Greedy Algo Notes

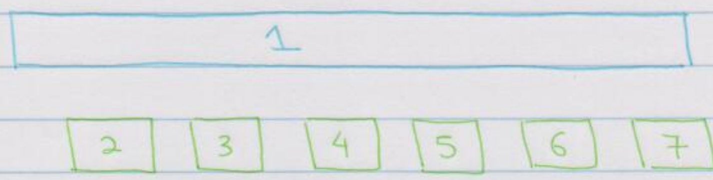
Table of Contents:

Interval Scheduling:

- We're given a list of jobs. Each job has a start time, s_i, and an end time, f_i.
  I.e. Job $i = [s_i, f_i)$
  2 jobs, Job i and Job j, are compatible if either $f_i \le s_j$ or $f_j \le s_i$.
  I.e. There's no overlap.
  Want to get the most num of compatible jobs.

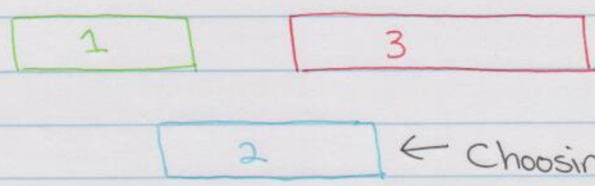
- There are a couple of ways we can choose the jobs:

1. By start time in ascending order.
   I.e. $s_1 \le s_2 \le \dots \le s_n$
   Problem: Doesn't work
   Counter Example:



   Job 1 blocks jobs 2-7.

2. By duration time in ascending order.
   I.e. Let $t_i = f_i - s_i$
   Then, $t_1 \le t_2 \le \dots \le t_n$
   Problem: Doesn't work
   Counter Example:

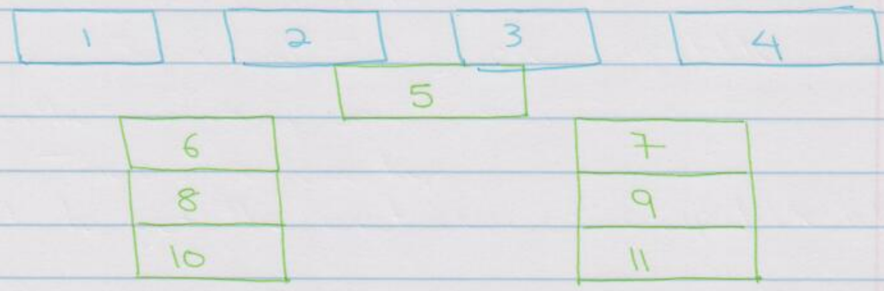


   ← Choosing job 2 blocks jobs 1 and 3.

3. By fewest conflicts

Problem: Doesn't work

Counter Example



Seeing as job 5 has the least amount of conflicts, 2, choose that first. Choosing job 5 blocks out 2 and 3.

Next, choose jobs 1 and 4. (Can choose other jobs)

With this strategy, we chose 3 jobs, but notice this isn't optimal as (1, 2, 3, 4) gives us 4 compatible jobs.

4. By earliest finish time

This strategy works.

Algo:

1. Sort the jobs by finish time in asc order. ← $O(n \lg n)$
2. keep track of the finish time of the last selected job. This way, comparing the end finish time to the start time(s) of future jobs takes $O(1)$.

Total Time Complexity: $O(n \lg n)$

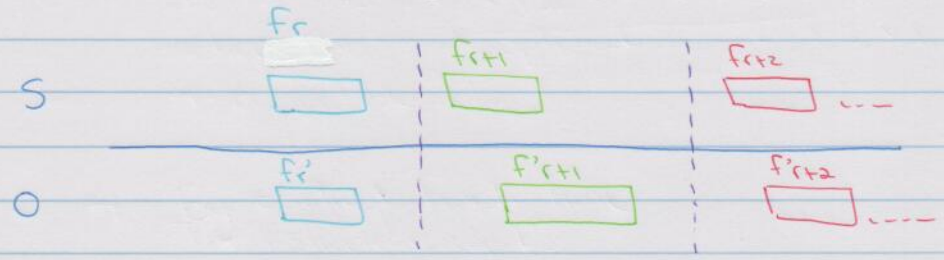
Proof of Optimality:

Let $S = \{f_1, f_2, \ldots, f_n\}$ be the greedy soln.
Let $O = \{f_1', f_2', \ldots, f_{n'}'\}$ be the opt soln.
Suppose that S is not opt.
I.e. $S \neq O$.

Let S and O match for as many indices as possible.
I.e. $f_1 = f_1', \ldots, f_r = f_r'$ for the biggest possible r.

