Due Monday 10^{th} of July at 6pm Sydney time (week 7)

In this assignment we review some basic algorithms and data structures, and we apply the greedy method. There are *three problems* each worth 20 marks, for a total of 60 marks. Partial credit will be awarded for progress towards a solution. We'll award one mark for a response of "one sympathy mark please" for a whole question, but not for parts of a question.

Any requests for clarification of the assignment questions should be submitted using the Ed forum. We will maintain a FAQ thread for this assignment.

For each question requiring you to design an algorithm, you *must* justify the correctness of your algorithm. If a time bound is specified in the question, you also *must* argue that your algorithm meets this time bound. The required time bound always applies to the *worst case* unless otherwise specified.

You must submit your response to each question as a separate PDF document on Moodle. You can submit as many times as you like. Only the last submission will be marked.

Your solutions must be typed, *not* handwritten. We recommend that you use LaTeX, since:

- as a UNSW student, you have a free Professional account on Overleaf, and
- we will release a LaTeX template for each assignment question.

Other typesetting systems that support mathematical notation (such as Microsoft Word) are also acceptable.

Your assignment submissions must be your own work.

- You may make reference to published course material (e.g. lecture slides, tutorial solutions) without providing a formal citation. The same applies to material from COMP2521/9024.
- You may make reference to either of the recommended textbooks with a citation in any format.
- You may reproduce general material from external sources in your own words, along with a citation in any format. 'General' here excludes material directly concerning the assignment question. For example, you can use material which gives more detail on certain properties of a data structure, but you cannot use material which directly answers the particular question asked in the assignment.
- You may discuss the assignment problems privately with other students. If you do so, you must acknowledge the other students by name and zID in a citation.
- However, you must write your submissions entirely by yourself.
  - Do not share your written work with anyone except COMP3121/9101 staff, and do not store it in a publicly accessible repository.
  - The only exception here is UNSW Smarthinking, which is the university's official writing support service.

Please review the UNSW policy on plagiarism. Academic misconduct carries severe penalties.

Please read the Frequently Asked Questions document, which contains extensive information about these assignments, including:

- how to get help with assignment problems, and what level of help course staff can give you
- extensions, Special Consideration and late submissions
- an overview of our marking procedures and marking guidelines
- how to appeal your mark, should you wish to do so.

Question 1

1.1 [12 marks] You are preparing to cross the Grand Canyon on a tightrope, and are building your balance pole out of steel rods. Naturally, you want to do this as cheaply as possible.

You have made a trip to Bunnings, and picked up n rods, the ith of which has length ℓ_i. You need to weld these all together to create your balance pole. Welding rods x and y costs $\ell_x + \ell_y$ dollars, and results in a new rod with length $\ell_x + \ell_y$.

For example, if we have rods with lengths $[2, 1, 3]$:

- After welding rods 2 and 3, costing \$4, we have rods with lengths $[2, 4]$
- We weld the remaining two rods, costing \$6
- The total cost to create the pole is \$10

Note that this is not necessarily the optimal sequence for these lengths.

Design an $O(n \log n)$ algorithm which finds the order you should weld the rods to minimize the total cost of welding.

This problem is solved using a method similar to the construction of the Huffman codes.

Algorithm: The core idea is to always take the two shortest rods and weld them together, producing a new rod, to minimise the number of times the larger rods are included in the cost. This can be done in several ways such as adding ℓ_i for $i = 1, \ldots, n$ to a min-heap Q, and repeatedly popping the head of Q twice with values ℓ_i, ℓ_j which will be the 2 rods being welded together. Then we push the value $\ell_i + \ell_j$ back onto Q and add $\ell_i + \ell_j$ to the total cost. We repeat this until Q has only one element.

Time complexity: Each step, we perform two pops and one push on Q, each of which has a time complexity of $\log n$. This reduces the size of Q by one each step, and we finish when the size is 1, so the algorithm runs for n steps, giving a time complexity of $O(n \log n)$.

