COL351
Assignment 1

Mallika Prabhakar, 2019CS50440
Sayam Sethi, 2019CS10399

August 2021

Contents

1 Question 1 (MSTs)

Let G be an edge-weighted graph with n vertices and m edges satisfying the condition that all the edge weights in G are distinct.

1.a Unique MST

Question 1.a

Question. *Prove that G has a unique MST.*

Proof. We will prove this by induction on the size of G using an idea similar to Kruskal's algorithm discussed in the class.
Hypothesis:

$$h(n) : \forall\ G = (V, E) : \ |V| = n \implies MST(G) \ is\ unique \tag{1}$$

Base case: $n = 1$ is true since there is no edge and $MST(G) = (V, \phi)$ is unique.

Induction Step: Assume $h(n-1)$ is true for $n \geq 2$, now for $h(n)$:
(Note: This proof assumes each edge to be an unordered pair of vertices)

Consider Kruskal's algorithm,

Algorithm 1 Recursive MST Routine — Kruskal's algorithm

```
procedure MST(G)
    e_0 ← (x, y) be edge with least weight
    H ← G
    remove x, y from H and add new vertex z
    for all v such that v is neighbour of x or y do
        add (v, z) to H
        wt(v, z) ← min(wt(v, x), wt(v, y))
        if wt(v, x) < wt(v, y) then
            map(v, z) ← (v, x)
        else
            map(v, z) ← (v, y)
        end if
    end for
    T_H ← MST(H)
    T_G ← (V, {e_0})
    for all e ∈ T_H do
        if e is not incident on z then
            add e to T_G
        else
            add map(e) to T_G
        end if
    end for
    return T_G
end procedure
```

In the above algorithm, it is clear that H has $n-1$ vertices. Thus, by our assumption, $h(n-1)$ is true and hence T_H is unique. Also, we know that T_G is a valid MST, from the correctness of Kruskal's algorithm. Now, assume by contradiction that T_G is not unique. Then there exists an MST, say $T' \neq T_G$.

Claim 1.1. *e_0 cannot be in T'*

Proof. This is because, if e_0 were in T', then $T \setminus \{e_0\} \neq T' \setminus \{e_0\}$ and thus, there would be two different MSTs for H which would be a contradiction to our assumption. Thus, $e_0 \notin T'$. $\square$

Consider the path from x to y in T'. Since $e_0 = (x, y)$ is not present in T', there exists a different path, say $P = (f_1, f_2 \cdots, f_k)$ where $f_i \in E(T'), 1 \leq i \leq k$. We know that

$wt(f_i) > wt(e_0), 1 \leq i \leq k$.
Swap any of the f_i with e_0 and let the subgraph formed be T'', i.e., $T'' = T' \setminus \{f_i\} \cup \{e_0\}$.
We know T'' is a spanning tree of G since $V(T'') = V(G)$ and there are no cycles formed on performing the swap operation (this can be proven using contradiction as discussed in the lecture).
Now, consider the weight of T'':

$$\begin{aligned} & wt(T'') = wt(T') - wt(f_i) + wt(e_0) \\ \implies & wt(T'') < wt(T') \end{aligned} \tag{2}$$

We have shown that the total weight of T'' is lesser than the weight of T'. However, this is a contradiction to the fact that T' is the MST of G. Thus our assumption that T_G is not the unique MST of G was wrong. Therefore, $h(n)$ is true.