Now, consider f_{r+1} and f'_{r+1}.
We know that $f_{r+1} < f'_{r+1}$ bc the greedy soln always chooses the job with the earliest finish time and $f_{r+1} \neq f'_{r+1}$.

Consider the diagram below:



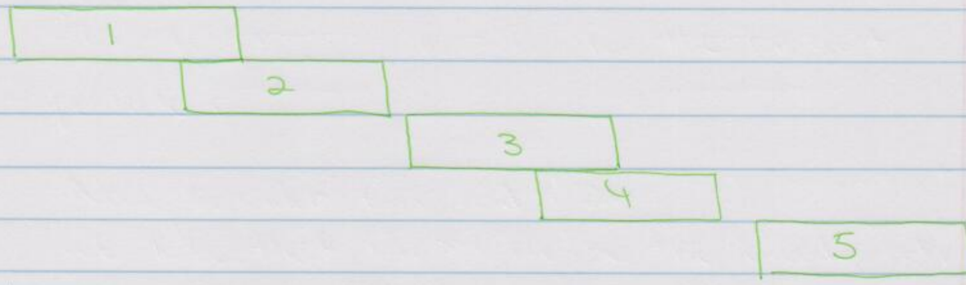
We can swap f_{r+1} and f'_{r+1} and O would still work.
Going down this path, we can switch f_{r+x} with f'_{r+x} and O would still be valid.

Hence, S is no worse than O.

Variations of Interval Scheduling Problem:

1. In this variation, suppose that instead of using the earliest finish time, we instead use the last activity to start that is compatible with all previously selected jobs.

   E.g.

   Here, we would choose jobs 5, 4, 2.

   This is still a greedy algo and it still provides an optimal soln.

   It provides an opt soln bc this is getting the earliest finish time if time flowed in reverse.

   It's a greedy algo because it gets the most opt soln at each point.

2. In this variation, we have m machines and n jobs. Each job can be run on any machine but it must be compatible with all prev jobs.

Time Complexity is $O(n\lg n)$

Greedy Algo:

- Use a min heap for the machines and have each machine keep track of its last finish time.

- Still select jobs by earliest finish time.

- For each job, compare its start time with the finish time of the machine at the top/beginning of the min heap. If the start time is less than the finish time, we know that the job isn't compatible with any machines that have jobs. Either add it to a new machine or if all of them are in use, discard it.

- If a job is compatible with exactly one machine, add that job to that machine, update the finish time for that machine and heapify.

- If a job is compatible with several (more than 1) machines, then add it to the machine with the biggest finish time to prevent blocking future jobs.

Finish times

E.g.

$M_1 \rightarrow 1$ Job $1 \rightarrow (2,3)$

$M_2 \rightarrow 2$ Job $2 \rightarrow (1,4)$

If Job 1 is added to M_1, Job 2 is blocked as it can't go on M_2. Hence, add Job 1 to M_2 and Job 2 to M_1.

Coin Change:

- In this problem, we're given a set of coin denominations and a value in cents. We want to use the least amount of coins to make the change amount.

- Greedy Algo:

```
def coin-change (change, denominations):   Assume this is sorted from greatest to least
    num-coins = 0

    while (change > 0):
        max-valid-deno  <-- Gets the biggest denomination no greater than change

        num-coins += change // max-valid-deno
        change %= max-valid-deno

    return num-coins
```

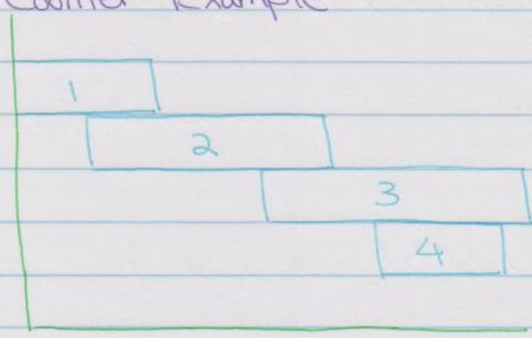
Note: This strategy can backfire.
E.g. change = 15, denos = [1, 7, 10]
By the greedy algo, you would choose the 10 then the 5 1's. But, this isn't opt as choosing (7, 7, 1) uses fewer coins

Interval Partition

- Similar to interval scheduling. This time, we're given a set of jobs with Job $i = [s_i, f_i)$ and we want to use the fewest possible machines to run them all. We can only put a job on a machine if its compatible with all other jobs on that machine.

