Final Exam

CIS-675 DESIGN AND ANALYSIS OF ALGORITHMS

PROF. IMANI PALMER

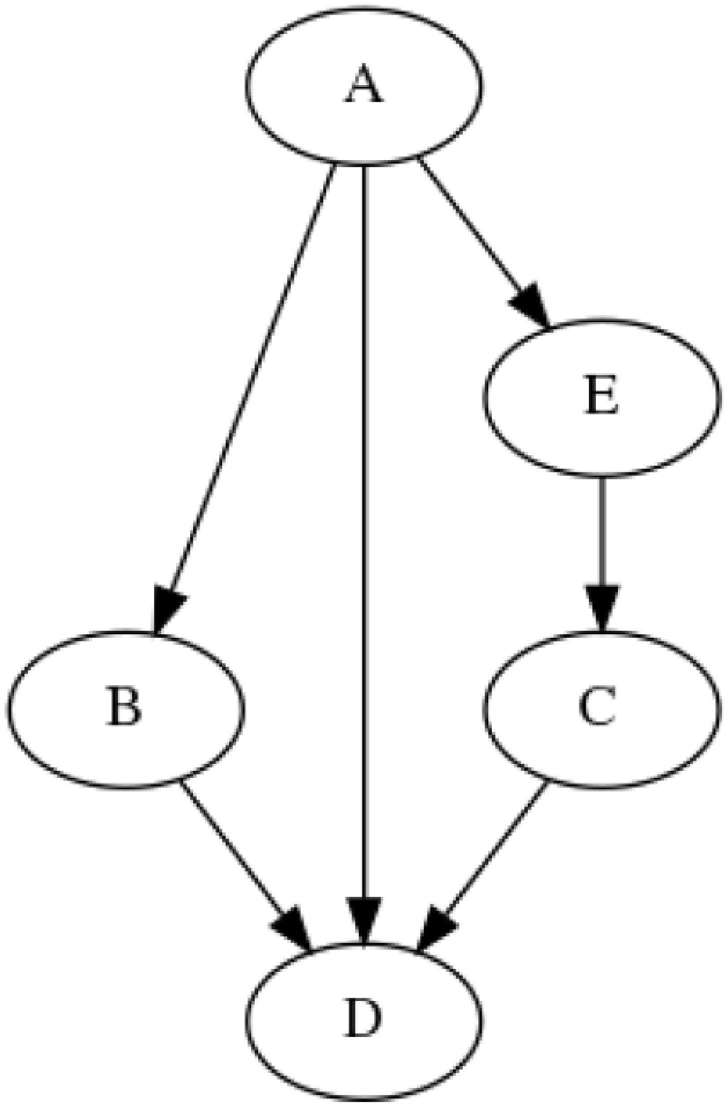
6/18/2021

Anthony Redamonti
SYRACUSE UNIVERSITY

Questions 1, 2, 3, 4, 5, 7, 8, and 10 have been answered.

Question 1:

Use the following graph for this problem. Draw either the adjacency matrix or the adjacency list representations of this graph.



The adjacency matrix for the graph is below.

	A	B	C	D	E
A	0	1	0	1	1
B	0	0	0	1	0
C	0	0	0	1	0
D	0	0	0	0	0
E	0	0	1	0	0

Question 2:

Match each algorithm below with its worst-case runtime by filling out the table. Do not justify you answers.

Algorithm	Running Time
Insertion Sort	O(n^2)
Heap Sort	O(n*log(n))
Bellman-Ford	O(|V|*|E|)
Depth-First Search	O(V+E)
Prim's	O(|E|*log(|E|))

Question 3:

Describe the worst case running time of the following psuedocode functions in Big-Oh notation interms of the variable n. Showing your work is not required (although showing work may allow for partial credit in the case your answer is wrong.)

```
1.
int silly(int n, int m) {
  if (n < 1) return m;
  else if (n < 10)
       return silly(n/2, m);
  else
       return silly(n-2, m);
```

The worst-case running time of *silly(int n, int m)* is O(n).

Explanation: For any n value greater than 10, the function will run n/2 times. Think of n equal to some very large number (1,000,000,000). The runtime would be roughly n/2, which simplifies to O(n).

```
2.
void warm(int n) {
  for (int i = 0; i < 2 * n; ++i) {
    j = 0;
    while (j < n) {
      print(j);
      j = j + 5;
    }
  }
}
```

The worst-case running time of *warm(int n)* is O(n^2).