This completes the induction and the proof that *if all edge weights in a graph are distinct, then its MST is unique.* □

1.b Algorithm Sketch

Question 1.b

Question. *If it is given that G has at most $n+8$ edges, then design an algorithm that returns a MST of G in $O(n)$ running time.*

Solution. The idea is to use the previous result along with the fact that the number of edges to be removed to form a spanning tree is $m-(n-1)$ which is atmost $(n+8)-(n-1) = 9$, assuming that G was initially connected (else no MST exists). The algorithm is as follows:

Algorithm 2 Compute MST for 1.b

procedure MST(G)
  while $|E(G)| > |V(G)| - 1$ **do**
    $C \leftarrow findCycle(G)$
    $e \leftarrow$ edge with largest weight in C
    remove e from G
  end while
  return G
end procedure

The procedure *findCycle* calls a DFS function on G which uses graph colouring and returns the first cycle it finds:

Algorithm 3 *findCycle*

```
1:  procedure FINDCYCLE(G)
2:      v ← any vertex of G
3:      colour ← map of vertices initialised to zero
4:      parent ← map of vertices initialised to null
5:      (u, v) ← DFS(G, v, colour, parent, null)
6:                  ▷ DFS returns the bottommost and topmost vertex of the cycle
7:      if DFS returned null then
8:          return null
9:      end if
10:     C ← empty array of edges
11:     add (u, v) to C
12:     while u ≠ v do
13:         add (u, parent(u)) to C
14:         u ← parent(u)
15:     end while
16:     return C
17: end procedure
```

The *DFS* function looks as follows:

Algorithm 4 Identify cycle using colouring and DFS

```
procedure DFS(G, v, colour, parent, p)
    parent(v) ← p
    colour(v) ← 1
    for all u such that u is neighbour of v in G do
        if colour(u) is 2 then              ▷ there is a forward edge from v to u
            return (u, v)
        else if colour(u) is 0 then         ▷ u is unvisited
            value ← dfs(G, u, colour, parent, v)
            if value is not null then       ▷ cycle found in subtree of u
                return value
            end if
        end if
    end for
    colour(v) ← 2
    return null                             ▷ no cycle found in subtree of v
end procedure
```

We will assume without proof that the *DFS* (Algorithm 4) function is correct and it takes $O(n+m)$ time since the algorithm is a standard one and has been discussed in the lecture.

Now consider the function *findCycle* (Algorithm 3), lines $3, 4$ take $O(n)$ time and line

5 takes $O(n+m)$ time. The while loop (lines 9–12) traverses up from u to v and each iteration takes $O(1)$ time. Therefore the entire while loop completes in $O(n)$ time (since the graph has n vertices and hence the loop cannot run for more than n iterations). Therefore the total time complexity of *findCycle* is $O(n+m)$.

We will now prove termination and compute complexity of Algorithm 2, which contains the code for computing MST:
Termination: The while loop terminates when $|E| = |V|-1$, that is, when the graph is a tree (since it assumes that the graph is connected). In each iteration of the while loop, *findCycle* returns a valid cycle since $|E| > |V| - 1$ and the graph is connected. After having found the cycle, we remove the edge with largest weight from G and therefore $|E|$ reduces by 1. Since $|V|$ remains constant, the while loop terminates after a finite number of steps.
Time Complexity: The while loop runs for exactly $m-(n-1)$ iterations, which is $O(m-(n-1)) = O((n+8)-(n-1)) = O(1)$ for the given constraints. Each iteration of the while loop calls *findCycle* which runs in $O(n+m) = O(n+(n+8)) = O(n)$. Finding the edge with least weight is $O(n)$ since a cycle cannot have more than n edges. Removing this edge from G can be implemented in as worse as $O(n)$ (better implementations in $O(1)$ and $O(\log(n))$ time exist but this won't change the complexity of the algorithm as we will show). Thus, each iteration of the while loop takes $O(n)$ time and the total time complexity of Algorithm 2 is:

$$\begin{aligned} T(MST) &= O(\text{iterations of while loop} \times \text{complexity of each iteration}) \\ &= O(O(1) \times O(n)) = O(n) \end{aligned} \tag{3}$$

Correctness: We now proceed to prove the correctness of the algorithm, using the following claim,

Claim 1.2. *If a cycle has edges of distinct weights, the edge with the largest weight can not be a part of any MST*

