Problem 1

- (1) D
- (2) 2
- (3) $k = 5$
- (4) $f(n) = O(g(n))$
- (5) $S = 1536$
- (6) Number of swap is 8
- (7) $[1, 2, 7, 8, 9, 6, 3, 4, 5]$

(8) Height of the node is the longest path or number of edges from the node to the children. Denote the height as h(N) where N is the node.

$$h(N) = \begin{cases} 0 \text{ if N is a leaf node} \\ 1 + \max(h(\text{N's left child node}), h(\text{N's right child node})) \end{cases}$$

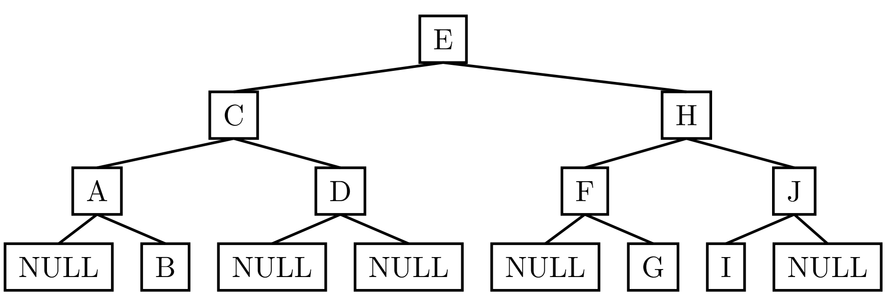
Depth of the node is the longest path or number of edges from the root to the node. Denote the depth as d(N) where N is the node.

$$d(N) = \begin{cases} 0 \text{ if N is the root node} \\ 1 + d(\text{N's parent node}) \end{cases}$$

(9)

- Preorder traversal:[a b d e c f h l m q i g j n o k p]
- Inorder traversal: [d b e a l h q m f i c n j o g k p]
- Postorder traversal:[d e b l q m h i f n o j p k g c a]

(10)



Problem 2

(a)

