SELECT ALGORITHM

1) Give a pseudo-code of Select algorithm.

```
function Select (arr, left, right, k)
  if left == right then
    return arr[left] // Base case: only one element

  Step 1: Find the pivot using the median of medians
      MedianOfMedians(arr, left, right)

  Step 2: Partition the array around the pivot
      Partition(arr, left, right, pivotIndex)

  Step 3: Determine the rank of the pivot
      if k == pivotIndex then
        return arr[k] // The pivot is the k-th smallest element
      else if k < pivotIndex then
        return Select(arr, left, pivotIndex - 1, k) // Search in the left part
      else
        return Select(arr, pivotIndex + 1, right, k) // Search in the right part
```

2) Derive it's time complexity for group of 5 elements.

- Group of 5, that means median of medians (mm) is less than at least **3** elements from half of the $\lceil n/5 \rceil$.
  It is greater than

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil\right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

  elements.
- Similarly, it is less than $\frac{3n}{10} - 6$ elemtns
- So we call the Select() recursively at most

$$n - \left(\frac{3n}{10} - 6\right) = \frac{7n}{10} + 6 \; elements$$

∴

$$T(n) = T\left(\left\lceil \frac{n}{5}\right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + O(n)$$
$$\leq C\left(\frac{n}{5} + 1\right) + C\left(\frac{7n}{10} + 6\right) + an$$
$$\leq \frac{Cn}{5} + C + \frac{7Cn}{10} + 6C + an$$
$$\leq \frac{9Cn}{10} + 7C + an$$
$$\leq Cn$$

n/5 + 7n/10 < 1, the worst-case running time is *O(n)*.

∴ Time Complexity O(n)

3) Derive it's time complexity for group of 7 elements.

- Group of 7, that means median of medians (mm) is less than at least **4** elements from half of the $\lceil n/7 \rceil$.
  It is greater than

$$4\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{7}\right\rceil\right\rceil - 2\right) \geq \frac{4n}{14} - 8$$

  elements.
- Similarly, it is less than $\frac{4n}{10} - 8$ elements
- So we call the Select() recursively at most

$$n - \left(\frac{4n}{14} - 8\right) = \frac{10n}{14} + 8 \quad elements$$

∴

n/7 + 10n/14 < 1, the worst-case running time is O(n).

$$\begin{aligned} T(n) &= T\left(\left\lceil \frac{n}{7}\right\rceil\right) + T\left(\frac{10n}{14} + 8\right) + O(n) \\ &\leq C\left(\frac{n}{7} + 1\right) + C\left(\frac{10n}{14} + 8\right) + an \\ &\leq \frac{Cn}{7} + C + \frac{5Cn}{7} + 8C + an \\ &\leq \frac{6Cn}{7} + 9C + an \\ &\leq Cn \end{aligned}$$

∴ Time Complexity O(n)

4) Prove or Disprove: Median select algorithm does not run in linear time when group size is 3

- If the input elements are divided into grp of 3 then X(median of medians) is grater than at least

$$2\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{3}\right\rceil\right\rceil - 2\right) \geq \frac{n}{3} - 4$$

- So we call the Select() recursively at most

$$n - \left(\frac{n}{3} - 4\right) = \frac{2n}{3} + 4 \quad elements$$

-

n/3 + 2n/3 = 1, so it reduces to *O(n log n)* in the worst case.

$$\begin{aligned} T(n) &= T\left(\left\lceil \frac{n}{3}\right\rceil\right) + T\left(\frac{2n}{3} + 4\right) + O(n) \\ &\leq C\left(\frac{n}{3} + 1\right) + C\left(\frac{2n}{3} + 4\right) + an \\ &\leq \frac{Cn}{3} + C + \frac{2Cn}{3} + 4C + an \\ &\leq \frac{3Cn}{3} + 5C + an \\ &\leq Cn + 5C + an \\ &> Cn \end{aligned}$$

Hence , Median Select Algorithm does not run in linear time when grp size is 3.

5) **Write the recurrence equation and solve it if select algorithm chooses a group of size 11**

- Group of 11, that means median of medians (mm) is less than at least **6** elements from half of the $\lceil n/11 \rceil$.
- It is greater than
  - $\mathbf{6}\left(\lceil \frac{1}{2}\left\lceil \frac{n}{11} \right\rceil \rceil - 2\right) \geq \frac{6n}{22} - 12$
- So we call the Select() recursively at most

$$n - \left(\frac{6n}{22} - 12\right) = \frac{8n}{11} + 12 \; elements$$

$$T(n) = T\left(\left\lceil \frac{n}{11} \right\rceil\right) + T\left(\frac{8n}{11} + 12\right) + O(n)$$

T(n): The time complexity for an array of size n.
$T\ (n/11\ )$: The time taken to find the median of the medians from the groups of size 11.
$T\ (9\ n/11)$: The time taken to recursively search in the larger subarray.
O(n): The linear time taken for partitioning the array.

6) Give a pseudo-code for randomized select algorithm.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2       return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k            // the pivot value is the answer
6       return A[q]
7  elseif i < k
8       return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

7) **Suppose we use randomized selection algorithm to select the minimum element of the array A ={3, 2, 9, 0, 7, 5, 4, 8, 6, 1}. Describe a sequence of partition that results in a worst case performance**

Worst-Case Scenario

The worst-case scenario occurs when the pivot chosen is consistently the largest or smallest element in the current subarray. This leads to highly unbalanced partitions where one side has no elements and the other side contains all remaining elements minus one.

