1

11 August 2023 09:15 PM

Zigzag Traverse

Write a function that takes in an n x m two-dimensional array (that can be square-shaped when n == m) and returns a one-dimensional array of all the array's elements in zigzag order.

Zigzag order starts at the top left corner of the two-dimensional array, goes down by one element, and proceeds in a zigzag pattern all the way to the bottom right corner.

Sample Input

```
array = [
  [1,  3,  4, 10],
  [2,  5,  9, 11],
  [6,  8, 12, 15],
  [7, 13, 14, 16],
]
```

Sample Output

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
```

```
the code is going to be a messy one, not so elegant with lot of if and else as we have to keep track of the direction everytime and multiple checks on the perimeter.

we push the current element in the result. we move down
 we check if are on col==0 or row==height, we have to make just 1 move either down or right. if we are not on col==0 or row==height, then we need to go diagonally down till we reach any of the col==0 or row==height so that flag goDown should become false.
same for row==0 ad col==width. if we are on either of these we need to move just once either right or down and make goDown =true for next iteration
else we need to digonally move up meaning row--, col++;
we have to run these till we go out of bound.

time complexity : O(n), where n is total no. of elements

space: O(n), as we need to store n elements in a result array and for any iterating stuff we don't need any auixiliary space we keep track of that using row and col variables.
```

Solution 1 | Solution 2 | Solution 3

```cpp
#include <vector>
using namespace std;
//O(1) fn to check if the traversing is not out of boundaries
bool isOutOfBound(int row, int col, int height, int width){
  return (row>height || row<0 || col<0 ||col>width);
}
vector<int> zigzagTraverse(vector<vector<int>> array) {
  int height =array.size()-1;
  int width =array[0].size()-1;
  vector<int>res = {};
  int row=0;
  int col=0;
  bool goDown = true;
  while(!isOutOfBound(row, col, height, width)){
    res.push_back(array[row][col]);
    if(goDown){
        if(col==0 || row==height){
          //if at boundaries for down, make going down false
          goDown = false;
            if(row==height)
              col++;
            else
              row++;
        }
        else{ // we have to move down diagonally till we reach for any boundary
            row++;
            col--;
        }
    }
    else{
        if(row==0 || col==width){
          // we are  on boundaries we need to move just once
          // going down is true for next iteration
            goDown = true;
            if(col==width)
              row++;
            else
              col++;
        }
        else{   //we are not on boundaries.
            row--;
            col++;
        }

    }
  }

  for(int i:res)
    cout << i << " ";
  return res;
}
```

2.1

12 August 2023 12:21 AM

```
dijkstra may or may not work for negative distance/weights as we use greedy approach and the direct connected path to a vertex from start vertex we consider it as the minimum distance b/w those two, but sometimes we may get better answer to it because of negative edge.

APPROACH:First we mark all dist as infinity, and a vector which shows inclusion of that vertex, initially it is false. then we pick a vertex using min function and mark it as true. we then mark its adjacent edges in dist array, for that we use relaxation which means distance of current vertex + the cost of current vertex to adjacent vertex should be less than cost of adjacent vertex.
if((d[u] + cost(u,v)) < d[v])
  d[v] = d[u] + cost[u,v]

time complexity : there are total n vertices, and in worst case we have to perform n relaxation(starting vertex is connected to all the other vertex).
time  = theta(n*n);
```

Dijkstra's Algorithm

You're given an integer `start` and a list `edges` of pairs of integers.

The list is what's called an adjacency list, and it represents a graph. The number of vertices in the graph is equal to the length of `edges`, where each index `i` in `edges` contains vertex `i`'s outbound edges, in no particular order. Each individual edge is represented by an pair of two numbers, `[destination, distance]`, where the destination is a positive integer denoting the destination vertex and the distance is a positive integer representing the length of the edge (the distance from vertex `i` to vertex `destination`). Note that these edges are directed, meaning that you can only travel from a particular vertex to its destination—not the other way around (unless the destination vertex itself has an outbound edge to the original vertex).

Write a function that computes the lengths of the shortest paths between `start` and all of the other vertices in the graph using Dijkstra's algorithm and returns them in an array. Each index `i` in the output array should represent the length of the shortest path between `start` and vertex `i`. If no path is found from `start` to vertex `i`, then `output[i]` should be `-1`.

Note that the graph represented by `edges` won't contain any self-loops (vertices that have an outbound edge to themselves) and will only have positively weighted edges (i.e., no negative distances).

If you're unfamiliar with Dijkstra's algorithm, we recommend watching the Conceptual Overview section of this question's video explanation before starting to code.

Sample Input