Proof. Let us assume by contradiction that the claim is false, then there exists an MST, say T such that the largest edge of cycle C (with distinct weights) is present in T. Let that edge be $e = (x, y)$. Now consider the paths from x to y in G. There exists atleast another path from x to y, which is exactly equal to $C \setminus \{e\}$, call it P. Consider the edge in P which is not in T, say $f = (p, q)$. We know such an edge exists since T is acyclic. Now, consider $T' = T \setminus \{e\} \cup \{f\}$. We will now prove that T' is a spanning tree using the following claim:

Claim 1.3. *Consider any edge $m = (a, b)$ in G which is not in T (spanning tree of G). Let $n = (u, v)$ be any edge on the unique path from a to b in T. Then on swapping m with n in T, we get another spanning tree of G.*

Proof. If $path_T(u, a)$ does not exist in T, then swap u, v (for ease of notation). Consider the graph $T \setminus \{n\}$. Define set

$$S = \{(c, d) \mid path_T(c, d) = \{k_1, k_2, \ldots, u, v, \ldots, k_l\}\} \tag{4}$$

All pair of vertices in this set are disconnected since all paths in the tree are unique. Now, consider the path

$$P_{T'} = \{c = k_1, k_2, \ldots, u\} \cup path_T(u, a) \cup path_T(b, v) \cup \{\ldots, d = k_l\}, \forall (c, d) \in S \quad (5)$$

Now, consider the graph $T' = T \setminus \{n\} \cup \{m\}$. All paths given by $P_{T'}$ are present in T' and thus, all pairs of vertices in S are connected in T'. Since all other paths are the same in T and T', T' is connected. Since $|E(T')| = |V(T')| - 1$, T' is a tree and also a spanning tree of G. This completes the proof of the claim. □

Thus, from Claim 1.3, we know that T' is an MST of G. Consider the weight of T':

$$\begin{aligned} & wt(T') = wt(T) - wt(e) + wt(f) \\ \implies & wt(T') < wt(T), \text{ since } wt(e) > wt(f) \end{aligned} \quad (6)$$

This is a contradiction to the fact that T is an MST of G. Therefore our assumption that Claim 1.2 is incorrect was wrong. This proves the correctness of Claim 1.2. □

Now consider Algorithm 2. In each iteration of the algorithm, we remove the largest edge of a cycle from G. Let the new graph obtained be G'. Therefore our algorithm transforms the problem from G to G'. We need to show that both graphs have the same MST.
From Claim 1.2, we know that e cannot be in any MST of G and from Question 1.a, we know that the MST of G will be unique. Therefore, the MST of G and $G' = G \setminus \{e\}$ will be the same. This completes the proof of correctness of Algorithm 2. □

2 Question 2 (Huffman Encoding)

2.a Optimal Huffman Encoding

Question 2.a

Question. *What is the optimal binary Huffman encoding for n letters whose frequencies are the first n Fibonacci numbers? What will be the encoding of the two letters with frequency 1, in the optimal binary Huffman encoding?*

Solution. We begin by observing the property of Fibonacci numbers:

$$\begin{aligned} & f_n = f_{n-1} + f_{n-2} \ \forall n \geq 3 \\ \text{and, } & f_1 = f_2 = 1 \end{aligned} \quad (7)$$

We are given an alphabet $A = (a_1, a_2, \ldots, a_n)$ such that it has a frequency vector $F = (f_1, f_2, \ldots, f_n)$.Before finding the encoding, consider the sum of first k Fibonacci numbers, call it s_k:

$$\begin{aligned} & s_k = f_1 + f_2 + \cdots + f_{n-2} + f_{n-1} + f_n \\ \implies & s_k = f_1 + f_2 + \cdots + f_{n-2} + f_{n+1} \\ \implies & s_k = s_{k-2} + f_{n+1} \\ \implies & s_k - s_{k-2} = f_{n+1} \end{aligned} \tag{8}$$