Example Sequence of Partitions
Let's detail a sequence of partitions that would lead to a worst-case performance while trying to find the minimum element:

1. Initial Array:
   A={3,2,9,0,7,5,4,8,6,1}

2. **First Pivot Selection**: Suppose we randomly select **9** as the pivot.
    - **Partitioning**: All elements are less than or equal to 9.
    - **Resulting Partition**:
        - Left: {3,2,0,7,5,4,8,6,1}
        - Right: {}
    - The pivot index is now at position 99 (the last position).
3. **Second Pivot Selection**: Next, we randomly select **8** from {3,2,0,7,5,4,6,1}.
    - **Partitioning**: All elements are again less than or equal to 8.
    - **Resulting Partition**:
        - Left: {3,2,0,7,5,4,6,1}
        - Right: {}
    - The pivot index is now at position 88.
4. **Third Pivot Selection**: Now we select **7** from {3,2,0,5,4,6,1}
    - **Partitioning**: All elements are less than or equal to 7.
    - **Resulting Partition**:
        - Left: {3,2,0,5,4,6,1}
        - Right: {}
    - The pivot index is now at position 77.
5. **Continuing this Process**: We continue this process by selecting pivots in descending order (e.g., always selecting the maximum remaining element):
    - Select **6**, then partition into:
        - Left: {3,2,0,5,4,1}
        - Right: {}
    - Select **5**, then partition into:
        - Left: {3,2,0,4,1}
        - Right: {}
    - Select **4**, then partition into:
        - Left: {3,2,0,1}
        - Right: {}
    - Select **3**, then partition into:
        - Left: {2,0,1}
        - Right: {}
    - Select **2**, then partition into:
        - Left: {0,1}
        - Right: {}
    - Finally select **1**, which gives us:
        - Left: {0}{0}
        - Right: {}
        -

8) In randomized select algorithm for n elements, what is the probability that a worst case pivot is chosen for every call for select?

Worst-Case Scenario

In randomized select, the worst-case scenario occurs when the algorithm consistently selects either the largest or smallest element as the pivot at each recursive call. This leads to unbalanced partitions, causing the algorithm to perform poorly with a time complexity of $O(n^2)$.
Probability Calculation

1. **Events Definition**: Let E_k be the event that we pick the largest or smallest element when there are k*k* elements left in the array. The overall event E corresponds to the worst-case runtime occurring, which requires that every pivot selected in each recursive call is either the maximum or minimum element.

2. **Independent Choices**: Since each pivot selection is independent, we can express the probability of E as:

$$P(\mathcal{E}) = P\left(\bigcap_{i=1}^{n} \mathcal{E}_i\right) = \prod_{i=1}^{n} P(\mathcal{E}_i)$$

3. **Calculating Individual Probabilities**:
    - For $i=1$, $P(E1)=1$ because that single element is trivially both the largest and smallest.
    - For $i>1$, there are two favorable outcomes (the largest or smallest element) out of i total elements, so:

      $$P(E_i)=2/i$$

4. **Combining Probabilities**: Therefore, we can write:

   $$P(E)=P(E_1)\cdot P(E_2)\cdots P(E_n)=2^{n-1}\cdot 1/n!$$

---------END OF SELECT ALGORITHM ----------

Closest Pair Algorithm

1. i) Rewrite the divide and conquer closest pair algorithm such that in the combined step every point to the left of the vertical line L is compared with every point to the right of L.

```
Closest-Pair(P)
  Construct P_x and P_y  (O(n log n) time)
  (p_0*, p_1*) = Closest-Pair-Rec(P_x,P_y)

Closest-Pair-Rec(P_x, P_y)
  If |P| ≤ 3 then
    find closest pair by measuring all pairwise distances
  Endif

  Construct Q_x, Q_y, R_x, R_y (O(n) time)
  (q_0*,q_1*) = Closest-Pair-Rec(Q_x, Q_y)
  (r_0*,r_1*) = Closest-Pair-Rec(R_x, R_y)

  δ = min(d(q_0*,q_1*), d(r_0*,r_1*))
  x* = maximum x-coordinate of a point in set Q
  L = {(x,y) : x = x*}
  S = points in P within distance δ of L.

  Construct S_y (O(n) time)
  For each point s ∈ S_y, compute distance from s
     to each of next 15 points in S_y
     Let s, s' be pair achieving minimum of these distances
     (O(n) time)

  If d(s,s') < δ then
     Return (s,s')
  Else if d(q_0*,q_1*) < d(r_0*,r_1*) then
     Return (q_0*,q_1*)

  Else
     Return (r_0*,r_1*)
  Endif
```

ii) Express the time complexity

The time complexity of the closest pair algorithm we discussed is $O(n\log n)$.

Explanation:

1. **Sorting**: Initially, the points are sorted by their x-coordinates and y-coordinates, which takes $O(n\log n)$ time.
2. **Recursive Calls**: The algorithm divides the points into two halves and makes recursive calls. This leads to a recurrence relation of $T(n)=2T(n/2)+O(n)$.
3. **Combining Results**: After the recursive calls, the algorithm checks points within a certain distance in a strip, which can be done in linear time $O(n)$.

When solving this recurrence using the Master Theorem, it confirms that the overall time complexity is $O(n\log n)$.

2. In Randomized closed pair algorithm , how randomization is used ?

Random Shuffling:

- Initially, the set of points is randomly shuffled. This randomization helps in ensuring that the input data is in a random order, which helps in averaging out the running time of the algorithm across different inputs.

Insert Points into the Dictionary:

- For each point in the shuffled set, use the randomized hash function to determine the index in the hash table.
- Use the hash table to efficiently find and compare points to determine the closest pair.