Correctness: The proof of correctness follows very closely to the proof of correctness of Huffman coding. The key observation is that, for a particular rod x, the number of times it is used in the construction is *exactly* its depth in the construction of the heap (from here, we will just refer to it as a *tree*). Let T be the labelled binary tree constructed by our greedy algorithm and $d_T(i)$ denote the depth of rod i in the tree T.

To prove correctness via the exchange argument, we need to prove the following claim:

Claim. *Let x and y be two rods with the smallest lengths, then there exist an optimal ordering where x and y are used the same number of times.*

Proof. Without the loss of generality, we may assume that $\ell_x \leq \ell_y$.

Observe that this is exactly the same as claiming that x and y have the same depth in any tree representing an optimal ordering. Therefore, we will prove that a given optimal tree can be transformed into a tree where x and y have the same depth. To do this, observe that the cost of a tree T is exactly

$$C(T) = \sum_{i=1}^{n} \ell_i \cdot d_T(i). \tag{1}$$

Let T be any tree representing an optimal ordering, and let rods a and b be the two rods of maximum depth in T. Without the loss of generality, assume that $\ell_a \leq \ell_b$. Since x and y

have the smallest lengths, we have that

$$\ell_x \leq \ell_a, \quad \ell_y \leq \ell_b.$$

Now, if $\ell_x = \ell_b$, then necessarily, we have that $\ell_x = \ell_b = \ell_a = \ell_y$ and there is nothing to prove. Therefore, we may assume that $\ell_x \neq \ell_b$ and so, $x \neq b$. We will now exchange x with a to obtain a new tree T'. From (1), we see that

$$\begin{aligned} C(T) - C(T') &= \sum_{i=1}^{n} \ell_i \cdot d_T(i) - \sum_{i=1}^{n} \ell_i \cdot d_{T'}(i) \\ &= (\ell_a \cdot d_T(a) + \ell_x \cdot d_T(x)) - (\ell_a \cdot d_{T'}(a) + \ell_x \cdot d_{T'}(x)) \\ &= (\ell_a \cdot d_T(a) + \ell_x \cdot d_T(x)) - (\ell_a \cdot d_T(x) + \ell_x \cdot d_T(a)) \\ &= (d_T(a) - d_T(x))\,(\ell_a - \ell_x)\,. \end{aligned}$$

Now, in tree T, rod a has maximum depth; therefore, $d_T(a) \geq d_T(x)$ which implies that $d_T(a) - d_T(x) \geq 0$. Since $\ell_x \leq \ell_a$, we have that $\ell_a - \ell_x \geq 0$. In other words, we see that $C(T) - C(T') \geq 0$ or equivalently, $C(T) \geq C(T')$. Thus, if tree T represents any optimal ordering, then necessarily, $C(T) \leq C(T')$ for any tree T'. But this implies that $C(T) = C(T')$ and we deduce that the ordering represented by T' is optimal. We can also exchange y with b in T' to generate our greedy tree T'' and the argument follows. Thus, we have transformed any optimal tree T into T'' while maintaining optimality. Therefore, there exist an optimal ordering where x and y are used the same number of times. □

To conclude that our greedy strategy is optimal, we inductively apply the exchange argument.

Proof. Let $\mathcal{G} = (g_1, \ldots, g_m)$ denote the our greedy ordering of the n rods and let $\mathcal{O} = (o_1, \ldots, o_{m'})$ denote the ordering from any optimal solution. Additionally, let g_i denote the cost of our greedy welding after the first i welds and let o_i denote the cost of $\mathcal{O}$'s welding after the first i welds. Clearly, if $n = 1$, any ordering is trivially optimal; therefore, assume that $n \geq 2$. We begin the induction proof.