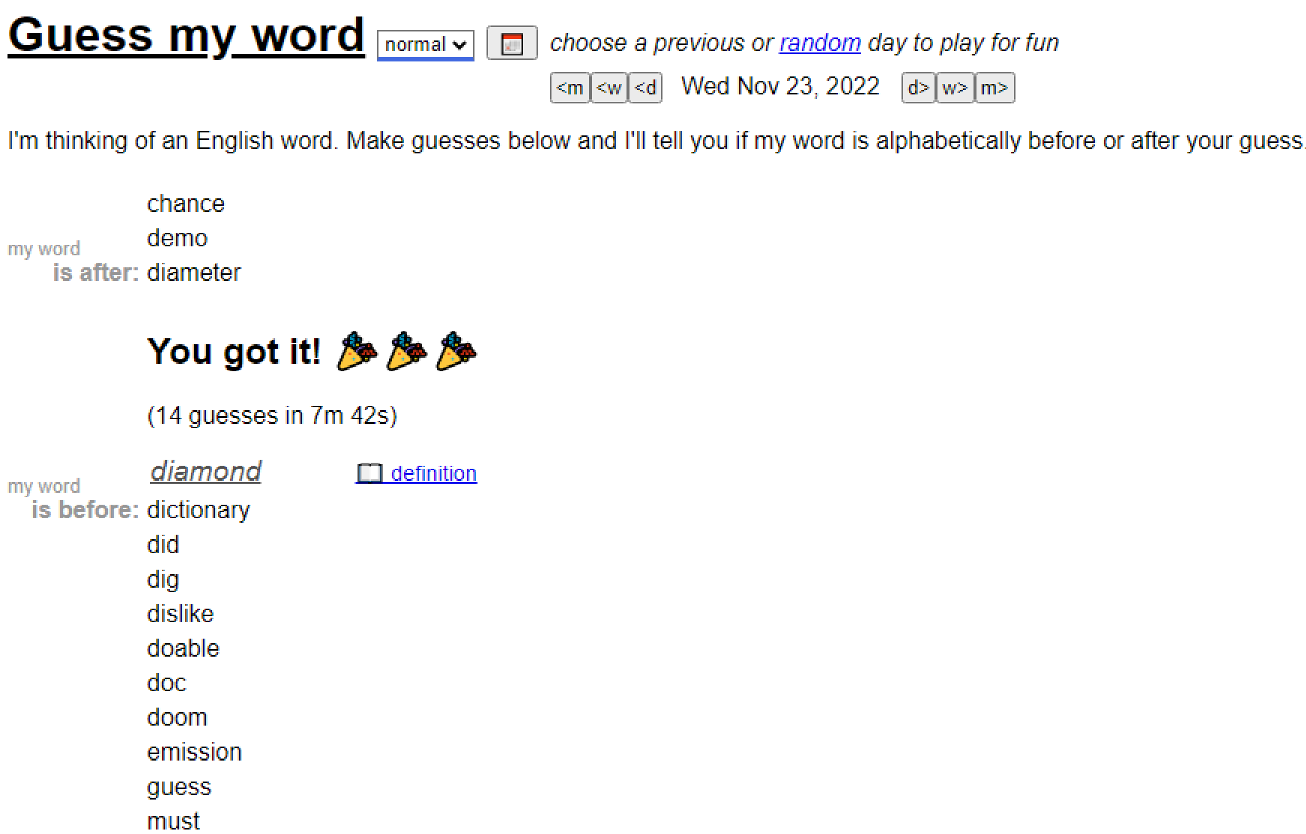


Figure 1: Solution (2-a)

(b)

Initilaize an offset to 1, denoting the depth of letter for comparison. For example, we the depth is 1 for castle and cattle, then we're comparing the 2nd letter of the two words.
Approximate the offset letter of the middle word in the dictionary, alphabetically, it is m-started word, the the ascii code is $\frac{\text{int('z')}-\text{int('a')}}{2}$ and compare it with the target word. If response says "before, then the target word must be in the first half of the dictionary. Otherwise, the target word must be in the second half of the dictionary.
Recursively repeat: Shrink the find range by half based on the previous step, approximate the offset letter of the middle word in the new range, and compare it with the target word. If the target word is less than the middle word, then the target word must be in the first half of the new range. Otherwise, the target word must be in the second half of the new range. If the range is reduced to 0 but the word is not found, then increase the offset by 1 and repeat the process.
This is an simplified version of search, actual search should consider the distribution of the words in the dictionary. For example, m may not be the middle word in the dictionary. and there's much less word contains letters like xyz than abc.

(c)

$T(n) = T\left(\frac{n}{2}\right) + O(1)$
Every time, the middle word is being compared and cost O(1) time. and the range is reduced by half every time by picking the middle value.
Master theorem has the following form: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$
where $a = 1, b = 2, f(n) = O(1)$

- Consider case 2: $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$ for some $k \geq 0$, then $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$

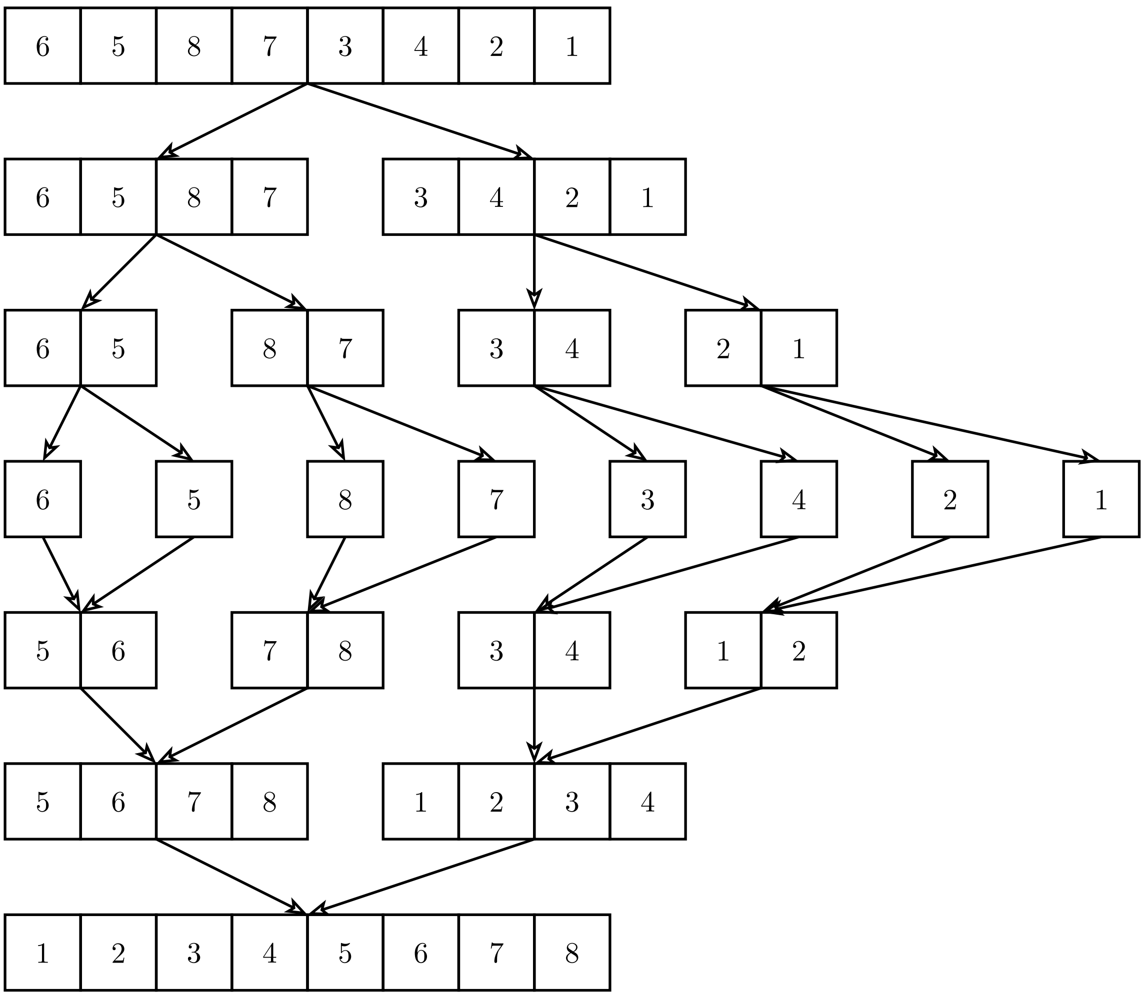
here, $\log_b a = 0$ and we let $k = 0$ we have $O(1) = \Theta(n^0) = \Theta(1)$ therefore $T(n) = \Theta(\log(n))$