On performing telescopic summation over Equation 8 (for $k > 2$), we get the following:

$$\begin{aligned} & s_k - \cancel{s_{k-2}} = f_{k+1} \\ + \quad & s_{k-1} - \cancel{s_{k-3}} = f_k \\ + \quad & \cancel{s_{k-2}} - \cancel{s_{k-4}} = f_{k-1} \\ & \vdots \\ + \quad & \cancel{s_4} - s_2 = f_5 \\ + \quad & \cancel{s_3} - s_1 = f_4 \\ \implies \quad & s_k + s_{k-1} - s_2 - s_1 = s_{k+1} - f_3 - f_2 - f_1 \\ \implies & (s_k + 1) + (s_{k-1} + 1) = (s_{k+1} + 1) \end{aligned} \tag{9}$$

This Equation 9 takes a form similar to Equation 7 and thus, $s_k + 1 = f_m$ for some m. On substituting value of $k = 1$:

$$\begin{aligned} & s_1 + 1 = f_m \\ \implies & f_m = 2 \\ \implies & m = 3 \\ \implies & s_k + 1 = f_{k+2} \\ \implies & s_k = f_{k+2} - 1 \end{aligned} \tag{10}$$

Now consider the Huffman tree for $|A| = n$. Each of the frequency f_i $(1 \le i \le n-2)$ is less than f_n and sum of all frequencies f_i $(1 \le i \le n-2)$, i.e., $s_{k-2} = f_n - 1$ is less than f_n. We also know that a_i is merged at the same time or before a_j for any $i < j$. From this, we can formulate the merging strategy with the help of the following inductive claim:

Claim 2.1. *The optimal Huffman tree for A with frequency vector F is constructed in a way such that $(a_1, a_2, \ldots, a_i)$ is merged in the first $i-1$ steps $\forall i : 1 \le i \le n$.*

Proof. **Base case:** $i = 1$ is trivially true since a_1 is a leaf node and is *merged* in 0 merges.
Induction Step: Assume the claim is true for $i-1$. After $i-2$ merges, $(a_1, a_2, \ldots, a_{i-1})$ have been merged into $tree(a_1, a_2, \ldots, a_{i-1})$, and the frequency vector will be as follows,

$$\begin{aligned} & F = (f_1 + f_2 + \cdots + f_{i-1}, f_i, f_{i+1}, \ldots, f_n) \\ & F = (s_{i-1}, f_i, f_{i+1}, \ldots, f_n) \\ & F = (f_{i+1} - 1, f_i, f_{i+1}, \ldots, f_n), \text{ from Equation 10} \end{aligned} \tag{11}$$

It is easy to see that the least two frequencies in the frequency vector are $f_i, f_{i+1} - 1$ which correspond to a_i and $tree(a_1, a_2, \dots, a_{i-1})$. Therefore the $(i-1)^{\text{th}}$ merge will merge these two into $tree(a_1, a_2, \dots, a_i)$.
We have shown that a_i is merged in the $(i-1)^{\text{th}}$ step and from induction we know that $(a_1, a_2, \dots, a_{i-1})$ are merged before $(i-1)$ steps and thus, $(a_1, a_2, \dots, a_i)$ are merged in $i-1$ steps. This completes the induction and proves the claim. $\square$

Therefore, from Claim 2.1, we know that a_n is merged in the last step (which is the $(n-1)^{\text{th}}$ step) and hence it is encoded using a single bit. We can now inductively define the encoding for each alphabet (for $n > 1$):

Claim 2.2. *a_i is encoded as $\underbrace{11\dots1}_{n-i\ \textit{times}}0$ for $n \geq i > 1$ and a_1 is encoded as $\underbrace{11\dots1}_{n-1\ \textit{times}}$*