- There are a couple of ways we can choose the jobs:

  1. By earliest finish time
     - Even though it worked for interval scheduling, it doesn't work here.
     - Counter Example



[0, 2)
[1, 4)
[3, 7)
[5, 6)

Using end-time first or earliest finish time, we'd first choose Job 1. Choosing Job 1 blocks Job 2, so we'd pick Job 4 next. Choosing Job 4 blocks Job 3.

Next, we'd choose Job 2, which blocks Job 3.

Lastly, we'd choose Job 3.

This uses 3 machines, which isn't opt

2. By Least Duration:
   - Doesn't work
   - Counter Example:
     Using the pic from the prev page, we'd choose Job 4 then Job 1. Jobs 2 and 3 are blocked.

     After, we'd choose Job 2, which blocks Job 3.

     Lastly, we'd choose Job 3.

     Uses 3 machines again.

3. By Fewest Conflicts:
   - Doesn't work
   - Counter Example:
     Using the pic from the prev page, we'd choose Job 1 then Job 4. Jobs 2 and 3 are blocked.

     We'd choose Job 2 next, which blocks Job 3.

     Lastly, we'd choose Job 3.

     Uses 3 machines.

4. By Earliest Start Time:
   - Works
   - Example:
     We'd choose Job 1, blocking Job 2, then Job 3, blocking Job 4.

     Next, we'd choose Job 2 and 4.
     Uses 2 machines

- Proof of Optimality:

Let the depth be the max number of jobs that overlap. Call this depth d.

Lemma 1: Any algo uses at least d machines.

Proof:

Suppose there are n jobs.
We know $d \leq n$. Suppose, wlog, that $d < n$.
In the worst case scenario, you use/have 1 machine for each job. $\rightarrow$ This uses n machines.
In the best case scenario you use d machines.

Put each of the d overlapping jobs on its own machine. We know that all other jobs must be compatible with at least 1 of these d jobs. Hence, we'll need d machines in total.

We've proved that any algo uses at least d machines. I'll prove now that our algo (Earliest Start Time) uses exactly d machines.

Proof:

We know from Lemma 1 that our algo uses at least d machines.
Furthermore, we know there are at most d overlapping jobs at any given time.
Machine d was opened bc job j isn't compatible with any of the prev d-1 machines. Each of those d-1 machines have an end time greater than s_j.
Hence, those are the d overlaps. Every other job can be placed on at least 1 of these d machines.
Hence, we need to use exactly d machines.

- Algo:

def Interval-Partition($J_1, J_2, \ldots, J_n$):

Sort the jobs by earliest start time

Use a min-heap for the machines

For each job:

1. Use binary search to see if its compatible with any existing machine. If it is, add it to that machine, update end/finish time and heapify.

2. Use a new machine. Update end time and heapify.

Time Complexity: $O(n \lg n)$

Minimizing Lateness:

- Given a single machine and n jobs s.t. each job j requires t_j units of time and is due by time d_j. If it's scheduled to start at s_j, it will finish at $f_j = s_j + t_j$. Its lateness is $\ell_j = \max\{0, f_j - d_j\}$. The goal is to minimize the max lateness, $L = \max_j \ell_j$.

- There are a couple of ways to choose the jobs:

  1. Shortest Processing Time First:
     I.e. Asc order of T_j.
     Doesn't work
     Counter Example:
     Job 1 = (1, 100)
     Job 2 = (10, 10)
     $\uparrow$ $\uparrow$
     T_j F_j

     In this case, we'd select Job 1 first then Job 2. This causes a lateness value of 1. However, if we chose Job 2 then Job 1, we'd get no lateness value.

  2. Smallest Slack First:
     I.e. Asc order of $d_j - t_j$
     Doesn't work.
     Counter Example:
     Job 1 = (99, 100) $f_t = 99, \ell_1 = 0$
     Job 2 = (1, 3) $f_t = 100, \ell_2 = 100 - 3 = 97$

     Consider the reverse:
     Job 1 = (1, 3) $f_t = 1, \ell_1 = 0$
     Job 2 = (99, 100) $f_t = 100, \ell_2 = 0$

3. Smallest Due Time:

I.e. Asc order of D_j

Works

- Proof of Optimality:

First, I'll define an inversion to be a pair of jobs (i, j) s.t. $d_i < d_j$ but j is scheduled before i.

Observations:

1. There exists an optimal soln with no idle time.
2. Our algo has no idle time.
3. Our algo has no inversions.
4. If a schedule with no idle time has at least one inversion, it has a pair of inverted jobs scheduled consecutively.

Proof:

Suppose jobs i and j are inverted, and no other pair of jobs are inverted and i and j are not consecutive.