Explanation: The outer loop will run 2n times, which simplifies to O(n). The inner loop will run n/5 times, which simplifies to O(n). Because one loop is nested inside the other, they are multiplied to produce the O(n^2) worst-case runtime.

Question 4:

Choose True or False for the following statement to indicate whether the statement is true or false, respectively. If the statement is correct, briefly state why. If the statement is wrong, explain why. The more content your provide in your justification, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.
True or False, Let A_1, A_2 and A_3 be three sorted arrays of n real numbers (all distinct). In the comparison model, constructing a balanced binary search stree of the set $A_1 \cup A_2 \cup A_3$ requires $\Omega(nlogn)$ time.

False.

We can merge all three sorted arrays into one sorted array in O(n) time. Then creating the balanced binary search tree would also take O(n) time. Thus O(n) + O(n) = 2*O(n), which reduces to O(n). Therefore, the statement that the process would take Ω(nlog(n)) is false.

A_1 = {2, 5, 9, 10, 11, 12, 15}

A_2 = {34, 56, 200, 204}

A_3 = {20, 21, 22, 23, 400}

Sorted_Union_Array = {}

The first elements of each array are compared. The first element of lowest value is copied to the Sorted_Union_Array and ignored in future iterations.

The new sorted array is below.

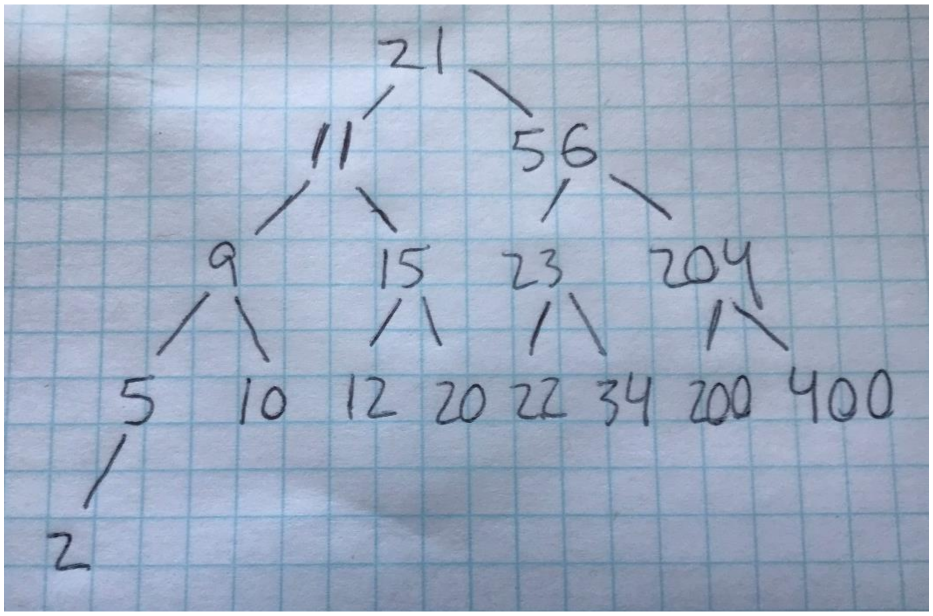
Sorted_Union_Array = {2, 5, 9, 10, 11, 12, 15, 20, 21, 22, 23, 34, 56, 200, 204, 400}