Proof. For $i > 1$, we will prove the claim using induction.
Base case: From Claim 2.1, we know that a_n will be merged in the last step and thus it is encoded using a single bit, we can choose this bit to be 0 and thus $enc(a_n) = \underbrace{11\dots1}_{n-n\ \text{times}}0 = 0$ and the claim is true for n.
Induction Step: Assume the claim is true for $i+1$, i.e., $enc(a_{i+1}) = \underbrace{11\dots1}_{n-(i+1)\ \text{times}}0$.
From the proof of the previous claim, we know that a_{i+1} and $tree(a_1, a_2, \dots, a_i)$ are siblings and thus, the encoding of the root of $tree(a_1, a_2, \dots, a_i)$ will be $\underbrace{11\dots1}_{n-i\ \text{times}}$.
From the base case, we know that a_n is encoded using a single bit with respect to the root of the tree. Therefore, with respect to $tree(a_1, a_2, \dots, a_i)$, we know that a_i is encoded using a single bit. Let that bit be 0. We then have the complete encoding of a_i as:

$$\begin{aligned} enc(a_i) &= enc(tree(a_1, a_2, \dots, a_i)).0 \quad (\text{. denotes concatenation}) \\ &= \underbrace{11\dots1}_{n-i\ \text{times}}0 \end{aligned} \tag{12}$$

This completes the induction for $i > 1$ and we now show the correctness of the claim for $i = 1$.
We know that a_1 and a_2 are siblings in the Huffman tree and thus they differ in their representation in exactly the last bit. Therefore, $enc(a_1) = \underbrace{11\dots1}_{n-1\ \text{times}}$. This completes the proof of the claim. $\square$

Thus, we have computed the optimal Huffman encoding for the alphabet $A = (a_1, a_2, \dots, a_n)$ which has frequency vector as $F = (f_1, f_2, \dots, f_n)$ and we restate Claim 2.2:
In the optimal Huffman encoding for A with frequency F such that $|A| = n$, *a_i is encoded as $\underbrace{11\dots1}_{n-i\ \textit{times}}0$ for $n \geq i > 1$ and a_1 is encoded as $\underbrace{11\dots1}_{n-1\ \textit{times}}$* (and for $n = 1$, $a_n = a_1 = 0$ trivially). $\square$

2.b Prove no Efficiency

Question 2.b

Question. *Suppose you aim to compress a file with 16-bit characters such that the maximum character frequency is strictly less than twice the minimum character frequency. Prove that the compression obtained by Huffman encoding, in this case, is same as that of the ordinary fixed-length encoding.*

Proof. Instead of proving the solution for a $16 - bit$ encoding, we will prove this for any $k - bit$ encoding with the help of the following claim:

Claim 2.3. *For an encoding of an alphabet A_k of size 2^k such that $f_1 < 2f_{2^k}$ where F is the frequency vector in increasing order, a_{2m-1} and a_{2m} are siblings for $1 \leq m \leq 2^{k-1}$ and the same inequality holds for the parents of a_i ($1 \leq i \leq 2^k$)*

Proof. Let $2^k = n$. Consider the frequency vector F, the least two frequencies are of a_1 and a_2. Let them be merged into b_1 where frequency of b_1 is $g_1 = f_1 + f_2$. It is easy to see that g_1 is greater than all f_i ($1 \leq i \leq n$) using the fact that $f_n < 2f_1 < g_1$.
We will now prove the following hypothesis using induction:

$$h(i) : b_p = parent(a_{2p-1}, a_{2p}), 1 \leq p \leq i, \text{ and } g_p < g_q, 1 \leq p < q \leq i \tag{13}$$

Base case: We have shown the base case, $h(1)$ to be true above.
Induction Step: Assume it is true for $i - 1$, and consider the frequency vector after these $i - 1$ merges in sorted order,

$$F = (f_{2i-1}, f_{2i}, \ldots, f_{n-1}, f_n, g_1, g_2, \ldots, g_{i-1}) \tag{14}$$

Thus, the least two frequencies are of a_{2i-1} and a_{2i} and these are merged into $b_i = parent(a_{2i-1}, a_{2i})$. Now,

$$g_i = f_{2i-1} + f_{2i}$$

we know that,

$$\begin{aligned} & f_{2i-1} > f_{2(i-1)-1}, \text{ and } f_{2i} > f_{2(i-1)} \\ \implies & f_{2i-1} + f_{2i} > f_{2(i-1)-1} + f_{2(i-1)} \\ \implies & g_i > g_{i-1} \end{aligned} \tag{15}$$