- **Base case**: Consider the smallest two rods. By the claim, the cost must be at most the cost of any optimal solution after the first weld; therefore, $g_1 \leq o_1$.
- **Inductive step**: Assume that, after the first k welds, $g_k \leq o_k$. Again, apply the claim. Welding the smallest two rods produces a tree whose cost C is minimal; in other words, for any other pair of rods to weld with cost C', we have that $C \leq C'$. Therefore, we have that $g_{k+1} = g_k + C \leq o_k + C' = o_{k+1}$, which proves the inductive step.

This proves that our greedy solution is correct and by the exchange argument, any solution that deviates from our greedy ordering is no better than our solution. □

1.2 **[8 marks]** Just before you step onto the wire, your pole snaps in half, and you realise going for the cheapest option might not have been the best idea. Fortunately, you brought spare rods and a welding device to the canyon.

As in part 1, you have n rods, the ith of which has length ℓ_i. It is given that n is odd.

Each rod in the pole contributes an *instability*, which is the product of the rod's length and the number of rods it is away from the middle. The order in which the welds are performed does not matter, only the order of the rods in the final pole.

That is, if you weld the n rods to create a pole $r_1, r_2, \ldots, r_n$, then the instability of the pole is

given by

$$\sum_{i=1}^{n} r_i \times \left| i - \frac{n+1}{2} \right|.$$

For example, if we create a pole $[2, 4, 6, 5, 2]$, then it has instability

$$\begin{aligned}
&= (2 \times |1-3|) + (4 \times |2-3|) + (6 \times |3-3|) + (5 \times |4-3|) + (2 \times |5-3|) \\
&= (2 \times 2) + (4 \times 1) + (6 \times 0) + (5 \times 1) + (2 \times 2) \\
&= 4 + 4 + 0 + 5 + 4 \\
&= 17.
\end{aligned}$$

Design an $O(n \log n)$ algorithm which finds the order of rods in the pole with minimal instability.

The main premise in this solution is to put the longest rods in the middle of the combined weld so they have minimal impact on the total instability of the combined rod.

Algorithm:

1. Sort the rods in descending order of length, using mergesort.
2. Take the longest rod, and attach the next two longest rods to the left and right of it.
3. Take the next two longest rods, and attach them to the left and right of the combined rods
4. Repeat step 3 until there are no more rods remaining

Time Complexity: Sorting the n rods with mergesort will take a total of $O(n \log n)$ time. Then, we go through the sorted rods and add them one by one. Adding each rod will take $O(1)$ time, for a total of $O(n)$ rods. So the total runtime of this algorithm is $O(n \log n) + O(n) = O(n \log n)$.

Correctness: First note that this is extremely similar to the tape storage problem in the lecture slides. This is because we are essentially given a set of multipliers to each of the rods lengths, and asked to minimise this - the multipliers in being $\left[0, 1, 1, 2, 2, ..., \frac{(n-1)}{2}, \frac{(n-1)}{2}\right]$. Our algorithm places the most expensive weights in the middle, i.e.: with lesser multipliers, so we have that our algorithm will produce a total instability of

$$(0 \times l_1) + (1 \times l_2) + (1 \times l_3) + ... + \left(\tfrac{(n-1)}{2} \times l_{n-1}\right) + \left(\tfrac{(n-1)}{2} \times l_n)\right),$$

where $l_1 \geq l_2 \geq ... \geq l_n$. In other words, we are allocating the greatest length rods with the least multipliers available.

First let us define an inversion in this context. An allocation δ has an inversion if it allocates a rod i of length l_i a multiplier of m_i and a rod j of length l_j a multiplier of m_j, where $m_i > m_j$ and $l_i > l_j$.

Claim. *Our greedy strategy is correct*

Now let us show that this allocation of lengths to multipliers is optimal. Consider any arbitrary solution δ that violates our greedy allocation by at least one pair of rods. For the first pair of rods that differ between our greedy solution G and δ, let the rods have denoted length l_i and l_j, where $l_i > l_j$, and G have allocated the multipliers m_1 to l_i and m_2 to l_i, where $m_1 > m_2$. Our greedy solution would have chosen to allocate the i the multiplier m_2, and the rod j the multiplier m_2. Since $m_1 > m_2$ and $l_i > l_j$, this clearly decreases the overall instability of the combined rods, and swapping any two arbitrary rods has no impact on the

instability of the others, as their multiplier is dependent on the number of rods it is away from the middle, not distance from the middle.