3. Randomised closest pair Algo and Time complexity

Algorithm:

```
Order the points in a random sequence p_1, p_2, ..., p_n
Let δ denote the minimum distance found so far
Initialize δ = d(p_1, p_2)
Invoke MakeDictionary for storing subsquares of side length δ/2
For i = 1, 2, ..., n:
  Determine the subsquare S_st containing p_i
  Look up the 25 subsquares close to p_i
  Compute the distance from p_i to any points found in these subsquares
  If there is a point p_j (j < i) such that δ' = d(p_j, p_i) < δ then
    Delete the current dictionary
    Invoke MakeDictionary for storing subsquares of side length δ'/2
    For each of the points p_1, p_2, ..., p_i:
      Determine the subsquare of side length δ'/2 that contains it
      Insert this subsquare into the new dictionary
    Endfor
  Else
    Insert p_i into the current dictionary
  Endif
Endfor
```

Time complexity analysis:

1. **Random Order**: The points are ordered randomly, which helps in evenly distributing the points for processing.
2. **Initialization**: The minimum distance δ is initialized based on the distance between the first two points.
3. **Dictionary Operations**: A dictionary (or hash table) is used to store points in sub-squares of a grid defined by $\delta/2$.
4. **Lookup and Distance Computation**: For each point, the algorithm looks up nearby sub-squares and computes distances to points in those squares.
5. **Updating Closest Pair**: If a closer pair is found, the algorithm updates δ and reinitializes the dictionary.

Time Complexity Breakdown

1. **Initialization**: Sorting the points randomly takes $O(n)$ time.
2. **Distance Computations**: For each point, a constant number of distance computations is performed, leading to $O(n)$ distance calculations overall.
3. **Lookup Operations**: Each point requires a constant number of lookup operations in the dictionary, resulting in $O(n)$ lookup operations.
4. **Dictionary Operations**: The algorithm performs $O(n)$ operations for creating and updating dictionaries.

Total Time Complexity$= O(n)$

4. Define the closest pair problem

The closest pair problem is a fundamental problem in computational geometry. It involves finding two points in a given set of points that are closest to each other in terms of distance. Specifically, given n points in a metric space (commonly in the Euclidean plane), the goal is to identify the pair of points that have the smallest distance between them. This problem is significant in various applications, such as computer graphics, robotics, and geographical information systems, as it serves as a key step in many algorithms designed for spatial analysis. Efficient algorithms can solve this problem using methods like divide-and-conquer, achieving a time complexity of $O(n \log n)$

5. Define Universal hash function and explain how is it used in randomized closest pair algorithm

Universal Hash Function

A universal hash function is a type of hash function that is selected randomly from a family of hash functions. The key property of universal hashing is that it minimizes the probability of collisions between distinct keys. Specifically, for any two distinct keys x and y, the probability that they hash to the same value is at most 1/M, where M is the number of possible hash values. This property ensures that even if the input data is chosen in an adversarial manner, the expected number of collisions remains low, leading to efficient performance in hash table operations.

Use of Universal Hash Functions in the Randomized Closest Pair Algorithm

In the context of the **randomized closest pair algorithm**, universal hash functions are utilized to efficiently manage and store points in a dictionary.

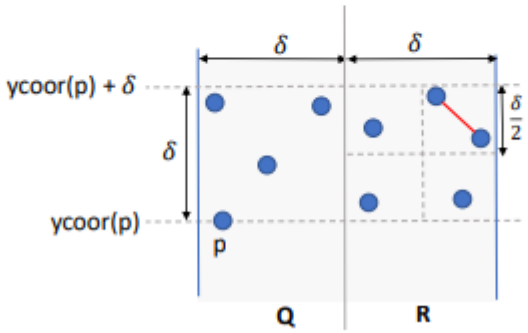
Here's how they are applied:

1. **Partitioning**: The algorithm divides the plane into sub-squares based on a defined size, Each point is mapped to a specific sub-square using a universal hash function.
2. **Collision Minimization**: By using a universal hash function, the algorithm reduces the likelihood of collisions when multiple points map to the same sub-square.
3. **Efficient Lookups**: When processing each point, the algorithm looks up nearby sub-squares (typically 25 surrounding sub-squares). The use of universal hashing allows for quick access to these points.

6.

Lemma 1. Fix any point $p \in S$. Then there are at most 8 points $q \in S$ such that $\text{ycoor}(p) \leq \text{ycoor}(q) < \text{ycoor}(p) + \delta$.

Proof. Suppose not. Suppose there are at least 9 such points. Concretely, define $S_p := \{q \in S : \text{ycoor}(p) \leq \text{ycoor}(q) < \text{ycoor}(q) + \delta\}$, and suppose for the sake of contradiction $|S_p| \geq 9$. Since these points of S_p either lie in Q or R, we are *guaranteed* at least 5 points in one side. Without loss of generality, suppose $|S_p \cap R| \geq 5$. See Figure 2 for an illustration where the points marked are the points in S_p.

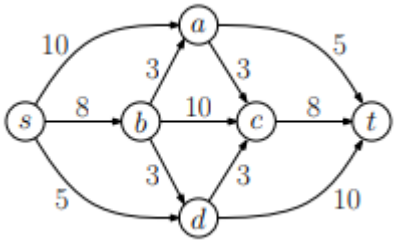


========== CLOSEST PAIR END ==========

NETWORK FLOW:

Flow Networks:

A flow network is a directed graph G = (V, E) in which each edge (u, v) ∈ E has a non-negative capacity **c**(u, v) ≥ 0. There are two special vertices: a source **s**, and a sink **t**.



Flows: Given an s-t network, a flow (also called an s-t flow) is a function f that maps each edge to a nonnegative real number and satisfies the following properties:

Capacity Constraint: For all (u, v) ∈ E, f(u, v) ≤ c(u, v).

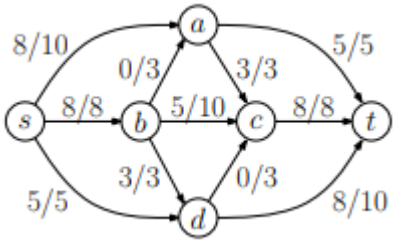
Flow conservation (or flow balance): For all v ∈ V - {s, t}, the sum of flow along edges into v equals the sum of flows along edges out of v.

$$f^{in}(v) = f^{out}(v), \text{ for all } v \in V - \{s, t\}.$$

1. State TRUE OR FALSE: Let G be an arbitrary flow network, with a source s, a sink t, and a integer capacity c. on every edge e. If f is a maximum s-t flow in G, then f saturates every edge out of s with flow (i.e., for all edges e out of s, we have f (e) = c_e).