```
start = 0
edges = [
  [[1, 7]],
  [[2, 6], [3, 20], [4, 3]],
  [[3, 14]],
  [[4, 2]],
  [],
  [],
]
```

Sample Output

```
[0, 7, 13, 27, 10, -1]
```

2.2

12 August 2023 12:32 AM

Solution 1 Solution 2 Solution 3

```cpp
#include <vector>
using namespace std;
int maxCostVertex(vector<int>dist, vector<bool>vertexIncluded){
  int min = INT_MAX;
  int index;
  for(int i=0; i<dist.size(); i++){
    if(vertexIncluded[i]==false && dist[i]<=min){
      min = dist[i];
      index=i;
    }
  }
  return index;
}
vector<int> dijkstrasAlgorithm(int start, vector<vector<vector<int>>> edges) {
  // Write your code here;
  int size = edges.size();
  vector<bool> vertexIncluded(size, false);
  vector<int> dist(size, INT_MAX);
  dist[start]= 0;  //distance from start to start is 0.
  //iterate all the vertex for vertex start.
  for(int j=0; j<size; j++){
    //pick a max vertex from dist array and it should not be true in
    // vertexIncluded.
    int u = maxCostVertex(dist, vertexIncluded);
    vertexIncluded[u] = true;
    //mark all its adjacent edges
    for(vector<int> v : edges[u]){   //this will give the each vector of that index
      int ver = v[0];
      int cost = v[1];
      //perform relaxation
      if(vertexIncluded[ver]==false && (dist[u]+cost<=dist[ver])){
        dist[ver] = dist[u]+cost;
      }
    }
  }
  //replace all INT_MAX with -1;
  for(int i=0; i<size ;i++){
    if(dist[i]==INT_MAX || dist[i]== -(INT_MAX))
      dist[i]=-1;
  }
  return dist;
```

3.1

12 August 2023 12:21 AM

also called kahn's algroithm. it states- linear ordering of u->v, u should appear before v. it should be the DAG, directed, acyclic graph. it a cycle is there v can come before u which violates the algo. Topological don't give an unique result, it can be different.

approach:(using BFS and queue)
we first make a array of indegree of all the nodes. then we make a queue and start traversing the indegree array, if any node indegree is 0, we push it in the queue.
once the traversing is complete we will have a minimum of 1 node in queue. then we will take out a node from queue and pop it, and will look for adjacent edges. we will decrement it and will check if after decrementing if the indegree of that node becomes 0 if yes then we get a element we push it in the queue. run till q is not empty.
here we are using size of resultant and size of nodes. if they are equal then no cycle.

time complexity : O(v+e), where v is vertices and e is edges.

approach:(using DFS and stack)
we make a visited array and initiate it with 0. we calculate the total nodes given. we start DFS with the first node and check if it is not visited and if has adjacent node we call DFS again on it. if not having adjacent and push it in the stack. then we take out in the LIFO manner and that's the one linear ordering of that graph.
we are using DFS to check if graph has cycle if yes simply return {}.

Topological Sort

You're given a list of arbitrary jobs that need to be completed; these jobs are represented by distinct integers. You're also given a list of dependencies. A dependency is represented as a pair of jobs where the first job is a prerequisite of the second one. In other words, the second job depends on the first one; it can only be completed once the first job is completed.

Write a function that takes in a list of jobs and a list of dependencies and returns a list containing a valid order in which the given jobs can be completed. If no such order exists, the function should return an empty array.

Sample Input

```
jobs = [1, 2, 3, 4]
deps = [[1, 2], [1, 3], [3, 2], [4, 2], [4, 3]]
```

Sample Output

```
[1, 4, 3, 2] or [4, 1, 3, 2]
```

3.2

12 August 2023 12:36 AM

Solution 1 Solution 2 Solution 3