Balanced_Binary_Search_Tree_Array = {}

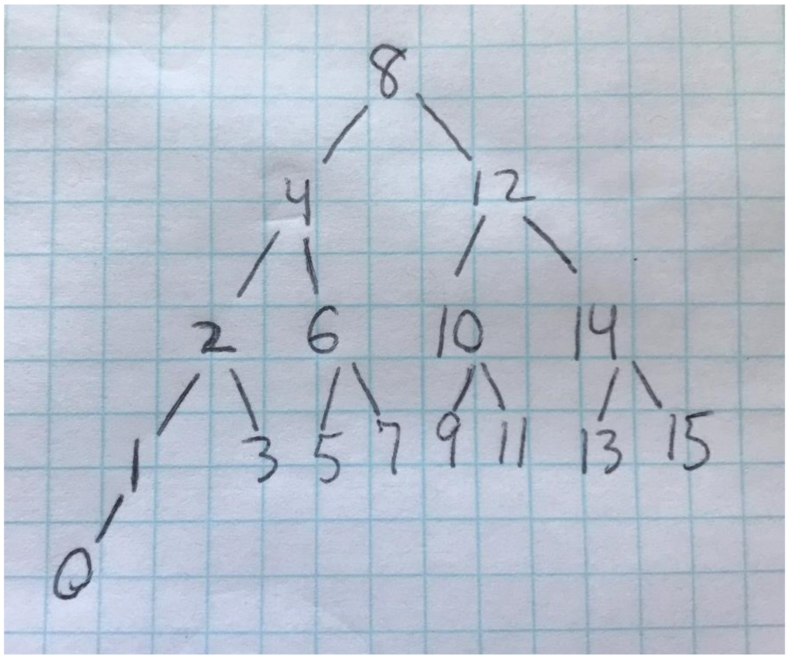
The elements from the sorted array are copied to the balanced binary search tree array. The element located in the middle of the union array is copied to the first element of the binary search tree. The array is then split into 2 halves and the process is continued until all elements have been copied. The runtime is O(n).

Therefore, the runtime of the entire process is O(n).

The representation of the Balanced Binary Search Tree is below.



Because the array is already sorted, we can use the size of the array to generate a balanced binary search tree of the indices (shown below).



The indices are then used to generate the balanced binary search tree using a simple for-loop.

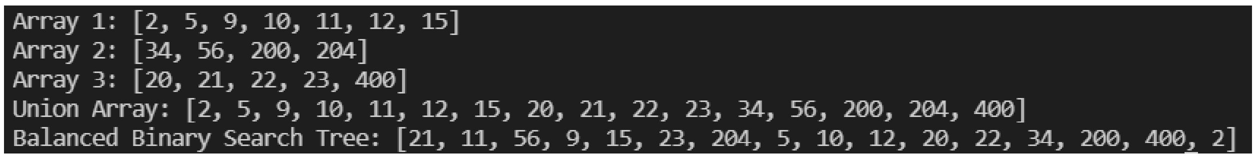
```
for(i = 0; i < count; i++){
   tree_array[i] = sorted_array[index_array[i]];
}
```