Therefore, $h(i)$ is also true and this completes the induction
Now, consider the statement of $h(n/2)$ ($n/2$ is an integer for $k > 0$):

$$b_p = (a_{2p-1}, a_{2p}), 1 \leq p \leq n/2 = 2^{k-1} \tag{16}$$

This completes the proof of the first half of the claim, now consider g_1 and $g_{n/2}$,

$$\begin{aligned} & 2a_1 > a_n, \text{ from the property of the alphabet} \\ \implies & 4a_1 > 2a_n \\ \implies & 2(a_1 + a_2) > a_n + a_{n-1} \\ \implies & 2g_1 > g_{n/2} \end{aligned} \tag{17}$$

Therefore, the property (inequality) also holds for the parents of A_k, i.e., for $A_{k-1} = (b_1, b_2, \ldots, b_{n/2})$. This completes the proof of the claim. □

Claim 2.4. *The depth of all nodes in the set A_i ($1 \leq i \leq k$) is equal to i (the set A_i is recursively defined as parent of A_{i+1} for $1 \leq i < k$)*

Proof. We will prove the claim using induction.
Base case: $i = 1$ is true since there is a single node and its depth is 0.
Induction Step: Assume the hypothesis is true for $i - 1$. Now, consider the children of each node in A_{i-1}. The depth of each child is 1 greater than the depth of its parent. From our assumption, we know that the depth of all nodes in A_{i-1} are equal to $i - 1$. Therefore, the depth of each node in A_i will be equal to i. This completes the induction and prove the claim. □

From Claim 2.4, we know that the depth of all nodes in A_k is equal to k and hence each letter in the alphabet will be encoded using k bits. This is the same as the fixed length encoding for an alphabet of size 2^k. Therefore, we have shown that the compression obtained by Huffman encoding is the same as the ordinary fixed-length encoding and this completes the proof. □

3 Question 3 (Party Management)

3.a Optimal Selection

Question 3.a

Question. *Present an efficient algorithm which outputs best choice of party invitees as per following specifications:*
1. Input: list of n people and list of pairs who know each other (undirectional)
2. Every person invited should have atleast 5 people they know and 5 people they don't know at the party

---

Solution. The main idea here is to obtain an adjacency list from the input and parse it. After creating an adjacency list, we keep on iterating over it until no new changes are being made. In every iteration of going over the adjacency list, we go over each vertex and see if the number of neighbours is less than 5 or greater than n-5-1. If either is the case, we "delete" the vertex and all of its edges else, continue. The algorithm is as follows:
add description of adj, people as well

Algorithm 5 Generate list of invitees for 3.a

```
procedure INVITE(G)
    initialise adj        ▷ empty hashmap of balanced BSTs created for adjacency list
    initialise people     ▷ empty balanced BST of degree,vertex pairs
    for all edge e {u, v} in E(G) do          ▷ sets adj
        add v to adj[u]
        add u to adj[v]
    end for
    for all vertex v in V(G) do               ▷ sets people
        add (degree(v), v) to people
    end for
    done ← false                              ▷ denotes if people is changing
    while not done do
        done ← true
        if people is empty then
            break
        end if
        i ← start(people)
        if degree(i) < k then
            done ← false
            deletePerson(vertex(i))
        end if
        i ← end(people)
        if size(people) − degree(i) − 1 < k then
            done ← false
            deletePerson(vertex(i))
        end if
    end while
    invitees ← []
    for all person in people do
        invitees.append(person.second)
    end for
    return invitees
end procedure
```

The procedure $deletePerson$ takes the vertex u into consideration for removal as a parameter and updates the adjacency list *adj* and *people*. Idea behind the algorithm is to look at all the neighbours of v and remove the edge $\{u, v\}$ from *adj* and update degree of v in *people*. If degree of v becomes zero, don't add it again.
Since people is a balanced BST, time complexity of removal and insertion is $O(log|V|)$ because $size(people) = |V|$. Also, adjacency list can be implemented as a hashmap of balanced BSTs, so locationg a vertex v takes $O(1)$ time and removing a vertex u from the hashmap[v] takes $O(log\ degree)$