```cpp
#include <vector>
#include<queue>
using namespace std;

//topo sort using BFS.

vector<int> topologicalSort(vector<int> jobs, vector<vector<int>> deps) {
  // Write your code here.
  int size = jobs.size();

  vector<int> indegree(size+1, 0);
  vector<int> result;

  //mark the indegree of nodes from 1.
  for(vector<int> i : deps){
    int adjacent_node  = i[1];
    indegree[adjacent_node] ++;
  }

  queue<int> q;

  //find if there is 0th vertex, as most vertex start from 1, but in
  // some it starts with 1, so we handle both cases using a flag.
  int flag=0;
  for(int i=0; i<size; i++){
    if(jobs[i]==0)
      flag=1;
  }

  if(flag==1){
    for(int i=0; i<size;i++){
      if(indegree[i]==0){
        q.push(i);
      }
    }
  }else{
  for(int i=1; i<=size; i++){
    if(indegree[i]==0){
      q.push(i);
    }
  }
  }

  //minimum 1 element will be there in the queue
  while(!q.empty()){
    int node = q.front();
    q.pop();
    result.push_back(node);
    for(vector<int> i:deps){
      if(i[0]==node){
        int adjacent = i[1];
        indegree[adjacent]--;
        if(indegree[adjacent]==0)
          q.push(adjacent);
      }
    }
  }
  if(result.size() != size)
    return {}; //cycle is there
  return result;
}
```

3.3

12 August 2023 12:36 AM

Solution 1 | Solution 2 | Solution 3

```cpp
#include <vector>
#include<stack>
using namespace std;
//DFS

bool cycleCheck(int node, vector<vector<int>>deps, vector<int>&vis, vector<int>&pat
  vis[node]=1;
  pathV[node]=1;

  //traversing adjacent
  for(vector<int>i:deps){
    if(i[0]==node){
      if(!vis[i[1]]){
        if(cycleCheck(i[1], deps, vis, pathV)==true)
          return true;
      }
      else if(pathV[i[1]]==1)  //node is previously visited and in the same path
        return true;
    }
  }

  pathV[node]=0; //if cycle is not found in that way, marking that node path 0 so t
                 // real cycle can be found.
  return false;
}

void DFS(int node, vector<int>& vis, stack<int>&s, vector<vector<int>> deps){
  vis[node] =1;
  for(vector<int> i: deps){
    if(i[0]==node){
      int adjacent=  i[1];
      if(vis[adjacent]==0)
        DFS(adjacent, vis, s, deps);
    }
  }
  s.push(node);
}
```

```cpp
bool isCycle(vector<int> jobs, vector<vector<int>>deps){
  int size = jobs.size();
  vector<int> Nodevisited(size, 0);
  vector<int> pathVis(size,0);
  for(int i=0; i<size; i++){
    if(!Nodevisited[i]){
      if(cycleCheck(i, deps, Nodevisited, pathVis)==true)
        return true;
    }
  }
  return false;
}
vector<int> topologicalSort(vector<int> jobs, vector<vector<int>> deps) {
  // Write your code here.
  int size= jobs.size();
  //cycle check
  bool isCyc = isCycle(jobs, deps);
  if(isCyc)
    return {};
  vector<int> visited(size+1, 0);
  stack<int> s;
  vector<int> res;
  for(int i=0; i<size; i++){
    int node = jobs[i];
    if(visited[node]==0){
      DFS(node, visited, s ,deps);
    }
  }
  while(!s.empty()){
    int node = s.top();
    s.pop();
    res.push_back(node);
  }