The following implementation is written in C below.

```c
#include <stdio.h>
#define N 7 // length of array1
#define M 4 // length of array2
#define P 5 // length of array3

// function that returns base^exponent
int power_func(int base, int exponent)
{
   int i;
   int answer = base;
   for(i = 1; i < exponent; i++)
   {
      answer *= base;
   }
   return(answer);
}

void return_min(int* arr1, int* arr2, int* arr3, int* index1, int* index2, int* i
ndex3, int* bigarr, int* count){
   // array1 and array2 are empty. return array3.
   if((*index1 == -1)&&(*index2 == -1)&&(*index3 != -1)){
      bigarr[*count] = arr3[*index3];
      *index3 = *index3 + 1;;
      *count = *count + 1;
      return;
   }
   // array2 and array3 are empty. return array1.
   if((*index2 == -1)&&(*index3 == -1)&&(*index1 != -1)){
      bigarr[*count] = arr1[*index1];
      *index1 = *index1 + 1;
      *count = *count + 1;
      return;
   }
   // array1 and array3 are empty. return array2.
   if((*index3 == -1)&&(*index1 == -1)&&(*index2 != -1)){
      bigarr[*count] = arr2[*index2];
      *index2 = *index2 + 1;
      *count = *count + 1;
      return;
   }

   // array1 is empty but array2 and array3 are not.
   if((*index1 == -1)&&(*index2 != -1)&&(*index3 != -1)){
      if(arr2[*index2] <= arr3[*index3]){
         bigarr[*count] = arr2[*index2];
         *index2 = *index2 + 1;
         *count = *count + 1;
         return;
      }
      else{
         bigarr[*count] = arr3[*index3];
         *index3 = *index3 + 1;
         *count = *count + 1;
         return;
      }
   }

   // array2 is empty but array1 and array3 are not.
   if((*index2 == -1)&&(*index1 != -1)&&(*index3 != -1)){
      if(arr1[*index1] <= arr3[*index3]){
         bigarr[*count] = arr1[*index1];
         *index1 = *index1 + 1;
         *count = *count + 1;
         return;
      }
      else{
         bigarr[*count] = arr3[*index3];
         *index3 = *index3 + 1;
         *count = *count + 1;
         return;
      }
   }

   // array3 is empty but array2 and array1 are not.
   if((*index3 == -1)&&(*index2 != -1)&&(*index1 != -1)){
      if(arr2[*index2] <= arr1[*index1]){
         bigarr[*count] = arr2[*index2];
         *index2 = *index2 + 1;
         *count = *count + 1;
         return;
      }
      else{
         bigarr[*count] = arr1[*index1];
         *index1 = *index1 + 1;
         *count = *count + 1;
         return;
      }
   }
   // none of the arrays are empty.
   if((arr1[*index1] <= arr2[*index2])&&(arr1[*index1] <= arr3[*index3])){
      bigarr[*count] = arr1[*index1];
      *index1 = *index1 + 1;
      *count = *count + 1;
      return;
   }
   if((arr2[*index2] <= arr1[*index1])&&(arr2[*index2] <= arr3[*index3])){
      bigarr[*count] = arr2[*index2];
      *index2 = *index2 + 1;
      *count = *count + 1;
      return;
   }
   if((arr3[*index3] <= arr1[*index1])&&(arr3[*index3] <= arr2[*index2])){
      bigarr[*count] = arr3[*index3];
      *index3 = *index3 + 1;
      *count = *count + 1;
   }
}

void combine_arrays(int* array1, int* array2, int* array3, int* big_array){
   int count = 0;
   int size_of_big_array = N+M+P;
   int index1 = 0;   int index2 = 0;   int index3 = 0;
   int* ptr_index1 = &index1; int* ptr_index2 = &index2;
   int* ptr_index3 = &index3; int* ptr_count = &count;
   int min;

   while(count != size_of_big_array){

      return_min(array1, array2, array3, ptr_index1, ptr_index2, ptr_index3, big_
array, ptr_count);
      if(index1 == N){index1 = -1;}
      if(index2 == M){index2 = -1;}
      if(index3 == P){index3 = -1;}
   }
}

// create array of indices
int balance_the_tree(int* tree_array, int* sorted_array, int size, int count, int
 tree_index, int iteration){
   int i, prev_index, value;
   if(count == size){return 0;}
   int number_of_elements_to_fill = power_func(2, iteration);
   int special_value = ((tree_array[0])/number_of_elements_to_fill);
   if(special_value == 0){return count;}
   int index1, index2;
   for(i = 0; i < number_of_elements_to_fill/2; i++){
      if(tree_index%2 == 0){
         prev_index = (tree_index/2)-1;
      }
      else{
         prev_index = (tree_index/2);
      }

      value = tree_array[prev_index]; // 8

      index1 = (value - special_value);
      index2 = (value + special_value);
      tree_array[tree_index] = index1;
      count++; tree_index++;
      if(count == size){return 0;}
      tree_array[tree_index] = index2;
      count++; tree_index++;
   }
   iteration++;
   balance_the_tree(tree_array, sorted_array, size, count, tree_index, iteration)
;
}

void create_balanced_binary_tree(int* sorted_array, int* tree_array, int size){
   int i;
   int count = 1; int tree_index = 1; int iteration = 1;
   int* index_array = (int*)(malloc(sizeof(int)*size));
   index_array[0] = size/2; // fill root node.
   count = balance_the_tree(index_array, sorted_array, size, count, tree_index, i
teration);
   int temp = 0;
   for(i = 0; i < count; i++){
      tree_array[i] = sorted_array[index_array[i]];
   }
   while(count != size){
      tree_array[i] = sorted_array[temp];
      i++; temp++; count++;
   }
   free(index_array);
}

void print_array(int* array, int size){
```

```c
    for(int i = 0; i < size-1; i++){
        printf("%d, ", array[i]);
    }
    printf("%d]\n", array[size-1]);
}

int main(){
    int i;
    int array1[N] = {2,5,9,10,11,12,15};
    int array2[M] = {34, 56, 200, 204};
    int array3[P] = {20, 21, 22, 23, 400};

    printf("Array 1: [");
    print_array(array1, N);

    printf("Array 2: [");
    print_array(array2, M);

    printf("Array 3: [");
    print_array(array3, P);

    int* union_array = (int*)(malloc(sizeof(int)*(N+M+P)));
    combine_arrays(array1, array2, array3, union_array);
    printf("Union Array: [");
    print_array(union_array, N+M+P);

    int* balanced_binary_tree = (int*)(malloc(sizeof(int)*(N+M+P)));
    create_balanced_binary_tree(union_array, balanced_binary_tree, N+M+P);
    printf("Balanced Binary Search Tree: [");
    print_array(balanced_binary_tree, N+M+P);

    free(union_array);
    free(balanced_binary_tree);
}
```

