ESO207 Theoretical Assignment 2

P1 Search Complicated

You are given an array $A[0, ..., n-1]$ of n distinct integers. The array has following three properties:

- First $(n-k)$ elements are such that their value increase to some maximum value and then decreases.
- Last k elements are arranged randomly
- Values of last k elements is smaller compared to the values of first $(n-k)$ elements.

(a)You are given q queries of the variable Val. For each query, you have to find out if Val is present in the array A or not. Write a pseudo-code for an $\mathcal{O}(k\log(k)+q\log(n))$ time complexity algorithm to do the task. (Higher time complexity correct algorithms will also receive partial credit)

`Motivations:`

- $k\log(k)$ term strongly suggests sorting last k elements
- $\log(n)$ again suggests binary search

▷ Since, we know that last k elements are arranged randomly, and they are smaller than the first $(n-k)$ elements, we sort them first in decreasing order. Now, the resultant array is such that the elements increase upto some maximum value and then decrease.

▷ As discussed in the lectures, we can find a local maxima in an array in $\mathcal{O}(\log(n))$ time. Also, the only local maxima in this array is the element upto which the elements increase and then decrease. So, we find its index, say j.

▷ Given the pseudo-sorted nature of the array, we employ triple binary search.

- If **Val** $< A[0]$ and **Val** $< A[n-k-1]$: **Val** is smaller than the first $(n-k)$ elements, search in the last k.
- If it isn't : Search in the range $[0, j]$. If not present, then search in $[j+1, n-k-1]$.

`Pseudocode:`

```
function ComplikATED-Search(q, A, n, k, Val)
    A.sort(n − k, n − 1)
    j = Loc-Max(A, n)
    for i → 0, q − 1 do
        print(q, Find-Val(A, n, j, k, Val))
    end for
```

```
function Find-Val(A, n, j, k, Val)
    if Val < A[0] and Val < A[n − k − 1] then
        i ← Bin-Search(A, n − k, n − k − 1, Val)
    else
        i ← Bin-Search(A, 0, j, Val)
        if i == −1 then
            i ← Bin-Search(A, j + 1, n − k − 1, Val)
```

```
        end if
    end if
    return i
```

```
function Loc-Max(A, n)
    l ← 0
    r ← n − 1
    while l < r do
        mid ← ⌊(l+r)/2⌋
        if A[mid] > A[mid + 1] then
            r ← mid                          ▷ Potential local maximum on the left side
        else
            l ← mid + 1                      ▷ Potential local maximum on the right side
        end if
    end while
    return l
```

```
function Bin-Search(A, l, r, x)
    while l ≤ r do
        mid ← ⌊(l+r)/2⌋
        if A[mid] = x then
            return mid
        end if
        if A[mid] < x then
            l ← mid + 1
        else
            r ← mid − 1
        end if
    end while
    return −1
```

(b) Explain the correctness of your algorithm and give the complete time complexity analysis for your approach in part (a).

We know that for a given sorted array, we can apply binary search and check whether an element exists or not. Now, after sorting the last k elements of original array, we can subdivide the resultant one in three arrays each of them sorted:

- $[0 : j]$: sorted in ascending order
- $[j : n-k-1]$: sorted in descending order
- $[n-k : n-1]$: sorted in descending order

(We obtain the value of j by finding the local(infact maximum) maxima)
Thus, we search in these 3 arrays and return its index if found else return -1.

`Time Complexity Analysis :`

```
function Complikated-Search(q, A, n, k, Val)
    A.sort(n − k, n − 1)                     ▷ O(k log(k))
    for i → 0, q − 1 do
        print(q, Find-Val(A, n, j, k, Val))  ▷ O(log(n))
    end for
```

```
function FIND-VAL(A, n, j, k, Val)
    if Val < A[0] and Val < A[n − k − 1] then
        i ← BIN-SEARCH(A, n − k, n − 1, Val)                    ▷ O(log(k))
    else
        i ← BIN-SEARCH(A, 0, j, Val)                            ▷ O(log(j))
        if i == −1 then
            i ← BIN-SEARCH(A, j + 1, n − k − 1, Val)            ▷ O(log(n − k))
        end if
    end if
    return i
```

Now, $\log(k), \log(n-k), \log(j) \leq log(n)$ and in the worst case either of the three arrays will be searched completely; for a single query, the time complexity is $\log(k) + \log(j) + \log(n-k) = \mathcal{O}(\log(n))$.
Thus, for q queries, it is $T(n) = q\mathcal{O}(\log(n)) + \log(n) + \mathcal{O}(k\log(k)) = \mathcal{O}(k\log(k) + q\log(n))$

P2 Perfect Complete Graph

A directed graph with n vertices is called Perfect Complete Graph if:

- There is exactly one directed edge between every pair of distinct vertices.
- For any three vertices a, b, c, if (a, b) and (b, c) are directed edges, then (a, c) is present in the graph.