 if(res.size()!=size){
   return {};
 }
  return res;
}
```

4.1

12 August 2023 12:21 AM

KRUSKAL algo:
It says find a minimum cost edge and make sure its not forming a cycle. it works on connected graph. if non connected graph is given, kruskal finds minimum cost spanning tree for each component, we can't find spanning tree for non connected graph.

In Kruskal's algorithm, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first at the maximum weighted edge at last. Thus we can say that it makes a locally optimal choice in each step in order to find the optimal solution. Hence this is a Greedy Algorithm.

Time complexity: (when using min heap)
as we are selecting (vertices-1) edges and for each edge we are finding min cost out of all edges which take log n time in min heap.
time = O(n.logn)

we create a adjacency matrix total col of edges each row represent vertex1, vertex 2, weights resp. we iterate through the last row and find minimum cost and check if that edge is included or not using included array and that edge is not forming cycle we will use find and union for that. a matrix to store the min cost edges to of length v-1.

Kruskal's Algorithm

You're given a list of `edges` representing a weighted, undirected graph with at least one node.

The given list is what's called an adjacency list, and it represents a graph. The number of vertices in the graph is equal to the length of `edges`, where each index `i` in `edges` contains vertex `i`'s siblings, in no particular order. Each of these siblings is an array of length two, with the first value denoting the index in the list that this vertex is connected to, and and the second value denoting the weight of the edge. Note that this graph is undirected, meaning that if a vertex appears in the edge list of another vertex, then the inverse will also be true (along with the same weight).

Write a function implementing Kruskal's Algorithm to return a new `edges` array that represents a minimum spanning tree. A minimum spanning tree is a tree containing all of the vertices of the original graph and a subset of the edges. These edges should connect all of the vertices with the minimum total edge weight and without generating any cycles.

If the graph is not connected, your function should return the minimum spanning forest (i.e. all of the nodes should be able to reach the same nodes as they could in the initial edge list).

Note that the graph represented by `edges` won't contain any self-loops (vertices that have an outbound edge to themselves) and will only have positively weighted edges (i.e., no negative distances).

If you're unfamiliar with Kruskal's algorithm, we recommend watching the Conceptual Overview section of this question's video explanation before starting to code. If you're unfamiliar with the Union Find data structure, we recommend completing that problem before attempting this one.

Sample Input

```
edges = [
  [[1, 3], [2, 5]],
  [[0, 3], [2, 10], [3, 12]],
  [[0, 5], [1, 10]],
  [[1, 12]]
]
```

Sample Output

```
[
  [[1, 3], [2, 5]],
  [[0, 3], [3, 12]],
  [[0, 5]],
  [[1, 12]]
]
```

4.2

12 August 2023 12:38 AM

```cpp
using namespace std;
int find(int v ,vector<int>&parents){
  if(v!=parents[v]){
    parents[v] =find(parents[v], parents);
  }
  return parents[v];
}

void createUnion(int ver1Root, int ver2Root, vector<int>&parents, vector<int>&ranks){
  if(ranks[ver1Root] < ranks[ver2Root])
    parents[ver1Root] = ver2Root;
  else if(ranks[ver1Root] > ranks[ver2Root])
    parents[ver2Root] = ver1Root;
  else{
    parents[ver2Root] = ver1Root;
    ranks[ver1Root]++;
  }
}

vector<vector<vector<int>>> kruskalsAlgorithm(vector<vector<vector<int>>> edges) {
  // Write your code here.

  // just converting edges 3d into 2d and each vector in 2d is [v1, v2, weight]
  vector<vector<int>> sortedEdges;
  for(int sourceIndex=0; sourceIndex<edges.size(); sourceIndex++){
    vector<vector<int>> vertex = edges[sourceIndex];
    for(vector<int> edge : vertex){
      if(edge[0] > sourceIndex){
        sortedEdges.push_back({sourceIndex, edge[0], edge[1]});
      }
    }
  }
```

```cpp
  //sorting the vectors in 2d vectors based on 3rd element of each vector which is weight
  sort(sortedEdges.begin(), sortedEdges.end(),
    [](vector<int>edge1, vector<int>edge2)->bool{return edge1[2] < edge2[2];});

  vector<int> parents;
  vector<int> ranks;
  vector<vector<vector<int>>>mst;

  for(int i=0; i<edges.size(); i++){
    parents.push_back(i);
    ranks.push_back(0);
    mst.push_back({});
  }