Total time complexity of the procedure $deletePerson$ hence becomes

$$O(degree(u)) \times O(\log|V| + \log(degree(v))) + O(\log|V|) = O(degree(u) \times \log|V|) \tag{18}$$

Algorithm 6 sub-algorithm for $deletePerson$

```
procedure DELETEPERSON(u)
    for all vertex v in neighbours of u do
        remove {degree(v), v} from people
        remove u from adj(v)
        if degree(v) ≠ 0 then
            insert {degree(v), v} into people
        end if
    end for
    remove {degree(u), u} from people
    return
end procedure
```

Initialisation

Maintenance

Termination

Validate best solution

□

3.b Optimal Seating

Question 3.b

Question. *Suppose finally Alice invited n_0 out of her n friends to the party. Her next task is to set a minimum number of dinner tables for her friends under the constraint that each table has a capacity of ten people and the age difference between members of each dining group should be at most ten years. Present a greedy algorithm to solve this problem in $O(n_0)$ time assuming the age of each person is an integer in the range* $[10, 99]$.

Solution. We have to implement an $O(n_0)$ greedy approach for the fore-mentioned question. Idea behind our algorithm is to create a frequency array which stores how many people of a certain age are there in the invited group from age 10 to 100. We then start grouping 10 people together from the smallest to largest age. If the difference between youngest and oldest person exceeds 10, we stop the group and create a new table. Following is the algorithm for the same:

Algorithm 7 Find number of tables needed

```
procedure SEAT(people)                          ▷ Finds the minimum number of tables needed
    arr[100] ← initialised as zero array        ▷ index 0 to 9 aren't used
    for person in people do                     ▷ get the frequency array
        arr[age(person)] ← arr[age(person)] + 1
    end for
    set index, smallest, tcount ← 0             ▷ helpful intermediate variables
    set toAdd ← 10
    while index < 100 do
        if smallest = 0 then                    ▷ smallest=0 signifies no person on a table yet
            smallest ← index
        end if
        if arr[index] = 0 then                  ▷ no person of that age
            index ← index + 1
        else if index > smallest + 10 then      ▷ large age gap
            tcount ← tcount + 1
            smallest ← 0
            toAdd ← 10
        else if arr[index] > toAdd then         ▷ table occupancy can reach atmost 10
            arr[index] ← arr[index] − toAdd
            tcount ← tcount + 1
            toAdd ← 10
            smallest ← 0
        else
            toAdd ← toAdd − arr[index]
            arr[index] ← 0
            index ← index + 1
        end if
    end while
    if toAdd ≠ 10 then                          ▷ take last non filled table into account
        tcount ← tcount + 1
    end if
    return tcount                               ▷ total number of tables required is returned
end procedure
```

Now we shall prove the correctness of the algorithm:

Initialisation

We generate a frequency array by declaring a zero array of size 100 and incrementing the value stored at an index by 1. The update at an index is done based on what age value the for loop comes across. Updated index is same as age.
index variable is used to refer the current index in the while loop. *toAdd* variable shows how many more people are to be added at the recent table. *smallest* denotes the smallest value of age of people sitting at a table. $smallest = 0$ means that the table is empty. *tcount* denotes the table number. All of the forementioned variables are initialsed to zero except toAdd which is initialised at 10.
A while loop is run. The termination condition depends on the *index* variable. The loop keeps incrementing index until it finds the index at which arr[index] is not 0.

Maintenance

For maintenance, we look at the iterations of the while loop.
smallest = 0:
$smallest = 0$ implies that the table is new and there are no people seated on it.