I.e. $j, \ell, \ldots, i$

Consider job ℓ. If $d_\ell < d_j$, then jobs (j, ℓ) are inverted. If $d_\ell \geq d_j$, then jobs (ℓ, i) are inverted. This contradicts our assumption.

Hence, j and i must be consecutive.

5. Swapping inverted jobs doesn't increase lateness but reduces the num of inversions by 1.

Proof:

1. It's easy to see how swapping a pair of inverted jobs can decrease the num of inversions.

2. If i and j are inverted, then we know $d_i < d_j$ but j is chosen first.

When j is chosen before i

Let f_i, ℓ_i be the finish time and lateness of job i.
Let f_j, ℓ_j be the finish time and lateness of job j.

When i is chosen before j.

Let f_i', ℓ_i' be the finish time and lateness of job i after i and j switch.
Let f_j', ℓ_j' be the finish time and lateness of job j after i and j switch.

We know that for any job k, s.t. $k \neq i$ and $k \neq j$, that $\ell_k = \ell_k'$. This is bc only jobs i and j switched.

We also know that $\ell_i' \leq \ell_i$.

$$\begin{aligned} \ell_j' &= f_j' - d_j \\ &= f_i - d_j \quad \leftarrow \text{Bc i and j were switched} \\ &\leq f_i - d_i \quad \leftarrow \text{Bc by the def of inversion, } d_i < d_j \\ &= \ell_i \end{aligned}$$

Let $L = \max_k \ell_k$ and $L' = \max_k \ell'_k$

$$L' = \max\{\ell_i', \ell_j', \max_{k, k \neq i,j} \ell'_k\}$$

$$\leq \max\{\ell_i, \ell_i, \max_{k, k \neq i,j} \ell_k\}$$

Proved that $\ell_i' \leq \ell_i$

Proved that $\ell_j' \leq \ell_i$

Proved that $\ell_k = \ell'_k \ \forall k \neq i,j$

$$\leq L \leftarrow L = \max\{\ell_i, \ell_j, \max_{k \neq i,j} \ell_k\}$$

Combining these observations, esp 4 and 5, we will prove that our algo is opt.

Let $S = \{S_1, S_2, \ldots, S_n\}$ be the greedy algo soln.
Suppose it's not opt.
Let $O = \{O_1, O_2, \ldots, O_n\}$ be the opt soln.
Suppose that S and O match for as many jobs as possible.
I.e. $S_1 = O_1, \ldots, S_r = O_r$ for the biggest possible r.

Consider S_{r+1} and O_{r+1}. We know that $d_{S_{r+1}} < d_{O_{r+1}}$ bc the greedy algo always chooses by earliest due time.

Swapping S_{r+1} and O_{r+1} decreases the lateness of O. This is a contradiction.

$\therefore$ S is opt

Lossless Compression:

- Given a document that is written with n distinct letters, we want to losslessly compress it.

- Naive Solution:
  - Represent each letter using n bits.
  - If we have m letters, this will take $m\log(n)$ bits.
  - E.g.
    Suppose we only have the letters: a, b, c, d, e.
    $\log_2 5 = 2.32 \rightarrow$ So we need to use 3 bits
    a → 000
    b → 001
    c → 010
    d → 011
    e → 100

  - This isn't efficient.

- Huffman Encoding:
- Uses a greedy algo based on the frequency of each letter used.
- Want to assign shorter symbols to more frequently used letters.
- However, to avoid conflicts, we need to use prefix-free encoding. This is used to prevent confusion if part of an encoding is separate or not.

E.g. Say $x=0$, $y=01$. $z=1$ Then, when we see 01, we don't know if its xz or y.

- With Huffman's algo, order the letters in asc order of frequency and combine the 2 lowest letters until there are no more.

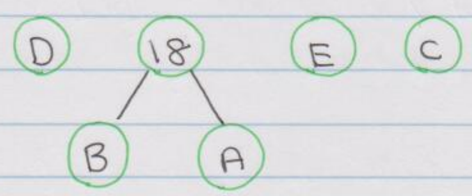
Note: When we "combine" 2 letters, we create a new node with frequency equal to the sum of the letters' frequency.