  //as the edges are sorted , we are taking it one at a time and
  //checking if edges dont make cycles.
  for(vector<int>edge:sortedEdges){
    int ver1Root = find(edge[0], parents);
    int ver2Root = find(edge[1], parents);
    if(ver1Root!=ver2Root){
      mst[edge[0]].push_back({edge[1], edge[2]});
      mst[edge[1]].push_back({edge[0], edge[2]});   //as they are undirected, we are sending both edges di
      createUnion(ver1Root, ver2Root, parents, ranks);
    }
  }

  return mst;
}
```

5

12 August 2023 12:21 AM

Reverse Linked List

Write a function that takes in the head of a Singly Linked List, reverses the list in place (i.e., doesn't create a brand new list), and returns its new head.

Each `LinkedList` node has an integer `value` as well as a `next` node pointing to the next node in the list or to `None` / `null` if it's the tail of the list.

You can assume that the input Linked List will always have at least one node; in other words, the head will never be `None` / `null`.

Sample Input

```
head = 0 -> 1 -> 2 -> 3 -> 4 -> 5 // the head node with value 0
```

Sample Output

```
5 -> 4 -> 3 -> 2 -> 1 -> 0 // the new head node with value 5
```

```
approach:
we take three pointers prev tracks the prev node initialized by nullptr, currentNode, nextNode points to head. we have to traverse till nextNode is not null.
first step is to secure currentNode next node address by using nextNode. then we update currentNode next to prev node, then prev comes forward on currentNode, currentNode moves to nextNode.
when the while loop is completed. next and currentNode points to nullptr and prev points on last node, so we make head= prevNode.


time complexity : O(n), n= no. of nodes.


space : O(1)
```

Solution 1 Solution 2 Solution 3

```cpp
using namespace std;

class LinkedList {
 public:
  int value;
  LinkedList* next;

  LinkedList(int value) {
    this->value = value;
    this->next = nullptr;
  }
};

LinkedList* reverseLinkedList(LinkedList* head) {
  // Write your code here.
  LinkedList* currentNode = head;
  LinkedList* nextNode = head;
  LinkedList* prev = nullptr;

  if(head->next==nullptr){
    return head;
  }

  while(nextNode!=nullptr){
     nextNode = currentNode->next;
      currentNode->next = prev;
      prev = currentNode;
      currentNode= nextNode;
  }

  head = prev;



  return head;
}
```

6.1

12 August 2023 12:21 AM

Merge Linked Lists

Write a function that takes in the heads of two Singly Linked Lists that are in sorted order, respectively. The function should merge the lists in place (i.e., it shouldn't create a brand new list) and return the head of the merged list; the merged list should be in sorted order.

Each `LinkedList` node has an integer `value` as well as a `next` node pointing to the next node in the list or to `None` / `null` if it's the tail of the list.

You can assume that the input linked lists will always have at least one node; in other words, the heads will never be `None` / `null`.

Sample Input

```
headOne = 2 -> 6 -> 7 -> 8 // the head node with value 2
headTwo = 1 -> 3 -> 4 -> 5 -> 9 -> 10 // the head node with value 1
```

Sample Output

```
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 // the new head node wi
```

```
1 we are given two sorted linked list. initial step if we compare first and
  second head which ever is less it becomes the mergehead and then the
  repeating steps starts.
2 we check both value which is less, helper points to that node, and that
  head points to next head. this we do till headOne and headTwo are not
  null. if any one of them is null stop the loop and check which pointer is
  still not null. the one which is not null is attached to helper->next.
3
4 time complexity :
5 theta(m+n)
6 where m is no. of nodes in first linkedList and n in second.
7
8 merging is suitable in linked list as it does not require extra space
  whereas in arrays for merging we require extra space.
```

6.2

12 August 2023 12:44 AM

Solution 1 Solution 2 Solution 3

```cpp
#include <vector>

using namespace std;

// This is an input class. Do not edit.
class LinkedList {
 public:
  int value;
  LinkedList* next;

  LinkedList(int value) {
    this->value = value;
    next = nullptr;
  }
};
```

```cpp
LinkedList* mergeLinkedLists(LinkedList* headOne, LinkedList* headTwo) {
  // Write your code here.
  LinkedList *headMerge;
  LinkedList *helper;

  if(headOne->value < headTwo->value){
    helper=headOne;
    headMerge=headOne;
    headOne = headOne->next;
    helper->next = nullptr;

  }else{
    helper=headTwo;
    headMerge=headTwo;
    headTwo = headTwo->next;
    helper->next = nullptr;