We simply continue to swap pairs of rods i and j that form an inversion, until there are no more inversions left, increasing optimally at each swap, until we eventually reach our greedy solution. Therefore our greedy schedule must be as optimal as any other allocation with no inversions, so our greedy solution is, indeed, optimal.

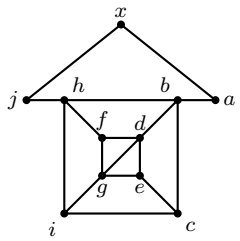
Question 2

Sam has recently acquired Friendbook, a large social media conglomerate, under dubious circumstances. Since their position as the company's CEO is only probational, they wish to curry favour with the employees by providing raises. However, they don't want to give raises to those who aren't well-known in the company (that would be a waste), and they can't give raises to those who are too well-known (that would raise suspicions). As such, they want to figure out the largest number of people in the company they can give a raise to, without wasting money, and without being caught.

Thankfully for Sam, the company keeps an absurd amount of data on its employees, so they have access to an undirected graph G of all n company employees and who they know within the company. In particular, this graph is provided as an **adjacency matrix** M such that $M[i, j] = 1$ if and only if employee i and j know each other. An **induced subgraph** H of G is a graph formed from a subset of the vertices of G, where pairs of vertices in H are adjacent if and only if the corresponding vertices are adjacent in G. In other words, an induced subgraph is formed by deleting some (or maybe none) of the vertices of G, and all adjacent edges.

Sam wishes to find the largest possible induced subgraph H of G such that every vertex in H is adjacent to at least k other vertices of H, but is also **not** adjacent to at least k other vertices of H. Such a subgraph is called "maximally nepotistic".

2.1 **[2 marks]** Consider the following graph G of company connections:



With $k = 3$, explain why any maximally nepotistic subgraph of G **cannot** include the vertex labelled x.

It is connected to only two vertices in G, so no matter what subgraph we choose including it, it cannot be connected to at least $k = 3$ vertices.

2.2 **[2 marks]** Using the graph G defined in 2.1, and again with $k = 3$, find a maximally nepotistic subgraph H of G, and explicitly list the vertices of H.

The set of vertices is $\{b, c, d, e, f, g, h, i\}$.

2.3 **[16 marks]** Given a graph G with n vertices and adjacency matrix M, and integer $k \geq 1$, design an algorithm which finds a maximally nepotistic subgraph of G, if it exists, and lists its vertices. If no such subgraph exists, the algorithm should not output anything. Your algorithm must run in time $O(n^3)$.

Consider deleting a vertex $v \in V$ and all of its incident edges. For each vertex u connected to v (i.e. $M[u, v] = 1$), the degree of u must necessarily decrease by 1. On the other hand, for each vertex w *not* connected to v (i.e. $M[w, v] = 0$), the degree remains unchanged. In either case, deleting a vertex does not increase the degree of any vertex and can only decrease by at most 1.

Algorithm: With this observation, we can start determining what vertices are *never* included in a *maximally nepotistic* subgraph. Clearly, any vertex that is either adjacent to fewer than k vertices in G or not adjacent to fewer than k vertices in G can never belong in such a subgraph; therefore, we can safely remove these vertices and the corresponding incident edges. We repeat this on the subgraph induced by the remaining vertices, deleting vertices that are either connected to fewer than k vertices or not connected to fewer than k vertices until we've removed all of the vertices of G or we've obtained an induced subgraph H such that no vertex can be deleted anymore. If such a subgraph exists, it must necessarily be *nepotistic*. We will prove that it is maximal.