The output of the program is below.

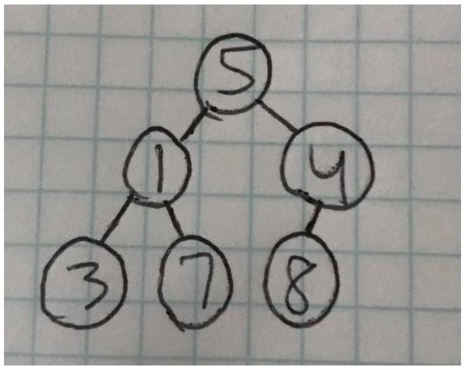


```
Array 1: [2, 5, 9, 10, 11, 12, 15]
Array 2: [34, 56, 200, 204]
Array 3: [20, 21, 22, 23, 400]
Union Array: [2, 5, 9, 10, 11, 12, 15, 20, 21, 22, 23, 34, 56, 200, 204, 400]
Balanced Binary Search Tree: [21, 11, 56, 9, 15, 23, 204, 5, 10, 12, 20, 22, 34, 200, 400, 2]
```

Question 5:

Choose True or False for the following statement to indicate whether the statement is true or false, respectively. If the statement is correct, briefly state why. If the statement is wrong, explain why. The more content your provide in your justification, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.
Let T be a complete binary tree with n nodes. Finding a path from the root of T to a given vertex $v \in T$ using breath-first search takes $O(logn)$ time.

False. Using breadth-first search to find a path from the root to a given vertex $v \in T$ in a complete binary tree with n nodes takes O(n) time. View an example complete binary tree below.



Starting at the root node (key value = 5), search for the node with the key value = 8. The breadth-first search would travel along every edge searching for the node with the correct key value. The runtime of Breadth-First Search is O(V+E) where V is the number of vertices and E is the number of edges in the graph or tree.

Below is the procedure for Breadth-First Search discussed in class. Each edge is traversed, and all nodes are found along each edge.

```
Procedure BFS(T, s)
Input: Complete Binary Tree T = (V,E), directed or undirected; vertex $s \in V$
Output: For all vertices u reachable from s, dist(u) is set to the distance from s to u.

For all $u \in V$:
    Dist(u) = ∞

Dist(s) = 0
Q = [s] // queue containing just s
While Q is not empty:
    u = eject(Q)
    for all edges$(u, v) \in E$:
```

```
if(Dist(v) == ∞):
    inject(Q, v)
    Dist(v) = Dist(u) + 1
```

Question 7:

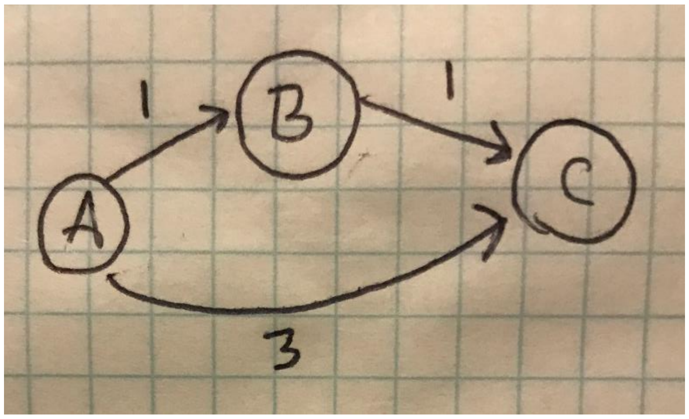
Choose True or False for the following statement to indicate whether the statement is true or false, respectively. If the statement is correct, briefly state why. If the statement is wrong, explain why. The more content your provide in your justification, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.
Let $G = (V, E)$ be a directed graph with negative-weight edges, but no negative-weight cycles. Then, one can compute all shortest paths from a source $s \in V$ to all $v \in V$ faster than Bellman-Ford using the technique of reweighting.