  }
  while(headOne!=nullptr && headTwo!=nullptr){
      if(headOne->value < headTwo->value){
        helper->next = headOne;
        helper=headOne;
        headOne = headOne->next;
        helper->next=nullptr;
      }else{
        helper->next = headTwo;
        helper = headTwo;
        headTwo = headTwo->next;
        helper->next=nullptr;
    }
  }
  // now we will check if any list is left we will just add it to the merg
  if(headOne!=nullptr){
    helper->next=headOne;
  }else{
    helper->next = headTwo;
  }
  return headMerge;
}
```

7

12 August 2023 12:21 AM

Solution 1 Solution 2 Solution 3

```cpp
using namespace std;
class LinkedList {
 public:
  int value;
  LinkedList* next;

  LinkedList(int value) {
    this->value = value;
    next = nullptr;
  }
};
int length(LinkedList* head){
  int length = 0;
  while(head){
    head= head->next;
    length++;
  }
  return length;
}
LinkedList* shiftLinkedList(LinkedList* head, int k) {
  // Write your code here.
  if(head==NULL || head->next==nullptr)
    return head;
  int l = length(head);
  k = k%l;    // if k is larger than the length.
  if(k<0)
    k = l + k;
  for(int i=0; i<k; i++){
    LinkedList* follower= head;
    while(follower->next->next!=nullptr){
    follower = follower->next;
    }
    LinkedList* last = follower->next;
    follower->next = nullptr;
    last->next = head;
    head = last;
  }
  return head;
}
```

Shift Linked List

Write a function that takes in the head of a Singly Linked List and an integer `k`, shifts the list in place (i.e., doesn't create a brand new list) by k positions, and returns its new head.

Shifting a Linked List means moving its nodes forward or backward and wrapping them around the list where appropriate. For example, shifting a Linked List forward by one position would make its tail become the new head of the linked list.

Whether nodes are moved forward or backward is determined by whether `k` is positive or negative.

Each `LinkedList` node has an integer `value` as well as a `next` node pointing to the next node in the list or to `None` / `null` if it's the tail of the list.

You can assume that the input Linked List will always have at least one node; in other words, the head will never be `None` / `null`.

Sample Input

```
head = 0 -> 1 -> 2 -> 3 -> 4 -> 5 // the head node with value 0
k = 2
```

Sample Output

```
4 -> 5 -> 0 -> 1 -> 2 -> 3 // the new head node with value 4
```

```
shifting means moving the linked list forward or backward.
positive value for k means shift forward for negative move backward.

for each value of k, we find the prev and last node and update prev->next
to null and last->next to head and make head point to last.
for negative value of k, we can convert to to the equivalent of positive
by doing k= k + length.

we do k= k%l to handle cases where k might be larger than the number of
nodes in the linked list, cause suppose length is 5 and key is 7 . then
shifting for 7 times is equivalent to 2 times.

Time Complexity: O(Number of nodes present in the list*k)
Reason: For k times, we are iterating through the entire list to get the
last element and move it to first.
```

8

12 August 2023 12:29 AM

Quickselect

Write a function that takes in an array of distinct integers as well as an integer `k` and that returns the kth smallest integer in that array.

The function should do this in linear time, on average.

Sample Input

```
array = [8, 5, 2, 9, 7, 6, 3]
k = 3
```

Sample Output

```
5
```

```
so the brute force can be sort the array and return k-1th index element from array. it will be O(nlogn).

we will use quick sort, not complete we fill find a pivot partition point and if its index+1 is equal to k, we find the element else we check if(index+1) < k, then we run partition on right side else on left side.

instead of running recursively , we use run while(1) which will run indefinite times unless there's an explicit break statement inside the loop.