Note: Outdegree of a vertex v in a directed graph is the number of edges going out of v

(a) Prove that a directed graph is a Perfect Complete Graph if and only if between any pair of vertices, there is at most one edge, and for all $k \in \{0, 1, ..., n-1\}$, there exist a vertex v in the graph, such that Outdegree$(v) = k$.

$(\Longleftarrow)$
We start by proving a lemma,
Lemma. Given a graph with vertices having at most one edge between any pair of vertices, such that for all $k \in \{0, 1, ..., n-1\}$, there exist a vertex v in the graph having **Outdegree**$(v) = k$, there can exist exactly **one** edge between any pair of vertices.
Proof. If **Outdegree** of a vertex is k, then there exist at least k edges, i.e. in such a graph there are at least (infact exactly) $0 + 1 + 2 + ... + (n-1) = \frac{n(n-1)}{2}$. On the other hand, if we assume exactly one edge between each pair of vertices, we get $\binom{n}{2} = \frac{n(n-1)}{2}$ edges. Thus, even if one edge is absent, the sum fall shorts of.

Lemma. **Indegree** of a vertex in a graph with n vertices and exactly one edge between any pairs of vertices can be given by **Indegree**(v) + **Outdegree**$(v) = (n-1)$
Proof. Except the given vertex, there are $n-1$ other vertices and there exact exactly one edge between any pair of vertices. Thus, the above equation satisfies.

Suppose we are given graph with n vertices such that for all $k \in \{0, 1, ..., n-1\}$ there exists a vertex v. Now, we can label the graph such that a vertex v_k denotes that **Outdegree**$(v) = k$. Consider v_0 which has **Outdegree** $= 0$ and **Indegree** $= (n-1)$. Thus, it has an incoming edge from all vertices $v_i, i > 0$. Now, consider v_1 with **Outdegree** $= 1$ and v_0 has an incoming edge from v_1, i.e. (v_1, v_0) is an edge.
Thus, we invoke Induction here by stating that for a vertex v_i, we have outward edges to $v_j, j < i$ and inward edges from $v_k, k > i$.
Consider v_{i+1}, we have an outward edge from v_{i+1} to all $v_m, m \leq i$, thus we cannot have inward edges from these vertices. As a result v_{i+1} has all inward edges from $v_p, p > i+1$.
Thus, for any two vertices v_i, v_j and $i < j \Leftrightarrow$ we have an edge (v_j, v_i). This leads to the second property viz. for v_i, v_j, v_k, if we have the edges $(v_k, v_j), (v_j, v_i) \implies k > j > i \implies (v_k, v_i)$ is an edge. Thus proving our claim.

$(\Longrightarrow)$
We observed that $0+1+2+...+(n-1) = \frac{n(n-1)}{2} = \binom{n}{2}$. We will use this later. We prove another lemma,
Lemma. No two vertices can have the same **Outdegree** given they follow the two conditions.
Proof. Consider two vertices u and v with the same **Outdegree** $= k$. Now, since there can be exactly one edge between u and v, consider WLOG we have the directed edge (u, v). Consider the pairs (u, v) and (v, w) where (v, w) are the outward edges from v (also, $w \neq u$, since only one edge), exactly k in number. Thus, by the second condition, we also have the edges (u, w) contributing k to the **Outdegree** making total **Outdegree**$(u) = (k+1)$ thus contradicting our assumption.

Now, since we have exactly one directed edge between every pair of vertices, total number of edges are $\binom{n}{2}$, and every outward edge is unique, we must have that the total of **Outdegree** over all vertices be equal to $\binom{n}{2}$.

Again, we cannot modify the **Outdegrees** of vertices while keeping the sum intact, as that would lead to multiple vertices having the same **Outdegree** which is not valid as proved in the above lemma. Thus, our claim remains true.

