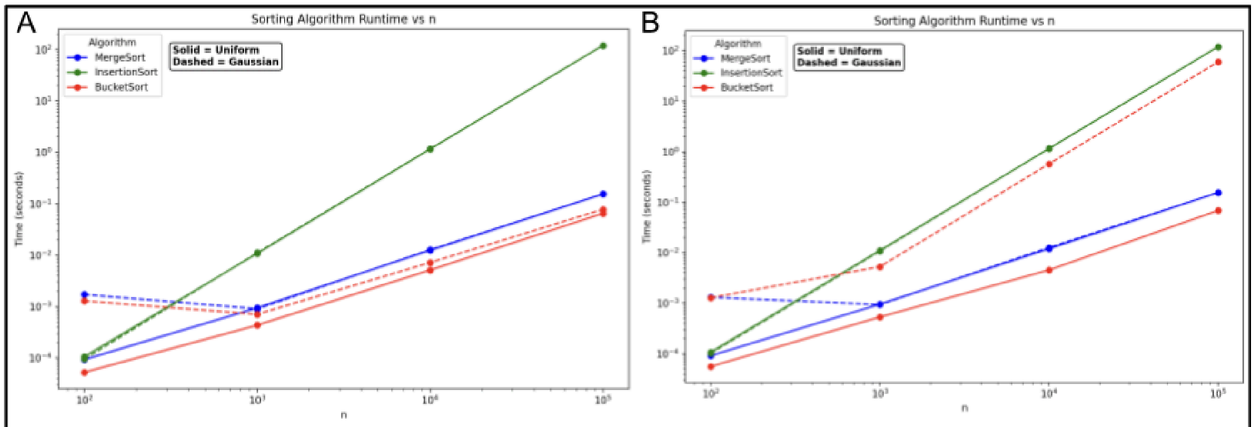


Figure 1. Plot representing input size versus time for an algorithm to complete using MergeSort, InsertionSort, and BucketSort using gaussian and uniformly distributed data. A) $\sigma = .01$, B) $\sigma = .00001$

Code Source: The code was modified from the pseudocode given in class into Python.