Correctness: Let H be the nepotistic subgraph generated by our greedy algorithm, and let H' be a nepotistic subgraph such that H is a subgraph of H'. Consider the set of vertices in H' but not H. We shall prove that such a set must be empty. Suppose otherwise. Let v be such a vertex. Moreover, since the greedy algorithm deletes vertices from G, it must have arrived at graphs that contain H as a subgraph. In particular, the greedy algorithm must arrive at H' before H. Since H' is nepotistic, this implies that v is adjacent to at least k vertices in H' and not adjacent to at least K vertices in H'. This implies that v belongs in the subgraph that is returned by the algorithm; in other words, $v \in H$, which is a contradiction. Therefore, the set of vertices in H' but not H must be empty, which implies that $H' = H$. This shows that H is maximal, proving that our greedy algorithm returns a maximal nepotistic subgraph.

Time complexity: We now examine the running time of the algorithm. To find a vertex to delete, we traverse through each row and count the number of 0s and 1s in the adjacency matrix. This has running time $O(n)$ since we have to check at most n vertices. To ensure that we do not reconsider any deleted vertex i, we can remove row i and column i by either blocking out the row and column, or replacing the entries with a value that is different to 0 or 1. Therefore, finding all vertices to delete in one iteration takes $O(n^2)$ time. Since each iteration removes a subset of vertices, we have *at most* n iterations of the algorithm, giving us a running time of $O(n^3)$.

Question 3

3.1 **[10 marks]** Santa's assistant has been laid off and it is now your job to ensure that as many **wrapped** presents get delivered to shopping centres as possible.

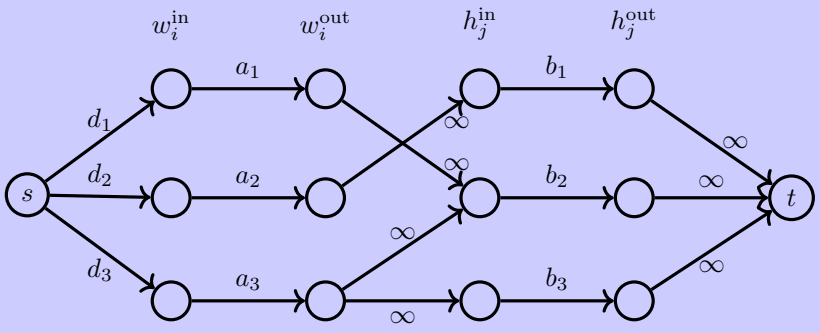
Santa has an infinite supply of **unwrapped** presents at the North Pole, and can distribute them to k workshops across the globe. At each workshop i, a maximum of a_i presents can be wrapped, and the North Pole can distribute to it at most d_i presents.

The workshops wrap the presents, and then deliver these to m shopping centres. Each shopping centre j can hold up to b_j packages, however, due to some legacy contractual agreements, each shopping centre can only receive presents from certain workshops. You are given the contractual agreements as a two dimensional array C, where $C[i][j]$ denotes that workshop i has an agreement with shopping centre j.

Design an algorithm that runs in $O((k+m)k^2m^2)$ time which finds the maximum number of wrapped presents that can be delivered to shopping centres.

Graph construction: We construct a flow network as follows.

- Construct a source vertex s that represents the North Pole.
- For each workshop i, construct two vertices, w_i^{in} and w_i^{out}.
- For each shopping centre j, construct two vertices, h_j^{in} and h_j^{out}.
- Construct a sink vertex t.
- For each workshop w_i, construct an edge from s to w_i^{in} with capacity d_i, representing the maximum amount of presents that the North Pole can send to workshop w_i.
- For each workshop i, construct an edge from w_i^{in} to w_i^{out} with capacity a_i, representing the maximum amount of presents that can be wrapped at workshop i.
- For each workshop i and shopping centre j, construct an edge from w_i^{out} to h_j^{in} of infinite capacity if and only if $C[i][j] = \mathsf{true}$.
- For each shopping centre j, construct an edge from h_j^{in} to h_j^{out} with capacity b_j, representing the maximum amount of presents that a shopping centre can hold.
- For each shopping centre j, construct an edge from h_j^{out} to t of infinite capacity.