In a flow network, having a maximum flow f*f* does not necessarily mean that every edge out of the source s*s* is saturated (i.e., has flow equal to its capacity). The maximum flow f*f* is defined as the largest amount of flow that can be sent from the source s*s* to the sink *t*, subject to the capacity constraints on the edges.

> **Saturation of Edges**: An edge is said to be saturated if the flow through that edge equals its capacity. While it is true that some edges leaving the source may be saturated in a maximum flow scenario, it is not required for all edges out of s*s* to be saturated. The flow can be distributed among multiple edges, and some edges may carry less than their full capacity.



2 .

Lemma: Let (X, Y) be any s-t cut in a network. Given any flow f, the value of f is equal to the net flow across the cut, that is, $f(X, Y) = |f|$.

Proof: Recall that there are no edges leading into s, and so we have $|f| = f^{\text{out}}(s) = f^{\text{out}}(s) - f^{\text{in}}(s)$. Since all the other nodes of X must satisfy flow conservation it follows that

$$|f| = \sum_{x \in X} (f^{\text{out}}(x) - f^{\text{in}}(x))$$

Now, observe that every edge (u, v) where both u and v are in X contributes one positive term and one negative term of value $f(u, v)$ to the above sum, and so all of these cancel out. The only terms that remain are the edges that either go from X to Y (which contribute positively) and those from Y to X (which contribute negatively). Thus, it follows that the value of the sum is exactly $f(X, Y)$, and therefore $|f| = f(X, Y)$.

3. Prove that minimum s-t cut is maximum flow in the graph

The **Max-Flow Min-Cut Theorem** states that in any flow network, the maximum flow from a source s to a sink t is equal to the capacity of the minimum cut that separates s and t.
In simpler terms, the minimum capacity cut acts as a bottleneck that limits how much flow can pass through the network.

When the Ford-Fulkerson method is used to find the maximum flow, it continues until it identifies this bottleneck. At this point, the algorithm has effectively found both the maximum flow and the minimum cut.

4 . Give example of a flow network where Ford Fulkerson algorithm takes the maximum time

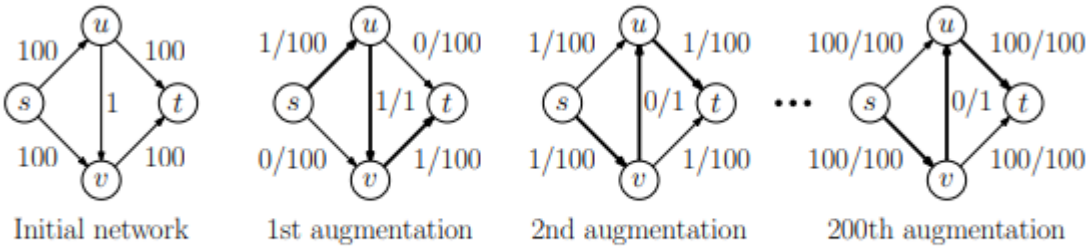
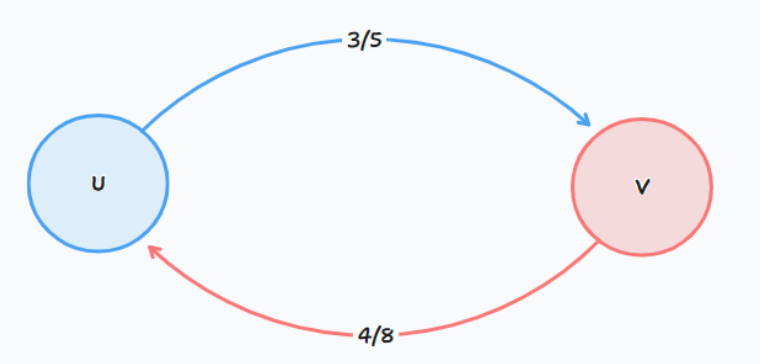


Fig. 56: Bad example for Ford-Fulkerson.

To see this, consider the example shown in Fig. 56. If the algorithm were smart enough to send flow along the topmost and bottommost paths, each of capacity 100, the algorithm would terminate in just two augmenting steps to a total flow of value 200. However, suppose instead that it foolishly augments first through the path going through the center edge. Then it would be limited to a bottleneck capacity of 1 unit. In the second augmentation, it could now route through the complementary path, this time undoing the flow on the center edge, and again with bottleneck capacity 1. Proceeding in this way, it will take 200 augmentations until we terminate with the final maximum flow.

5. Given vertices u and v in a flow network, where capacity c(u, v) = 5 and c(v, u) = 8, suppose that 3 units of flow are shipped from u to v and 4 units are shipped from v to u. Compute the net flow from u to v by giving a diagram.



The net flow from vertex u to vertex v is **-1**, indicating that there is effectively a surplus of flow moving back from v to u.

6.

After a devastative flood in a region, the food needs to be supplied to m different locations $R = \{r_1, r_2, r_3,...r_m\}$. The food distribution centers have been opened in a n number of places. Let the locations are represented by $L = \{l_1, l_2, l_3,...l_n\}$. The organizer knows how much supply of food can be passed through the road network N = (V, E), which are still accessible. Here V is the set of road junctions and E is the set of road segment between junctions. The amount of food supplies at every l_i, and the food requirement in every affected location r_j is also known. Design a flow network along with the algorithm to help the organizer to decide supply from which location should go to which affected location and whether the amount of food supplies is enough to satisfy the demands at all locations.