(d)

$\log(267751) \approx 18$
Assume you cannot quit the game half way The maximum number of comparisons is 18, which means it costs at maximum 18 dollars to find the word. So to profit 1 bucks, we can only do 14 comparison, which minimize the range to $2^4 = 16$ and and the expectation is $E(\text{profit}) = \frac{1}{16} * 1 + \frac{15}{16} * (-4) = -2.75$ if we do 13 comparison the expectation is even lower.

Problem 2

(a)



- 4 Comparison: 6 and 5, 8 and 7, 3 and 4, 2 and 1
- 4 comparison: 5 and 7, 7 and 6, 3 and 1, 3 and 2,
- 4 comparison: 5 and 1, 5 and 2, 5 and 3, 5 and 4

Total 12 comparison

(b)

$M = 2M\left(\frac{n}{2}\right) + \frac{n}{2}$
So merge sort is a divide and conquer algorithm, it divides the array into two halves and recursively compare and merge the two halves. The division step therefore, results in 2 subproblems of size n/2, which gives $2M\left(\frac{n}{2}\right)$ to merge from.
The merge step is the conquer step, it compare and merges the two sorted halves into a single sorted array. And the comparison takes up from $n-1$ (alternating subarray) to $\frac{n}{2}$ (all elements from one is smaller than the other). Since we only consider the minimum, then the number of comparison is $\frac{n}{2}$
Therefore, we have $M(n) = 2M\left(\frac{n}{2}\right) + \frac{n}{2}$

(c)

We use the substitution method to solve the recurrence.

$$\begin{aligned}
M(n) &= 2\frac{\frac{n}{2}\log\left(\frac{n}{2}\right)}{2} + \frac{n}{2} \\
M(n) &= \frac{n}{2}\log\frac{n}{2} + \frac{n}{2} \\
M(n) &= \frac{n}{2}(\log n - \log 2) + \frac{n}{2} \\
M(n) &= \frac{n}{2}\log n - \frac{n}{2} + \frac{n}{2} \\
M(n) &= \frac{n}{2}\log n
\end{aligned}$$

Therefore, by substitution method, we have $M(n) = \frac{n\log n}{2}$

Problem 3

(a)

In seleciton sort, we find the minimum element in the unsorted array and swap it with the first element. Then we find the minimum element in the remaining unsorted array and swap it with the second element. We repeat this process until the array is sorted. If the input is sorted, the algorithm still needs to compare every element with the minimum element in the unsorted array. Therefore, The number of comparison is $n + (n-1) + (n-2) + ... + 1 = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$
By definition of big O, we need to find c and n_0 such that $\frac{n^2}{2} + \frac{n}{2} \leq cn$ for all $n \geq n_0$
Note that $\frac{n^2}{2} + \frac{n}{2} \leq \frac{n^2}{2} + \frac{n^2}{2} \leq n^2$
Let $c = 1$

$$\frac{n^2}{2} + \frac{n}{2} \leq n^2$$
$$\frac{n}{2} \leq \frac{n^2}{2}$$
$$1 \leq n$$

so $n_0 = 1$
Therefore, $\frac{n^2}{2} + \frac{n}{2} = O(n)$
For Insertion sort, we start from the second element and compare it with the first element, if it is smaller, we swap them. Then we compare the third element with the second element and swap if necessary. We repeat this process until the array is sorted. However, for this input, the inner loop will not be executed since the 2nd element is never smaller than the first element. Therefore, the number of comparison is $n - 1$
By definition of big O, we need to find c and n_0 such that $n - 1 \leq cn$ for all $n \geq n_0$ note that $n - 1 \leq n$
Therefore, $n - 1 = O(n)$
So O(n) is smaller than O(n^2), therfore the insertion sort is better