True.

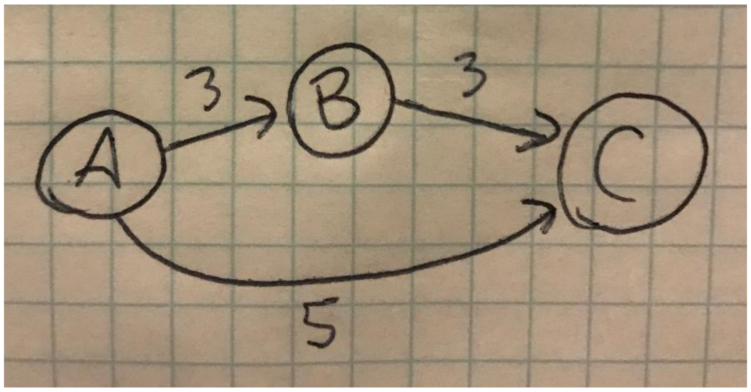
As discussed in class, Dijkstra's Algorithm has a faster runtime than Bellman-Ford. The Bellman-Ford algorithm has a runtime of O(E*V), and Dijkstra's algorithm has a runtime of O((|E| + |V|)log(V)) when a binary min-heap is used as a priority queue. However, Dijkstra's Algorithm does not work with some graphs containing negative-weight edges. Therefore, to guarantee that Dijkstra's Algorithm will successfully find the shortest paths from a source $s \in V$ to all $v \in V$, apply the technique of reweighting to remove all negative-weight edges.

We must be careful when reweighting to preserve the shortest path. Adding a fixed amount to all edges in the graph is the incorrect way to reweigh a graph.

For example, consider the graph below. The shortest path is ABC with a value of 2.



Now, increase all edge weights by a value of 2.



The shortest path is now AC with a weight of 5. The shortest path was not preserved!

Thus, we cannot increase all edges by a constant value, since shortest paths may have a different number of edges.

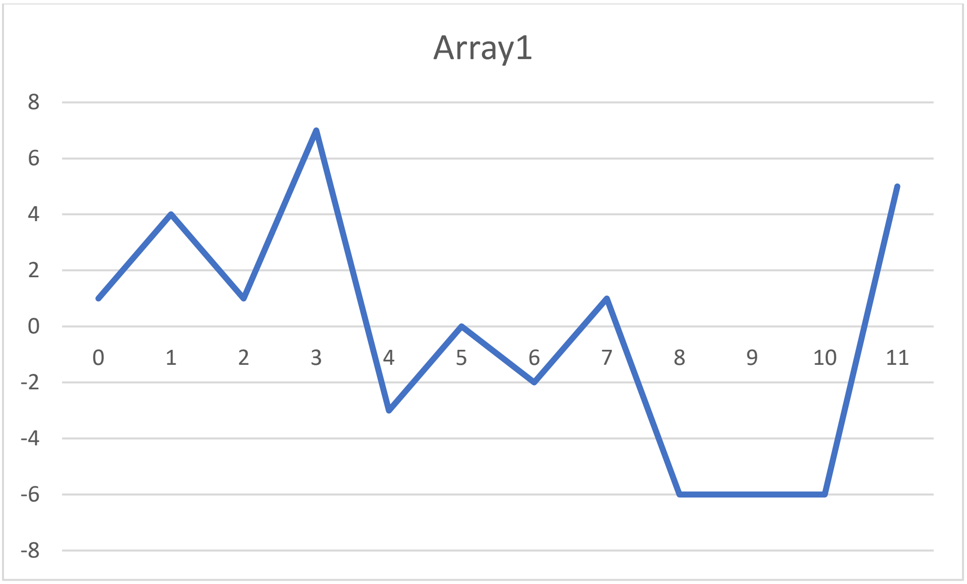
Instead, use Johnson's Algorithm to remove negative-weight edges while preserving the shortest paths. Johnson's Algorithm uses the Bellman-Ford algorithm to correctly reweigh a graph so that Dijkstra's Algorithm can be used on it later.