=>

Flow Network Design

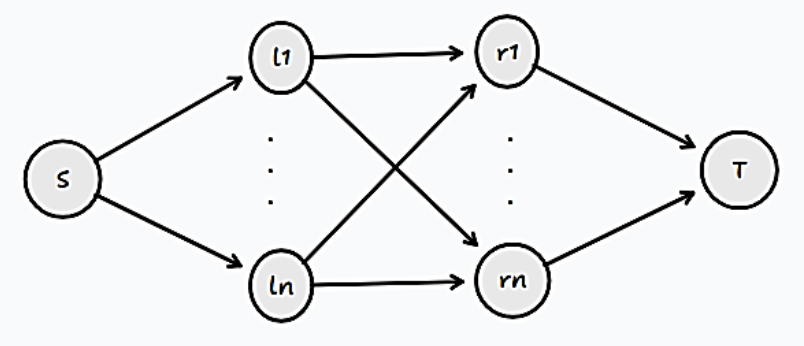
Vertices:

- Source (S): Represents the supply of food from distribution canters.
- Supply Locations (L): Each location l_i where food is available.
- Demand Locations (R): Each affected location r_j where food is needed.
- Sink (T): Represents the total demand being met.

Edges:

- From the source SS to each supply location l_i with a capacity equal to the amount of food available at that location.
- From each supply location l_i to each demand location r_j with a capacity that represents the maximum amount of food that can be transported between them (this could depend on road accessibility and distance).

- From each demand location r_j to the sink T with a capacity equal to the food requirement at that location.



Algorithm

To solve this flow network problem, we can use the Ford-Fulkerson algorithm using BFS for finding augmenting paths.

Steps:

I. Initialization:
   a. Create a flow network with vertices and edges as described above.
   b. Initialize all flows to zero.

II. Calculate Maximum Flow:
   a. While there exists an augmenting path from source SS to sink T T in the residual graph, increase the flow along this path until no more augmenting paths can be found.

III. Check Demand Satisfaction:
   a. After calculating the maximum flow, check if the total flow into each demand location equals its requirement.
   b. If all demands are met, then the supply is sufficient; otherwise, it is insufficient.

7. Let f and f' be two feasible (s, t)-flows in a flow network G, such that | f `| > | f |. Prove that there is a feasible (s. t)-flow with value | f ` | - | f | in the residual network G_f.

• **Flow Difference:** Let $\Delta f = f' - f$ be the flow difference between f' and f. Clearly, Δf is also a feasible flow in the network because both f and f ′ are feasible, and subtracting one feasible flow from another yields another feasible flow.

• **Residual Network:** The residual network G_f for a flow f is constructed by considering the residual capacities for each edge. The residual capacity on an edge (u , v) in the residual network is defined as:

$$C_f(u, v) = c(u, v) - f(u, v)$$

where c(u , v) is the original capacity of the edge (u , v) and f(u ,v) is the flow through that edge .

In the residual network G_f, there are two types of edges:

- Forward edges: If there is a flow f(u , v) the residual capacity is c(u , v) − f u , v).
- Backward edges: If there is a flow f(u , v), the residual capacity is f(u , v) on the reverse edge (v , u).

Feasibility of Δ f : To show that Δf is a feasible flow in the residual network G_f, we need to check the flow conservation and capacity constraints:

- **Capacity constraint:** For each edge (u , v), the flow Δf(u , v) must respect the residual capacity, meaning:

  $$0 \leq \Delta f(u , v) \leq C_f(u , v)$$

  Since f ′ and f are both feasible flows, it follows that the flow difference Δf = f ′ - f does not violate the capacity constraints.

  Flow conservation: For each vertex v, the flow conservation condition must hold for Δ f. This means that the total incoming flow to v should equal the total outgoing flow from v. Since f and f ′ are both feasible flows, their difference Δ f will also satisfy flow conservation at each vertex

====== Network Flow Ends here (do some more exercise) =========

Approximation

1. Define Approximation ratio for a maximization and minimization problem

Given an instance **I** of a problem, let **C(I)** be the cost of the solution obtained by the approximation algorithm and let **C`(I)** be the optimal solution.

+ For a minimization problem, Approximation ratio:

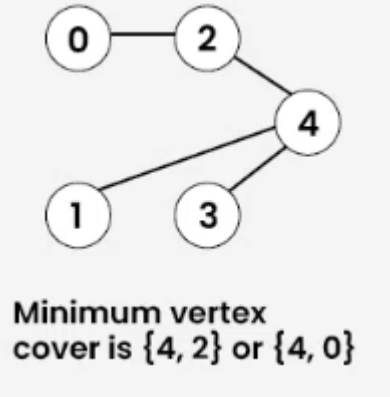
$$\frac{C(I)}{C`(I)} \geq 1$$

+ For a maximization problem, Approximation ratio:

$$\frac{C`(I)}{C(I)} \geq 1$$

2. State the vertex cover problem and comment on the complexity class of the problem

- Vertex cover is a subset of vertices such that every edge in the graph is incident to at least one of these vertices.
- Vertex Cover Problem is to find a vertex cover of minimum size.