(b) Given the adjacency matrix of a directed graph, design an $\mathcal{O}(n^2)$ algorithm to check if it is a perfect complete graph or not. Show the time complexity analysis. You may use the characterization given in part (a).

`Motivations:`

- Just used the given hint :)

(We assume no cycles and no parallel edges are present) We use the results obtained in (a), and incorporate the checking of the two properties in our algorithm. We run 2 loops:

1. Checking that **Outdegree** of a vertex is in $\{0, 1, 2, ..., n-1\}$
2. Checking that there is exactly one edge between all u and v

---

`Pseudocode:`

```
function IS-COMPERFECT(G, n)
    od ← set{0, 1, 2, ..., n − 1}

    for i → 0, n − 1 do
        odsum ← 0
        for j → 0, n − 1 do
            odsum ← odsum + G[i][j]
        end for
        od.delete(odsum)
    end for

    if !isEmpty(od) then
        return −1
    end if

    for i → 0, n − 1 do
        for j → 0, i − 1 do
            if ((G[i][j] + G[j][i]) % 2) == 0 then
                return −1
            end if
        end for
    end for

    return 1
```

`Time Complexity Analysis :`

function IS-COMPERFECT(G, n)
$od \leftarrow set\{0, 1, 2, ..., n-1\}$

for $i \rightarrow 0, n-1$ **do**
$odsum \leftarrow 0$
for $j \rightarrow 0, n-1$ **do**
$odsum \leftarrow odsum + G[i][j]$
end for
$od.delete(odsum)$ ▷ max $\mathcal{O}(\log n)$
end for

if !isEmpty(od) **then** ▷ max $\mathcal{O}(\log n)$
return -1
end if

$\mathcal{O}(n^2)$

for $i \rightarrow 0, n-1$ **do**
for $j \rightarrow 0, i-1$ **do**
if $((G[i][j] + G[j][i])\ \%\ 2) == 0$ **then**
return -1
end if
end for
end for

return 1

We can implement the set deletion operation in $\mathcal{O}(1)$ or $\mathcal{O}(\log(n))$ and likewise for the isEmpty() function.
Thus, the nested loop dominates giving us total $\mathcal{O}(n^2)$ complexity.

$\mathcal{O}(n^2)$

P3 PnC

You are given an array $A = [a_1, a_2, a_3, ..., a_n]$ consisting of n **distinct, positive** integers. In one operation, you are allowed to swap the elements at any two indices i and j in the **present array** for a cost of $\max(a_i, a_j)$. You are allowed to use this operation any number of times.
Let Π be a permutation of $1, 2, ..., n$. For an array A of length n, let $A(\Pi)$ be the permuted array $A(\Pi) = [a_{\Pi(1)}, a_{\Pi(2)}, ..., a_{\Pi(n)}]$.
We define the score of an array A of length n as

$$S(A) = \sum_{i=1}^{i=n-1} |a_{i+1} - a_i|$$

(a) Explicitly characterise all the permutations $A(\Pi_0) = [a_{\Pi_0(1)}, a_{\Pi_0(2)}, a_{\Pi_0(3)}, ..., a_{\Pi_0(n)}]$ of A such that

$$S(A(\Pi_0)) = \min_{\Pi} S(A(\Pi))$$

We call such permutations, a *"good permutation"*. In short, a *good permutation* of an array has minimum score over all possible permutations.

We make the *ansatz* that all the *good permutations* will be monotonic in nature, i.e. either $a_{\Pi_0(i)} < a_{\Pi_0(i+1)}$ for $1 \leq i < n$, or $a_{\Pi_0(i)} > a_{\Pi_0(i+1)}$ for $1 \leq i < n$. Since, all the numbers are distinct, there exist exactly two such permutations (one increasing and the other decreasing).

(b) Provide an algorithm which computes the minimum cost required to transform the given array A into a *good permutation*, $A(\Pi_0)$.
The cost of a transformation is defined as the sum of costs of each individual operation used in the transformation.
You will only be awarded full marks if your algorithm works correctly in $\mathcal{O}(n \log n)$ in the worst case, otherwise you will only be awarded partial marks, if at all.

`Motivations:`

- $\mathcal{O}(n \log n)$ hints either binary search or sorting
- Greediness for minimum cost also hints towards constructing array starting from the largest elements

---

We follow the below steps to transform the array into a *good permutation*:

▷ Since we want to transform the given array into a sorted one, we store the integers in a new data structure made of pairs of two integers (a_i, i).