A flow network with $k = 3$ workshops and $m = 3$ shopping centres.

Algorithm and solution: We now run Edmonds-Karp on the flow network. Each unit of flow corresponds to one present being delivered from the North Pole to the shopping centres. The flow is bottlenecked by the amount of presents in the workshops and shopping centres.

In particular, each capacity from w_i^{in} to w_i^{out} enforces at most a_i presents in workshop i, the capacity from h_j^{in} to h_j^{out} enforces at most b_j presents in shopping centre j. Therefore, the maximum number of presents delivered from the North Pole to the shopping centres is exactly the maximum flow.

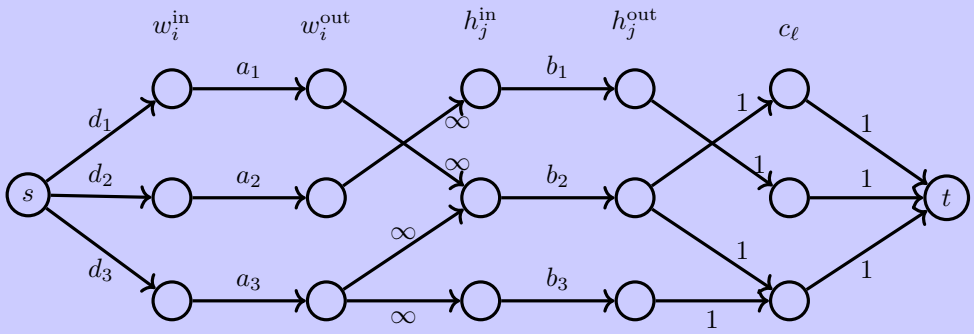
Time complexity: We now analyse the time complexity. To construct the graph, we construct $2(k+m+1) = O(k+m)$ vertices and at most $2(k+m) + km = O(km)$ edges. Therefore, the construction of the graph has running time $O(km)$. The running time of Edmonds-Karp is $O(|V| \cdot |E|^2) = O((k+m)(km)^2) = O((k+m)k^2m^2)$. Therefore, the overall time complexity is $O((k+m)k^2m^2)$.

3.2 [6 marks] On Christmas Eve, you realise that you should ensure that each of the n children on Santa's nice list receive at least one present. Children are willing to travel at most D distance to a shopping centre to receive a present. You are given the location of all children and shopping centres.

Design an algorithm that runs in $O(nm(k+n))$ time which finds the maximum number of children on the nice list who can receive at least one present.

Graph construction: We construct a flow network similar to **3.1** with a slight modification, as follows. Let $\mathsf{dist}(\ell, j)$ denote the distance from child ℓ to shopping centre j.

- Construct a source vertex s that represents the North Pole.
- For each workshop i, construct two vertices, w_i^{in} and w_i^{out}.
- For each shopping centre j, construct two vertices, h_j^{in} and h_j^{out}.
- For each child ℓ, construct a vertex c_ℓ.
- Construct a sink vertex t.
- For each workshop w_i, construct an edge from s to w_i^{in} with capacity d_i, representing the maximum amount of presents that the North Pole can send to workshop w_i.
- For each workshop i, construct an edge from w_i^{in} to w_i^{out} with capacity a_i, representing the maximum amount of presents that can be wrapped at workshop i.
- For each workshop i and shopping centre j, construct an edge from w_i^{out} to h_j^{in} of infinite capacity if and only if $C[i][j] = \mathsf{true}$.
- For each shopping centre j, construct an edge from h_j^{in} to h_j^{out} with capacity b_j, representing the maximum amount of presents that a shopping centre can hold.
- For each child ℓ and shopping centre j, construct an edge from h_j^{out} to c_ℓ with capacity 1 if and only if $\mathsf{dist}(\ell, j) \leq D$, representing that child ℓ is willing to travel to shopping centre j.
- For each child ℓ, construct an edge from c_ℓ to t with capacity 1.