The following cases arise in the if-else block:
Case 1: arr[index] is 0
Here, we move to the next index since there are no people of this age invited to the party.
Case 2: age gap between youngest person at table and index is > 10
Here, we cannot add the people at this index to the table since it violates our condition of the maximum age gap at table to be 10. We increase *tcount* by 1 and reset values of *smallest* and *toAdd* to 0 and 10 respectively.
Case 3: number of people with age same as index are more than *toAdd*
In this case, the current table can be filled completely. We subtract the number of people to be added from arr[index] and increment *tcount* by 1. *toAdd* and *smallest* are reinitialised.
Case 4: number of people with age same as index are $\leq toAdd$
Number of people of age same as index is not enough to fill the current table. We still allot them the table and move to next index. *toAdd* is decremented by $arr[index]$. $arr[index]$ is set to 0.
Since last group of statements is an else and no other conditions remain, our maintenance step is complete.

Termination and Time Complexity

In each iteration of the while loop, the value of *index* increases by 1 or the total sum of *arr* decreases by some non-zero number or *tcount* increments by 1. Thus, the maximum number of iterations of the loop is bounded by:

$$\begin{aligned} 100 + sum(arr) + \max(tables) &= 100 + O(n_0) + O(n_0) \\ &= O(n_0) \end{aligned} \tag{19}$$

Therefore, the while loop runs for $O(n_0)$ iterations and each iteration takes $O(1)$ time since it involved making finite checks and updates all of which take $O(1)$ time.
The loop from lines 3–5 takes $O(n_0)$ time since it involves iterating over the entire array of size n_0 once.
Therefore, the total time complexity of the algorithm is $O(n_0)$.

Proof of Correctness

To prove the correctness of the algorithm, we first prove the following claim:

Claim 3.1. *If there exists an optimal seating where p_i, p_k are on one table and p_j is on another table such that $age(p_i) < age(p_j) < age(p_k)$, there exists another optimal seating where p_i, p_j are on one table and p_k is on another table.*

Proof. Since p_i and p_k are on the same table, we know that $age(p_k) \leq age(p_{l1}) + 10$, where p_{l1} is the person with least age on the table. Consider the person with smallest age on the table containing p_j, say p_{l2}. We know that $age(p_j) \leq age(p_{l2}) + 10$. Therefore, we have that:

$$age(p_k) \leq age(p_{l2}) + 10, \text{ and } age(p_j) \leq age(p_{l1}) + 10 \tag{20}$$

Therefore, it is possible to swap the positions of p_k and p_j. Thus, we have obtained an arrangement where p_i, p_j are on the same table and p_k is on a different table. This completes the proof of the claim. $\square$

Consider the person with the smallest age, say p_0. Let the set of people be P and the set of people arranged on the same table as p_0 using our algorithm be A_0. Let the optimal number of tables needed for a set of people be denoted by $opt(.)$. Now, we will show that,

Claim 3.2. $opt(P) = opt(P \setminus A_0) + 1$

Proof. We first note that from Claim 3.1, it is possible to generate a seating where no more swaps are possible. The set A_0 which is generated by our algorithm is one such arrangement for a single table. Thus, there exists an optimal arrangement with A_0 as one table. We will now prove the claim in two parts:

1. $opt(P) \leq opt(P \setminus A_0) + 1$
   An arrangement such that one table consists of people in A_0 and the remaining are arranged optimally will have the number of tables equal $opt(P \setminus A_0) + 1$. This is by definition $\geq$ the most optimal seating for the set P and thus this equality trivially holds.

2. $opt(P \setminus A_0) \leq opt(P) - 1$
   We know that there exists an optimal arrangement which contains A_0 as one table. Therefore, the optimal arrangement for the remaining set of people, i.e., $P \setminus A_0$ can be done using atleast $opt(P) - 1$ tables. Thus, we get that $opt(P \setminus A_0) \leq opt(P) - 1$ which proves the second part of the claim.

From 1 and 2, we get that $opt(P) = opt(P \setminus A_0) + 1$, and this completes the proof of the claim. $\square$

Therefore, from Claim 3.2, we have shown that our algorithm *greedily* selects the most optimal seating. This completes the proof of correctness of our algorithm. $\square$