▷ Now, we sort this data structure according to the first key in all the pairs, in both ascending and descending order.

▷ This gives us the relative positions of the integers and then we construct a new array correspondingly in the following way:
We create two arrays; with elements sorted in ascending and descending order, and iterate over them from one of the extreme ends (accordingly to construct increasing and decreasing array) and then start adding elements from the sorted array one by one in a decreasing order.

▷ We increment the cost by the current element(if it is not in the expected position) since any element we swap the current one with is essentially smaller. We also swap the indices of lesser element with the original index of larger element.

▷ Finally, we get the two permutations and all we are left is to compare the costs incurred in constructing those and choosing the cheaper one. (In some sense, we are characterizing how far is our given array far from a sorted one.)

`Pseudocode:`

function GOOD-PERM(A, n)
  $AA = [(,)]_n$
  for $i \to 0, n-1$ **do**
    $AA[i] \leftarrow (a_i, i)$
  end for
  $A_1 \leftarrow AA$.SORT(first) ▷ Sort by first key
  $A_2 \leftarrow AA$.SORT(first, $>$) ▷ Sort by first key, in descending order
  $asccost, descost \leftarrow 0, 0$
  for $i \to n-1, 0$ **do** ▷ Overall $\mathcal{O}(n\log(n))$
    $j \leftarrow A_1[i].second$
    if $j <> i$ **then**
      $asccost \leftarrow ascost + A_1[i].first$
      $xx \leftarrow$ BIN-SEARCH($A_1, AA[i]$) ▷ $\mathcal{O}(\log n)$
      $A_1[xx].second \leftarrow j$
    end if
  end for
  for $i \to 0, n-1$ **do** ▷ Overall $\mathcal{O}(n\log(n))$
    $j \leftarrow A_2[i].second$
    if $j <> i$ **then**
      $descost \leftarrow descost + A_2[i].first$
      $xx \leftarrow$ BIN-SEARCH($A_2, AA[i]$) ▷ $\mathcal{O}(\log n)$
      $A_2[xx].second \leftarrow j$
    end if
  end for
  return $\min(asccost, descost)$

(c) Bonus: Prove that your algorithm computes the minimum cost of converting any array A into a *good permutation*.

We apply greedy strategy and start constructing the sorted arrays by placing the maximum elements first and then in decreasing order. This ensures the cost, if incurred, is least as each maximum element is counted atmost once (only if a swap is necessary).
Also, if we consider WLOG, $a < b < c$ then (a, b, c) configurations gives $S = b - a + c - b = (c - a)$, while other configuration like (a, c, b) gives $S' = c - a + b - a > S$, also (b, a, c) gives $S'' = b - a + c - a > c - a$. Thus, monotonic nature is needed.

P4 Mandatory Batman Question

Batman gives you an undirected, unweighted, connected graph $G = (V, E)$ with $|V| = n$, $|E| = m$, and two vertices $s, t \in V$.
He wants to know $dist(s, t)$ given that the edge (u, v) is destroyed, for each edge $(u, v) \in E$. In other words, for each $(u, v) \in E$, he wants to know the distance between s and t in the graph $G' = (V', E')$, where $E' = E \backslash \{(u, v)\}$.

(a) Batman expects an algorithm that works in $\mathcal{O}(|V| \cdot (|V| + |E|) = \mathcal{O}(n \cdot (n + m))$.

`Motivations:`

- $\mathcal{O}(|V| + |E|)$ strongly suggests binary search
- $|V|$ factor suggests performing BFS for all vertices (very misleading), or... "almost all vertices"

Since, we want to know $dist(s, t)$ we perform a BFS starting from the vertex s. This will help us efficiently find the shortest paths from s to any other vertex in G. Now, there can be multiple paths (even shortest) from s to t. By BFS, we'd have atleast one such path. We check if the edge (u, v) lies on this path. The dichotomy follows as:

1. If (u, v) doesn't lie on this path: Deletion of (u, v) does not affect this path in any way. Choose this. We are done.
2. If it lies: Then just delete it and redo BFS from vertex s (since now the graph has become G'). Then get a shortest path and its corresponding distance

Now, in the worst case, it may happen that all of the remaining vertices (except s and t) lie on the same shortest path from s to t. So we may need to delete the edges corresponding to these vertices $(n - 1)$ in total and redo the BFS as many times.
We get the shortest distance between s and t and the update the matrix $M[u, v]$ for all such edges (u, v).