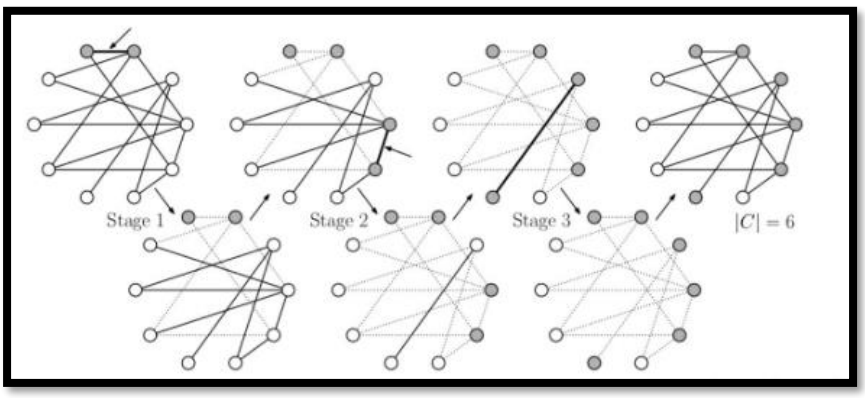
The vertex cover problem is an NP-Complete problem and it has been proven that the NP-Complete problem **cannot be solved in polynomial time.**

3. Give an approximation algorithm to solve the vertex cover problem and derive the approximation bound

(two algorithms are written)

- 2-for-1 heuristic algorithm

```
two-for-one-vc (Graph(V , E)){
        C = empty
        While (E is nonempty) do {
        Pick and edge (u , v) from E
        Add both u and v to C
        remove all edges that incident to either u or v
    }
    return C
}
```



- Approximation bound
  - → **C** is the output of two-for- one algorithm, let **C`** be the optimum vertex cover.
  - → Let A be the set of edges selected by the two-for-one algo
  - → As we have added both vertices (u , v) of each edge of A to C , we have |C| = 2 |A|
  - → However, the optimum vertex cover **C`** must contain at least one of (u , v) vertices. Therefore, |**C`**| ⩾ |A|

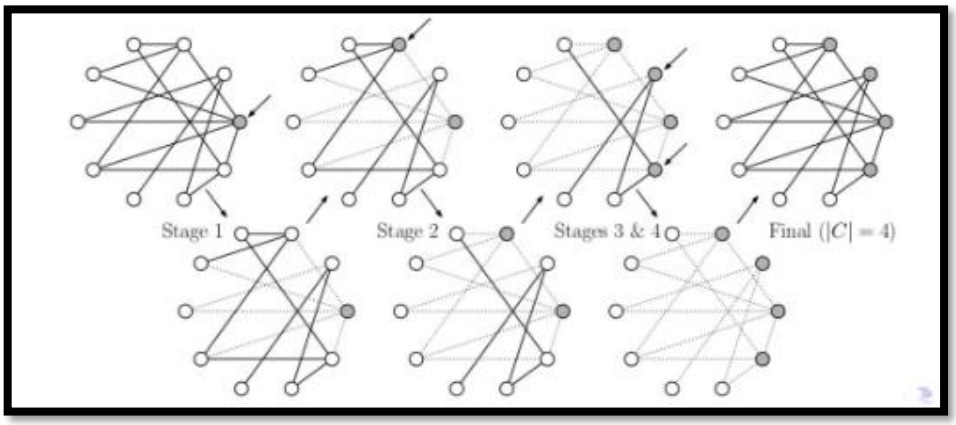
    ∴ |C| = 2|A|

    |C| ⩽ 2 |C`|

    $\frac{|C|}{|C`|} \leqslant 2$

    Hence 2-for-1 heuristic for VC achieves a performance ratio of 2.

- <u>Greedy heuristic</u>
  - greedy-VC (Graph(V , E)) {
    C = empty
    while (E is non empty) do {
    pick the vertex (let u) of maximum degree in G
    add u to C
    remove all edges from E that are incident to u
    }
    return C
    }
  - Though greedy heuristic for VC gave optimal result, it does not achieve a constant performance bound.
  - Studies shows greedy approach works better than two-for-one for typical graphs.



4. what is the significance of approximation ratio in approximation algo

In approximation algorithms, the "approximation ratio" is a crucial metric that indicates how close a solution produced by an algorithm is to the optimal solution, essentially measuring the "quality" of the approximation achieved, especially when dealing with complex problems where finding the exact optimal solution might be computationally impractical; a lower approximation ratio signifies a solution closer to the optimal one.

5. Define the bin packing problem.

=> Given a set of n items, where S_i denotes the size of the i^{th} item and $0 < S_i < 1$. Given infinite numbers of bins with capacity 1 , the problem is to pack those items into those bins such that number of bins required is minimum . It is a NP- complete problem , thus no polynomial time algorithm .

6. Give the first-fit approximation algorithm for Bin packing problem and Show that the first fir strategy to solve the bin packing problem gives an approximation factor of 2.

The **First-Fit Approximation Algorithm** for the bin packing problem is a straightforward and efficient method for packing items into a minimum number of bins. Here's how the algorithm works

→ **Input**: A list of items, each with a specific size, and a fixed bin capacity.

→ **Process**:

- Initialize unlimited number of empty bins.
- For each item in the list:
  - Place the item in the first bin that has enough remaining capacity to accommodate it.
  - If no such bin exists, create a new bin and place the item in this new bin.

→ **Output**: The total number of bins used and the distribution of items in these bins.

Proof:

b_{opt} = number of bins used in the optimal solution.

Let $S = \sum_i^n s_i$ denote the sum of n items .

Since no bin can hold more than 1 unit of item , we will need minimum of S bins to store S units of item. Thus $b_{opt} \geq S$

Now , Suppose that First-Fit uses m bins.

– Then, at least (m-1) bins are more than half full.

Because We never have two bins less than half full. If there are two bins less than half full, items in the second bin can be substituted into the first bin by First-Fit.

∴ For First Fit ,

$$b_{ff} = \sum_{i=1}^{bff} 1 \leq \sum_{i=1}^{bff} t_i + \sum_{i=1}^{bff} t_{i+1} = S + S = 2S \leq 2b_{opt}$$

Thus. $\frac{b_{ff}}{b_{opt}} \leq 2$

7. Define K – Center problem and Prove the Approximation bound for k-centre problem.

Formally, given:

- A set of vertices V in a metric space, where distances satisfy the triangle inequality.
- An integer k*k*, which represents the number of centers to be selected.