it runs  in O(n) time in average and O(n^2) in worst time scenario.
```

```cpp
#include <vector>
using namespace std;

int partition(vector<int>&arr, int low, int high){
  int pivot= arr[low];
  int i=low; int j=high;
  while(i<j){
    while(arr[i]<=pivot)
      i++;
    while(arr[j]> pivot)
      j--;

    if(i<j)
      swap(arr[i], arr[j]);
  }
  swap(arr[low], arr[j]);
  return j;
}
int quickselect(vector<int> array, int k) {
  // Write your code here.
  int size = array.size();
  if(size==0)
    return -1;
  if(size==1 && k==1)
    return array[0];
  int low=0; int high=size-1;
  int kth;
  while(1){
    int idx = partition(array, low, high);
    if(idx+1==k){
      kth = array[idx];
      break;
    }
    if (idx + 1< k ) {
      low = idx + 1;  //run on right side
   } else {
      high = idx - 1;  //run on left side.
    }
  }
  return kth;
}
```

9

12 August 2023 12:29 AM

Quick Sort

Write a function that takes in an array of integers and returns a sorted version of that array. Use the Quick Sort algorithm to sort the array.

If you're unfamiliar with Quick Sort, we recommend watching the Conceptual Overview section of this question's video explanation before starting to code.

Sample Input

```
array = [8, 5, 2, 9, 5, 6, 3]
```

Sample Output

```
[2, 3, 5, 5, 6, 8, 9]
```

```
quick sort uses partitioning function.
what it does is it returns the position of the first element of index after its sorting position.
We take first element as pivot, then we run a i from start which looks for greater element from pivot and j from last which looks for equal or less than pivot. when they both found it we swap. we run this while i<j.
when it comes out of while loop we swap pivot element with A[j]. then j is the partition point and we return it. then we run quick sort on(left side of partition point and right side recursively.


quick sort doesn't mean it is fast. quick sort follows divide and conquer strategy and divide the array into sub array and run same quick sort


Time complexity: O(N(logN))
```

```cpp
#include <vector>
using namespace std;
int partitioningFunction(vector<int> &arr,int low, int high){
  int pivot = arr[low];
  int i=low, j=high;
  while (i <= j) {
        while (i <= j && arr[i] <= pivot)
            i++;
        while (i <= j && arr[j] > pivot)
            j--;

        if (i < j) {
            swap(arr[i], arr[j]);
            i++;
            j--;
        }
    }
  swap(arr[low], arr[j]);
  return j;
}

void quickSortRecursively(vector<int>&arr, int low, int high){
  if(low< high){
    int j= partitioningFunction(arr, low, high);
    quickSortRecursively(arr, low, j-1);
    quickSortRecursively(arr, j+1, high);
  }
}

vector<int> quickSort(vector<int> array) {
  // Write your code here.
  int size = array.size();
  quickSortRecursively(array, 0 ,size-1);
  for(int i:array)
    cout << i << " ";

  return array;
}
```

10

12 August 2023 12:29 AM

```cpp
#include <vector>
using namespace std;

void heapify(vector<int>&array, int last, int i){
  int parent = i;
  int l = 2*i+1;
  int r = 2*i+2;
  int large = l;
  if(l < last && array[l] > array[parent])
    parent  = l;
  if(r<last && array[r] > array[parent])
    parent = r;
  if(parent!=i){ //means left or right is greater than parent, just swap with the right one
    swap(array[i], array[parent]);

    heapify(array, last, parent); //runnin recursively to adjust that particular element.
  }
  // if parent was equal to i after comparing it with l and r
  // means parent was the highest then there was no need of modify.
}

vector<int> heapSort(vector<int> array) {
  // Write your code here.
  int n=array.size();
  //running heapify from last non leaf to the 0, as left of last non
  //leaf node are parents of some child.
  int lastParentIndex = n/2 -1;
  for(int i = lastParentIndex; i>=0; i--){
    heapify(array, n, i);
  }
  // it will rearrange the array, the max element at the start for each heapify
    // then what  we do is separate that element at the last and run heapify on
    // the first element.
    for(int i=n-1; i>=0; i--){
      swap(array[0],array[i]); //bring max element at last.
      heapify(array, i, 0); //run heapify on reduced array everytime
    }
  return array;
}

```

Heap Sort

Write a function that takes in an array of integers and returns a sorted version of that array. Use the Heap Sort algorithm to sort the array.

If you're unfamiliar with Heap Sort, we recommend watching the Conceptual Overview section of this question's video explanation before starting to code.

Sample Input

```
array = [8, 5, 2, 9, 5, 6, 3]
```

Sample Output

```
[2, 3, 5, 5, 6, 8, 9]
```

```
Heap Sort:
Create Heap
Delete Heap(delete an element and store it in the array too)
it will sort, so it is heap sort.