`Pseudocode:`

function SAVE-GOTHAM(G, M, n, s, t)
  $distance, parent \leftarrow$BFS$(G, s, n)$
  $v \leftarrow t$
  $r \leftarrow distance[t]$
  while $v <> s$ **do**
    $tfam$.APPEND(v)
    $v \leftarrow parent[v]$
  end while
  $tfam$.APPEND(s)
  for $u \rightarrow 0, n-1$ **do**
    for $v \leftarrow G[u]; v <>$ NULL$; v \leftarrow v.next$ **do**
      $M[u, v] \leftarrow r$
    end for
  end for
  for $u \rightarrow 1, r-1$ **do**
    $G[tfam[r-u]]$.DELETE$(G[tfam[r-u+1]])$
    $G[tfam[r-u+1]]$.DELETE$(G[tfam[r-u]])$

```
        distance_new ← BFS(G, s)
        M[r − u, r − u + 1] ← distance_new[t]
        G[tfam[r − u]].ADD(G[tfam[r − u + 1]])
        G[tfam[r − u + 1]].ADD(G[tfam[r − u]])
    end for
```

```
function BFS(G, x, n)
    Queue Q
    parent ← []_n
    distance ← [−1]_n
    distance[x] ← 0
    Q.enqueue(x)
    while !Q.isEmpty() do
        v ← Q.dequeue()
        for i ← G[v]; i <> NULL; i ← i.next do
            if distance[i] ← −1 then
                Q.enqueue(i)
                distance[i] ← distance[v] + 1
                parent[i] ← v
            end if
        end for
    end while
    return distance, parent
```

(b) He also wants you to provide him with proof of runtime of your algorithm, i.e., a Time-Complexity Analysis of the algorithm you provide.

`Time Complexity Analysis :`

```
function SAVE-GOTHAM(G, M, n, s, t)
    distance, parent ←BFS(G, s, n)                          ▷ O(|V| + |E|)
    v ← t
    r ← distance[t]
    while v <> s do                                          ▷ O(|V|)
        tfam.APPEND(v)
        v ← parent[v]
    end while
    tfam.APPEND(s)
    for u → 0, n − 1 do                                      ▷ O(|V| · |E|)
        for v ← G[u]; v <> NULL; v ← v.next do
            M[u, v] ← r
        end for
    end for
    for u → 1, r − 1 do
        G[tfam[r − u]].DELETE(G[tfam[r − u + 1]])            ▷ O(|V|)
        G[tfam[r − u + 1]].DELETE(G[tfam[r − u]])
        distance_new, p ← BFS(G, s)                          ▷ O(|V| + |E|)
        M[r − u, r − u + 1] ← distance_new[t]
        G[tfam[r − u]].ADD(G[tfam[r − u + 1]])
        G[tfam[r − u + 1]].ADD(G[tfam[r − u]])
    end for
```

BFS takes $\mathcal{O}(|V| + |E|)$ time and we do it maximum for $n - 1$ times (when all the $n - 2$ vertices other than s, t lie on the same path). Adding and deleting in adjacency list takes $\mathcal{O}(|V|)$ time. Thus, giving us total $T(n) = |V| \cdot [\mathcal{O}(|V| + |E|) + \mathcal{O}(|V|)] + \mathcal{O}(|V|) + \mathcal{O}(|V| \cdot |E|)$
$= \mathcal{O}(|V| \cdot (|V| + |E|))$

(c) Lastly, you also need to provide proof of correctness for your algorithm.

BFS processes the graph layer by layer with increasing distance from the target vertex, and processes each vertex exactly once. Thus, if a vertex is encountered at some distance d, it is the shortest distance, otherwise, it would've been visited earlier.
After the initial BFS from vertex s, We identify the vertices which do not lie on the shortest path and hence deleting any edges amongst those has no effect on $dist(s, t)$. Further, for each vertex on the shortest path, we delete the edge between it and the next element, and reperform BFS to get $dist(s, t)$ as required. This new shortest distance inherently satisfies the condition that edge between the two vertices is deleted. Thus, we are able to find $dist(s, t)$ for all the vertices.