The goal is to find a subset **C ⊆ V** such that: **|C| ≤ k**

- The maximum distance from any vertex ***v ∈ V*** to its closest center in ***C*** is minimized.

This maximum distance is often referred to as the **covering radius** of the selected centers.

Proof

======== End of Approximation =========

String Matching

1 . **Compute the prefix function for the pattern ABBBACABB in a string-matching algorithm.**

The length of the pattern PPP is n = 9

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
P [i]	**A**	**B**	**B**	**B**	**A**	**C**	**A**	**B**	**B**
π [i]	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**2**	**3**

Final Prefix Function Table: π=[0,0,0,0,1,0,1,2,3]

2. Compute the prefix function for the pattern "ababaca" in a string-matching algorithm.

i	0	1	2	3	4	5	6
P[i]	a	b	a	b	a	c	a
π [i]	0	0	1	2	3	0	1

3. Define String Matching problem

The **String Matching Problem** involves finding occurrences of a specific pattern string **P** within a larger text string **T**. Formally, given a text **T[1..n]** of length n and a pattern **P[1..m]** of length m(**where m ≤ n),** the goal is to identify all valid shifts ***s*** such that the substring of T starting at position **s+1** matches the entire pattern **P .**

4 . Give the naive string-matching algorithm and find its time complexity.

```
NAÏVE-STRING-MATCHING (Text , Pattern)
    n = Text.length
    m = Pattern.length
    for s = 0 to n - m              // O (n-m +1)
        if P[1 .. m] == T [ s+1 … s+m]]          // O ( m)
            print "Pattern occurs with shift" s
```

Time Complexity : worst case - when every character in the text needs to be compared with every character in the pattern, particularly when there are many mismatches or when the pattern is not found.

$$O((n-m+1)m)$$

5. Suppose that the pattern P may contain occurrences of a gap character <> that can match an arbitrary string of characters (even one of 0 length). For example, the pattern ab<>ba<>c occurs in the text cabccbacbacab as cab<>ba<>c . The gap character may occur an arbitrary number of times in the pattern but not at all in the text. Give a polynomial-time algorithm to determine whether such a pattern P occurs in a given text T , and analyze the running time of your algorithm. (2022, 2023)

We know that one occurrence of P in T cannot overlap with another, so we don't need to double-check the way the naive algorithm does. If we find an occurrence of Pk in the text followed by a nonmatch, we can increment s by k instead of 1. It can be modified in the following way:

```
function matchWithGaps(T, P):
  n = length(T)
  m = length(P)
  s = 0  // Start position in text

  while s <= n - m do:
    // Check if the current character matches the first character of P
    if T[s] == P[0] then:
      k = s  // Record the starting position of the match
      i = 0   // Index for pattern

      // Match segments of P with T
      while i < m do:
        if P[i] == '<>' then:
          // Skip over gap; move to next character in T
          i += 1
          while k < n and T[k] != P[i]:
            k += 1
        else if k < n and T[k] == P[i] then:
          // Characters match; move to next character in both T and P
          k += 1
          i += 1
        else:
          break  // Mismatch found; exit inner loop

      // If we matched all characters in P
      if i == m then:
        print "Pattern occurs with shift", s

    s += 1  // Move to the next character in T
```

Time Complexity : The time complexity remains O(n) because each character in the text is processed linearly. The handling of gaps allows us to skip over portions of the text efficiently without unnecessary comparisons.

6 . Draw the state transition diagram for a string matching automata; pattern: ABABBABBABABBABABBABB.

States	A	B
0	1	0
1	1	2
2	3	0
3	1	4
4	3	5
5	6	0
6	1	7
7	3	8
8	9	0
9	1	10
10	11	0
11	1	12
12	3	13
13	14	0
14	1	15
15	16	8
16	1	17
17	3	18
18	19	0
19	1	20
20	3	21
21	9	0

7. **Give an algorithm to compute only the prefix function in KMP string matching along with time complexity.**

```
COMPUTE-PREFIX-FUNCTION(P, m)
1  let π[1 : m] be a new array
2  π[1] = 0
3  k = 0
4  for q = 2 to m
5      while k > 0 and P[k + 1] ≠ P[q]
6          k = π[k]
7      if P[k + 1] == P[q]
8          k = k + 1
9      π[q] = k
10 return π
```

Time Complexity : O(m) , where m is the length of the pattern.

String matching ends

Non-deterministic Polynomial

1. Define P and NP Class

P : the class of problems that have polynomial-time deterministic algorithms.

– That is, they are solvable in O(p(n)), where p(n) is a polynomial on n

– A deterministic algorithm is (essentially) one that always computes the correct answer

NP: the class of decision problems that are solvable in polynomial time on a nondeterministic machine (or with a nondeterministic algorithm)

- A nondeterministic computer is one that can "guess" the right answer or solution
- Thus NP can also be thought of as the class of problems "whose solutions can be verified in polynomial time"

NP-Hard: L is NP-hard if for all L` ∈ NP, L` ≤ P L . By transitivity of ≤ P , we can say that L is NP-hard if L` ≤P L for some known NP-hard problem L`.

NP-Complete: L is NP-complete if (1) L ∈ NP and (2) L is NP-hard.

The Boolean **satisfiability problem (SAT**) is a Boolean formula F, is it possible to assign truth values (0/1, true/false) to F's variables, so that it evaluates to true .

Vertex Cover (VC): A vertex cover in an undirected graph G = (V, E) is a subset of vertices V ' ⊆ V such that every edge in G has at least one endpoint in V ' . The vertex cover problem (VC) is: given an undirected graph G and an integer k, does G have a vertex cover of size k?