TWO APPROACHES CAN BE THERE:
1. Create the heap using insert fn , and then delete and store element at last.

2. use heapify for bottom up approach, and run heapify from last non leaf node to all the left node in array. Then when creation is completed, swap first and last element , run heapify on the 0 element with size of array excluding the last element placed using swap.


NOTE : if vector starts from 0.
0th left child is at 2*i+1;
right child = 2*i+2;

ith parent = floor((i-1)/2) or without floor.
ex if it returns 2.5, floor will increase to 3.

TIME COMPLEXITY-:
O(N*logN), as heap is a binary complete tree, height is logN, and we are running for n elements.
```

11.1

12 August 2023 12:29 AM

Radix Sort

Write a function that takes in an array of non-negative integers and returns a sorted version of that array. Use the Radix Sort algorithm to sort the array.

If you're unfamiliar with Radix Sort, we recommend watching the Conceptual Overview section of this question's video explanation before starting to code.

Sample Input

```
array = [8762, 654, 3008, 345, 87, 65, 234, 12, 2]
```

Sample Output

```
[2, 12, 65, 87, 234, 345, 654, 3008, 8762]
```

```
We sort the array using the number system, if the digits are in decimal system we take an array of size 10(0-9).
suppose if no. were in binary ony array of size 2 was required.and we see the largest system if it is of 3 digits, we require 3 passed to sort.
How it works?In first pass we arrange the element in the array based on last digit.( (A[i]/1)%10 gives the last digit.
then we pop the elements and send it to the array.
in second pass we arrange the elements as based on second digit. ( (A[i]/10)%10 gives second digit.
In third pass we arrange the elements as based on third digit. ( (A[i]/100)%10 gives third digit.

Time complexity:
we are copying n elements d no. of times(d is passed based on the largest element digits).
O(d*n)

we also use count sort in radix sort.
we count the occurance of the digit for each pass and increment it in the count array which is (0-9). update count[i] so that count[i] now contains actual position
of the digits for current pass in output array[we know this by analysing]. then we start filling o/p from last element of main array. to fill o/p we use.
we check the digit of the element acc to the pass. then we go to that index of count as count contains the position(which starts from 1) we decrement it to find the position in output array as outputArray starts from 0.
```

11.2

12 August 2023 12:54 AM

```cpp
#include <vector>
using namespace std;
int findMax(vector<int> arr){
  int max=arr[0];
  for(int i=1; i<arr.size();i++){
    if(arr[i]>max)
      max= arr[i];
  }
  return max;
}
void countSort(vector<int>&arr, int size, int dec){
  vector<int> count(10, 0);
  vector<int> output(size);
  //count the occurance of digit for the current pass
  for(int i=0; i<size; i++){
    count[(arr[i]/dec)%10]++;
  }
  //update count so that count[i] contains actual position
  // of digit in the o/p array.
  for(int i=1; i<10; i++){
    count[i]+=count[i-1];
  }
  //build output array
  for(int i=size-1; i>=0; i--){
    output[count[(arr[i]/dec)%10] -1 ] = arr[i];
    count[(arr[i] / dec) % 10]--; // this is decrementing the count value for that index
    // cause if that digits come again we can get the position.
    // suppose 398, take 8 index in count whatever the value is decrease by 1 to find
    // position in output array, then decrement that count value by 1
    // so that if suppose 298, we have 8th again if we didn't decrement it
    // we would have overlap the element in same index.
    // 398, count[8] = 7. fill 398 in 6th index of o/p [as 7 is position
    // which starts from 1, 6th is index which starts from 0].
    // decrement count[8] = 6. now if any element has digit 6, it will
    // be filled in 5th index.
  }

  //copy from output to array to reflect changes done from the pass
  for(int i=0; i<size; i++)
    arr[i]= output[i];
}



vector<int> radixSort(vector<int> array) {
  // Write your code here.
  int size=array.size();
  if(size==0)
    return {};

  int max =findMax(array);

  //iterate the passes for max element digits.
  // max/dec > 0 to run for the passes we need.
  for(int dec=1; max/dec >0; dec= dec*10){
    countSort(array, size, dec);
  }

  return array;

}
```