Question 8:

An array $A[1...2n+1]$ is wiggly if $A[1] \leq A[2] \geq A[3] \leq A[4] \geq ... \leq A[2n] \geq A[2n+1]$. Given a an unsorted array $B[1..2n+1]$ of real numbers, describe an efficient algorithm that outputs a permutation $A[1..2n+1]$ of B such that A is a wiggly array.

Below is an example “wiggly” array along with its plots of index vs value.

Array1 = [1, 4, 1, 7, -3, 0, -2, 1, -6, -6, -6, -5]



From the graph, it is illustrated that for even-numbered indices, their corresponding values must be less than or equal to the value of its predecessor. A similar statement can be made of the odd-numbered indices, the values of which must be greater than or equal to their predecessors.

From this conclusion, the following rules can be derived:

- If index is even, array[index] must be ≤ array[index-1]
- If index is odd, array[index] must be ≥ array[index-1]

From these rules follows a linear-time algorithm:

```
void Make_Wiggly(int* array){
   int i;
   for(i = 1; i < N; i++){
      if(((i%2==0)&&(array[i] > array[i-
1])) // if even, make sure array[i] <= array[i-1]
      ||((i%2==1)&&(array[i] < array[i-
1]))){ // if odd, make sure array[i] >= array[i-1]
         swap(&array[i], &array[i-1]);
```

```
        }
    }
}
```

Walking through an example: array = {11, 34, 35, 5, 16, 10, 1}

When i = 1, array[i] >= array[i-1] so no swap is performed.

When i = 2, array[i] > array[i-1] so a swap is performed with array[i] and array[i-1].

The new array is [**11**, **35**, **34**, 5, 16, 10, 1]

When i = 3, array[i] < array[i-1] so a swap is performed with array[i] and array[i-1].

The new array is [**11**, **35**, **5**, **34**, 16, 10, 1]

When i = 4, array[i] <= array[i-1] so no swap is performed.

When i = 5, array[i] < array[i-1] so a swap is performed with array[i] and array[i-1].

The new array is [**11**, **35**, **5**, **34**, **10**, **16**, 1]

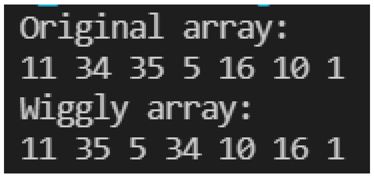
When i = 6, array[i] <= array[i-1] so no swap is performed.

Below is the implementation in C.

```
#include <stdio.h>
#define N 7

void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void Make_Wiggly(int* array){
    int i;
    for(i = 1; i < N; i++){
        if(((i%2==0)&&(array[i] > array[i-
1])) // if even, make sure array[i] <= array[i-1]
        ||((i%2==1)&&(array[i] < array[i-
1]))){ // if odd, make sure array[i] >= array[i-1]
            swap(&array[i], &array[i-1]);
        }
    }
}
```

```c
void print_array(int* array, int n){
   for (int i = 0; i < n; ++i){
      printf("%d ", array[i]);
   }
   printf("\n");
}

int main(){
   int array[N] = {11, 34, 35, 5, 16, 10, 1};
   printf("Original array:\n");
   print_array(array, N);
   Make_Wiggly(array);
   printf("Wiggly array:\n");
   print_array(array, N);
}
```

The output is below.



```
Original array:
11 34 35 5 16 10 1
Wiggly array:
11 35 5 34 10 16 1
```

Question 10:

If you are sorting a million items, roughly how much faster is heap sort than insertion sort? (Note: $\log(1{,}000{,}000) = 20$.)

Algorithm	Worst-Case Running Time
Insertion Sort	$O(n^2)$
Heap Sort	O(n*log(n))

When sorting one million items, the runtime of Insertion Sort will be roughly $O(1{,}000{,}000^2)$ or 1,000,000,000,000 operations. The runtime of Heap Sort will be O(1,000,000*log(1,000,000)) ≈ 1,000,000 * 20 = 20,000,000 operations.

$$\frac{1{,}000{,}000{,}000{,}000}{20{,}000{,}000} = 50{,}000$$

Therefore, when sorting 1,000,000 items, Heap Sort is roughly 50,000 times faster than Insertion Sort.