2. Prove that SAT is polynomial time reducible to Vertex cover problem.

Given an instance I of SATISFIABILITY, we transform it into an instance I' of VERTEX COVER.
Let I the formula f = $C_1 \wedge C_2 \wedge ... \wedge C_m$ with m clauses and n Boolean variables $x_1, x_2, ..., x_n$.
We construct I' as follows :
(1) For each clause C_j containing n_j literals, G contains a clique C_j of size n_j .
(2) For each Boolean variable x_i in f , G contains a pair of vertices x_i and x_i` joined by an edge.
(3) For vertex w in C_j , there is an edge connecting w to its corresponding literal in the vertex pairs (x_i , x_i`)
(4) Let

$$k = n + \sum_{j=1}^{m} (n_j - 1)$$

(not complete)

3. **Prove that SAT is polynomial time reducible to Clique problem**

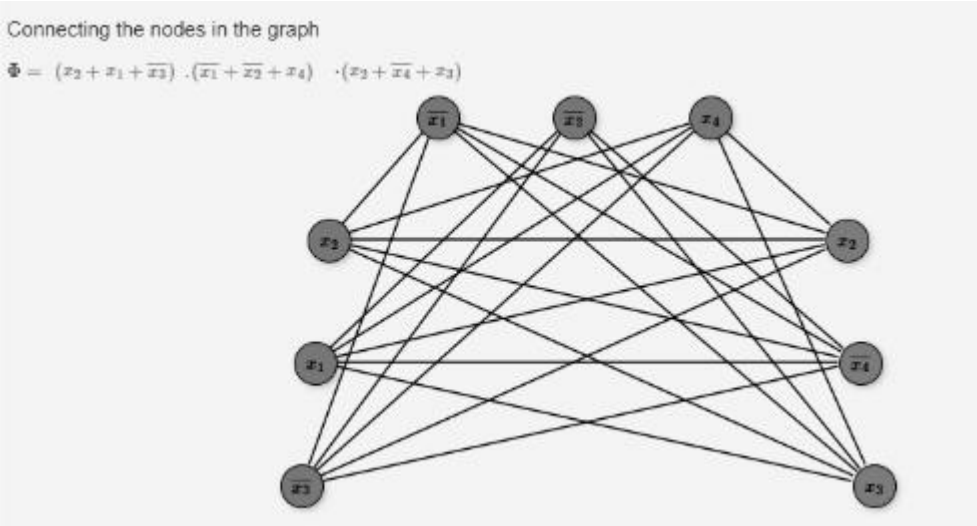
Given a Boolean formula in 3 CNF, the 3-SAT problem is to find whether the formula is satisfiable.
For a given graph G (V,E) and integer k, the CLIQUE problem is to find whether G contains a clique of size > k.

A formula Ø of k-literal clauses can be reduced to a k-Clique problem in the following way:

Construct a graph G of k clusters with a maximum of 3 nodes in each cluster.

Each cluster corresponds to a clause in Ø .

Each node in a cluster is labelled with a literal from the clause.
An edge is put between all pairs of nodes in different clusters except for pairs of the form (x , x`)
No edge is put between any pair of nodes in the same cluster



Insights about the graph

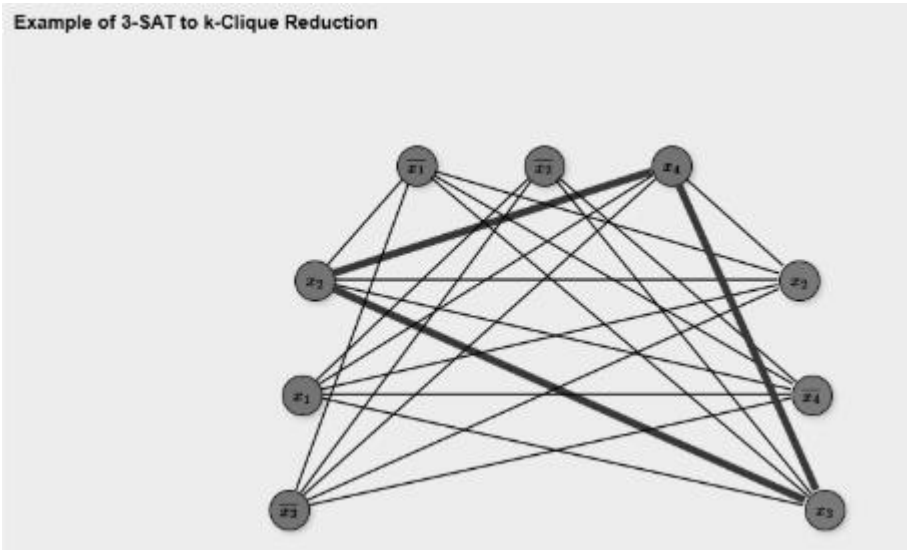
1. If two nodes in the graph are connected, the corresponding literals can simultaneously be assigned **True**. (This is true since there is no edge between nodes corresponding to literals of type x and x` .)
2. If two literals not from the same clause can be assigned **True** simultaneously, the nodes corresponding to these literals in the graph are connected.
3. Construction of the graph can be performed in polynomial time

3-SAT to k-Clique Reduction
G has a k-clique **if and only if** ∅ is satisfiable.

1. **If the graph G has a k-clique**, the clique has exactly one node from each cluster.
   (This is because no two nodes from the same cluster are connected to each other, hence they can never be a part of the same clique.)

All nodes in a clique are connected, hence all corresponding literals can be assigned **True** simultaneously.
Each literal belongs to exactly one of the k-clauses- Hence ∅ is satisfiable.

2. If ∅ is satisfiable, let A be a satisfying assignment. Select from each clause a literal that is **True** in A to construct a set S.
IISII = k. Since no two literals in S are from the same clause and all of them are simultaneously **True**, all the corresponding nodes in the graph are connected to each other, forming a k-clique. Hence, **the graph has a k-clique.**



DEFINE THE 3SAT And Vertex COVER PROBLEM

SKIP LIST