P5 No Sugar in this Coat

You are given an **undirected**, **unweighted** and **connected** graph $G = (V, E)$, and a vertex $s \in V$, with $|V| = n, |E| = m$ and $n = 3k$ for some integer k. Let distance between u and v be denoted by $dist(u, v)$ (same definition as that in lectures).
G has the following property:

- Let $V_d \subseteq V$ be the set of vertices that are at a distance equal to d from s in G, then

$$\forall i \geq 0 : u \in V_i, v \in V_{i+1} \implies (u, v) \in E$$

Provide the following:

(a) An $\mathcal{O}(|V| + |E|)$ time algorithm to find a vertex $t \in V$, such that the following property holds for every vertex $u \in V$:

$$min(dist(u, s), dist(u, t)) \leq k$$

Note that your algorithm can report s as an answer if it satisfies the statement above.

`Motivations:`

- $\mathcal{O}(|V| + |E|) \Leftrightarrow BFS$

We can start with the BFS traversal of the given graph G from the vertex s and construct a BFS tree ass discussed in the lectures. We can choose any of the vertices in V_1 as t for s.
Now, the vertices in the levels $V_i, 1 \leq i \leq k$ are atmost at a distance of k from the vertex s (since there exists an edge between each pair of vertices in V_i and V_{i+1}).
Thus, for all such $u \in V_i, 1 \leq i \leq k$, the given condition is satisfied and we report s.
Now, dealing with the vertices $u \in V_i, (k + 1) \leq i \leq 3k$, we can choose a vertex t from either V_{2k} or V_{2k+1} since the maximum distance $dist(u, t) = k$ for the pairs $(V_i, V_j) := (i, j) = (k + 1, 2k)$ or $(2k, 3k - 1)$
If, there are no vertices at the $2k$ level, we can choose from $2k - 1$ or further up (if not even in $2k - 1$) until we can choose a vertex.

`Pseudocode:`

function GET-THERE-QUICKLY(G, n, s)
  $distance, V \leftarrow BFS(G, n, s)$
  $partner \leftarrow []_n$
  $k \leftarrow n/3$
  if $V[2k][0] \leftarrow$ NULL **then**
    for $i \leftarrow 2k; V[i] ==$ NULL$; i \leftarrow i - 1$ **do**
    end for
    $g \leftarrow i$
  end if
  $partner[s] \leftarrow V[1][0]$
  for $i \rightarrow 1, 3k - 1$ **do**
    for $j \leftarrow 0; V[i][j] <>$ NULL$; j \leftarrow i + 1$ **do**
      if $1 \leq i \leq k$ and $V[i][j] <>$ NULL **then**
        $partner[V[i][j]] \leftarrow s$
      else if $k + 1 \leq i < 3k$ and $V[i][j] <>$ NULL **then**
        $partner[V[i][j]] \leftarrow V[g][0]$

```
            end if
        end for
```

```
function BFS(G, x)
    Queue Q
    V ← [[]]_n
    distance ← [−1]_n
    distance[x] ← 0
    Q.enqueue(x)
    while !Q.isEmpty() do
        v ← Q.dequeue()
        for i ← G[v]; i <> NULL; i ← i.next do
            if distance[i] ← −1 then
                Q.enqueue(i)
                distance[i] ← distance[v] + 1
                V[distance[i]].append(i)
            end if
        end for
    end while
    return distance, V
```

`Time Complexity Analysis :`

In BFS, each vertex is processed exactly once and each edge is also processed exactly once giving us $\mathcal{O}(|V| + |E|)$ time complexity.
Finding the value of g is $\mathcal{O}(n)$ while the another loop surprisingly takes $\mathcal{O}(n)$ too, since we are just processing each vertex exactly once. Thus, giving us the total time complexity $T(n) = \mathcal{O}(|V| + |E|) + \mathcal{O}(|V|) = \mathcal{O}(|V| + |E|)$

(b) Proof of correctness for your algorithm.

`Proof of Correctness:`

BFS give us the shortest distance to any vertex (since if it were further shorter, it would have been visited earlier). We choose any vertex from V_1 for s. Now, every vertex will either lie in one of the levels V_i and for all those u lying in $V_i, 1 \leq i \leq k$, distance between u and $s, dist(u, s) = (i-0) \leq k$. Thus $t = s$. For rest of the vertices distance between u and $t, dist(u, t) = |g - i|$ where $g \leq 2k, k + 1 \leq i \leq 3k - 1 \implies (g - i) \in [1 - k, k - 1] \implies dist(u, t) \leq (k - 1) \leq k$.
Thus, we could find a vertex t for all the vertices.