(b)

In heap sort, we first build a max heap from the array, then we swap the first element with the last element and heapify the array. We repeat this process until the array is sorted.
The heap building process can be expressed

$$
\begin{aligned}
\sum_{i=\lfloor \log n \rfloor}^{0} 2^i(\lfloor \log n \rfloor - i) &\leq \sum_{i=\lfloor \log n \rfloor}^{0} \frac{n}{2^h} h \\
&\leq n \sum_{i=\lfloor \log n \rfloor}^{0} \frac{h}{2^h} \\
&\leq n \sum_{i=0}^{\infty} \frac{h}{2^h} \\
&= 2n
\end{aligned}
$$

By letting $c = 2$ and $n_0 = 1$, we have $2n = O(n)$ Therefore, the heap building process is $O(n)$
then we have the extraction process, which consist of n extraction and heapify process. The heapify process is $O(\log n)$, so the extraction process is $O(n \log n)$
Therefore, the total time complexity of heap sort is $O(n) + O(n \log n) = O(n \log n)$
Qucik sort consist of partition and sort process. The partition process will select a pivot and partition the array into two halves, then recursively partition the two halves. In this case, the partition will always select the last element as the pivot, and the partition process will always result in a partition of 1 and n-1. Therefore, the partition process makes k-1 comparison. where k is the kth partition. The total number of comparison is thus $n + (n-1) + (n-2) + ... + 1$ and we have proved in the previous question that this is $O(n^2)$
Therefore, the total time complexity of quick sort is $O(n^2)$ and heap sort is better than quick sort

(c)

The bubble sort does not depends on the input, it will always compare every element with the next element and swap if necessary. Therefore, the number of comparison is $\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$
By definition of big O, we need to find c and n_0 such that $\frac{n^2}{2} - \frac{n}{2} \leq cn$ for all $n \geq n_0$
since $\frac{n^2}{2} - \frac{n}{2} < \frac{n^2}{2} + \frac{n}{2} \leq c^n$ which we have proved in the previous question, we can let $c = 1$ and $n_0 = 1$
Therefore, $\frac{n^2}{2} - \frac{n}{2} = O(n^2)$
In bucket sort, we first divide the array into buckets, then sort each bucket and concatenate them. Since n is an integer, We initialize n empty buckets, this takes O(n) time. Then we insert element to its bucket and sort each bucket using insertion sort, which takes O(n) time since each bucket have the same element. Finally, we concatenate the buckets, which takes O(n) time. Therefore, the total time complexity is $O(n)$
Therefore, the bucket sort is better than bubble sort for this input

(d)

In Counting sort with the given input, we Initilaize the number of occurance from 0-9 to zero. This takes 10 operations. Then we iterating through the array and count the occurance of each element, which takes n operations. Then we iterate through the occurance array and sum the previous element, which takes n operations Finally, we iterate through the array and place the element in the correct position, which takes n+10 operations. Therefore, the total time complexity is $3n + 20 = O(n)$
let $c = 4$

$$3n + 20 \le 4n$$
$$n \ge 20$$

Therefore, $n_0 = 20$
In merge sort, as we shown in the previous question, the best time complexity is $O(n \log n)$
The worst time complexity of counting sort is still $O(n \log n)$ since we can express the number of comparison as $M(n) = 2M\left(\frac{n}{2}\right) + n - 1$
Using Master theorem, we have $a = 2, b = 2, f(n) = n - 1$

- Consider case 2: $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$ for some $k \ge 0$, then $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$

here, $\log_b a = 1$ and we let $k = 0$ we have
$O(n - 1) = \Theta\left(n^1 \log^0 n\right) = \Theta(n)$ therefore $T(n) = \Theta(n \log n)$
Therefore, the counting sort is better than merge sort for this input

Problem 4

BFS traverse the graph level by level, starting from the root node, then each immediate neighbor before moving the to the neighbors's neighbor. It utilizes a queue (in which elements lastly inserted is popped out last(LILO)), by enqueuing the starting node and dequeuing the node and enqueuing its neighbors, then repeat. The time complexity of BFS is $O(V+E)$ where V is the number of vertices and E is the number of edges. It also finds the shortest path in an unweighted graph.
DFS traverse the graph by going as deep as possible along each branch before backtracking. It utilizes a stack (in which element lastly inserted is poped first (LIFO)), by pushing the starting node and popping the node and pushing its neighbors, then repeat. The time complexity of DFS is $O(V+E)$ where V is the number of vertices and E is the number of edges. It is not guaranteed to find the shortest path in an unweighted graph, but it can be used to find the connected components of a graph.
For the graph above,