A flow network with $k = 3$ workshops, $m = 3$ shopping centres, and $n = 3$ children.

Algorithm and solution: Again, we run Edmonds-Karp on the modified flow network. The constraints from **3.1** remains unchanged; therefore, instead of aggregating the number of presents, the sink now aggregates the number of children that receives at least one present. Thus, the number of children who receive a present is exactly given by $|f|$ for a given flow f. This implies that the maximum number of children who receive a present is *exactly* the maximum flow of the flow network.

Time complexity: We now analyse the time complexity. To construct the graph, we construct $2(k + m + 1) + n = O(k + m + n)$ vertices and at most $2k + km + m + nm + n = O(m(k + n))$ edges. Thus, the running time to construct the graph is $O(m(k + n))$.

Now, since the maximum flow is bounded by n (it is bounded by the number of edges to the sink), we have that Edmonds-Karp's running time is bounded by $O(|E| \cdot |f|) = O(m(k + n) \cdot n) = O(nm(k + n))$.

3.3 **[4 marks]** Let the farthest distance from any child to a shopping centre be F.

Design an algorithm that runs in $O(nm(k+n) \log F)$ time which finds the minimum **integer** value of D such that all n children on the nice list receive at least one present, or returns that no such D is possible.

Algorithm: For each distance $D \in \{1, \ldots, F\}$, perform the algorithm from **3.2** and check whether the maximum flow is equal to n.

- If the maximum flow is equal to n, then we conclude that it is possible for all n children on the nice list to receive at least one present. In this case, we recurse on the left subarray (including D) and check for distances smaller than D by performing a binary search.
- If the maximum flow is smaller than n, then we conclude that it is not possible for all n children to receive at least one present. In this case, we recurse on the right subarray and check for distances larger than D by performing a binary search.

The algorithm then returns the smallest distance such that the maximum flow is equal to n.

Correctness: We first prove that binary search is applicable in this setting. In **3.2**, we described an algorithm that finds the maximum number (i.e. the maximum flow) given a distance D. Let S denote the set of children who received a present given distance D and let $|f_D|$ denote the maximum flow with distance D.

- Since the dist function is monotone, increasing the distance to $D' \geq D$ increases the number of edges from $h_j^{\text{out}} \to c_k$. Since the same children from S remain unaffected

when we increase the distance to D', the maximum flow given distance D' must be *at least* the maximum flow given distance D (i.e. $|f_D| \leq |f_{D'}|$). However, since $|f_D| \leq n$ for all D, this implies that, if it was possible with distance D, it must also be possible with distance D'.

- We now prove the reverse. Suppose that it is not possible with distance D; that is, $|f_D| < n$. We copy the same reasoning from earlier and indeed, this implies that $|f_{D'}| \leq |f_D|$ for all $D' \leq D$. However, this implies that $|f_{D'}| < n$ and thus, it is also not possible for any distance smaller than D.

This shows that we have a monotonic behaviour. We now show that there are suitable lower and upper bounds. Since F is the farthest distance from any child, *every* child would be willing to travel with distance F. Therefore, it is unnecessary to check any distance greater than F and so, a suitable upper bound is F. A suitable lower bound is trivially 1. Thus, we have a suitable range to apply binary search, which proves that binary search is applicable.

We now prove the correctness of the algorithm. From the above discussion, the binary search procedure correctly traverses onto the appropriate subarray and so, finds the minimum value of D. The correctness of the subroutine comes directly from the correctness of **3.2**. Thus, the algorithm correctly returns the minimum value of D such that all n children on the nice list receives at least one present.

Time complexity: Since the binary search has a search space of size $O(F)$, our binary search performs $\log F$ iterations. In each iteration, we perform the algorithm from **3.2** which has running time $O(nm(k+n))$. Therefore, the running time of the algorithm is $O(nm(k+n)\log F)$.