E.g. Say we use the following letters with the following frequency:

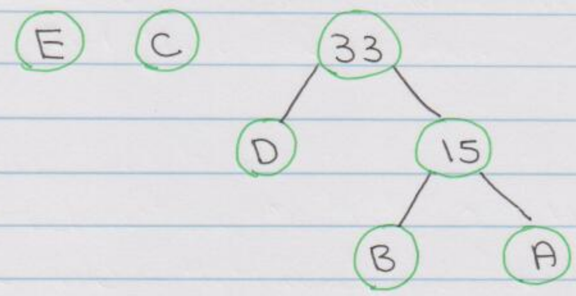
Letter	Frequency
A	10
B	8
C	30
D	15
E	24

1. B A D E C ← Ordered the letters in asc order of frequency.

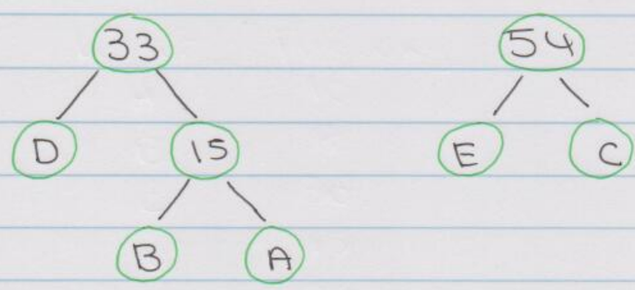
2. "Combine" the 2 letters with the least frequency → b, a



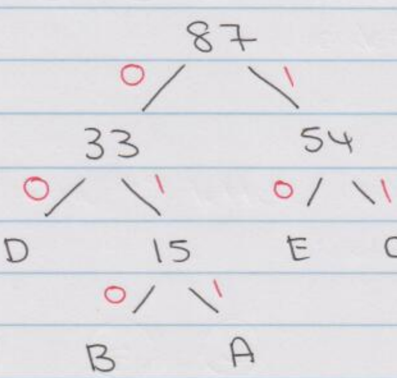
3. "Combine" the 2 letters with the lowest freq → D, 18



4. "Combine" the 2 letters with the lowest freq → E, C



5. "Combine" the 2 letters with the lowest freq → 33, 54



We stop now since there is 1 letter left.
Each time we go down a left path, add a 0.
Each time we go down a right path, use a 1.

A → 011, B → 010, C → 11, D → 00, E → 10

- Consider the chart below

Letter	Freq.	Huffman Encoding	Naive Encoding	H-L	N-L
a	10	011	000	30	30
b	8	010	001	24	24
c	30	11	010	60	90
d	15	00	011	30	45
e	24	10	100	48	72
		↑ From pg 18	↑ From pg 16	192	261

H-L is the total length used for each letter encoded using Huffman algo. It is the sum of the length of each encoding * the freq of each letter used.

N-L is the total length used for each letter encoded using the naive algo. It is the sum of the length of each encoding * the freq of each letter used.

We can see that with Huffmann's algo, we need 192 bits compared to the 261 bits needed if we were to use the naive algo.

Proof of Optimality:

We will use a proof by induction to prove this.

Let ω_n denote the freq of letter n.

Lemma 1: In any opt tree, if $\omega_x < \omega_y$, then $l_x \geq l_y$.

Proof:

Suppose for contradiction that $l_x < l_y$.

Since $(l_y - l_x) > 0$, we can do:

$W_x(l_y - l_x) < W_y(l_y - l_x)$

$W_x l_y - W_x l_x < W_y l_y - W_y l_x$

$\underbrace{W_x l_y + W_y l_x}_{\text{Total length of tree if x and y swapped}} < \underbrace{W_x l_x + W_y l_y}_{\text{Total length of opt tree}}$ ← Contradiction. $W_x l_x + W_y l_y$ must be smallest

Lemma 2: Suppose letters x and y have the smallest frequency. Then, there exists an opt tree where x and y are siblings.

I.e. x and y are the first to be combined.

Proof:

Suppose that x and y are not siblings.
We know from Lemma 1 that l_x and l_y must be the longest among all other nodes in the tree.
Wlog, suppose $\omega_x < \omega_y$.
If X and Y are not neighbours then there must exist at least one other node Z s.t. $\omega_x < \omega_z < \omega_y$.
But, that would mean X and Z are combined first, not X and Y.
This is a contradiction.

Proof that Huffman Algo creates an opt tree:

Base Case:

There are just 2 letters. Use 0 and 1 to encode.

Ind Hyp:

Suppose that the algo returns an opt soln for up to and including n-1 nodes/letters.

Ind Step:

Suppose that x and y have the lowest freq.

By lemma 1 and 2, we know that x and y must be siblings and have the longest length.

Let T be the tree created by Huffman after combining x and y.
Let O be the tree created by an opt soln combining x and y.
Let T' be the tree of n leaves Huffman Algo creates.
Let O' be the tree an opt algo creates in the first step.

$L(T) \leq L(O) \leftarrow$ By IH

$L(T') = L(T) + (\omega_x + \omega_y)$

$L(O') = L(O) + (\omega_x + \omega_y)$

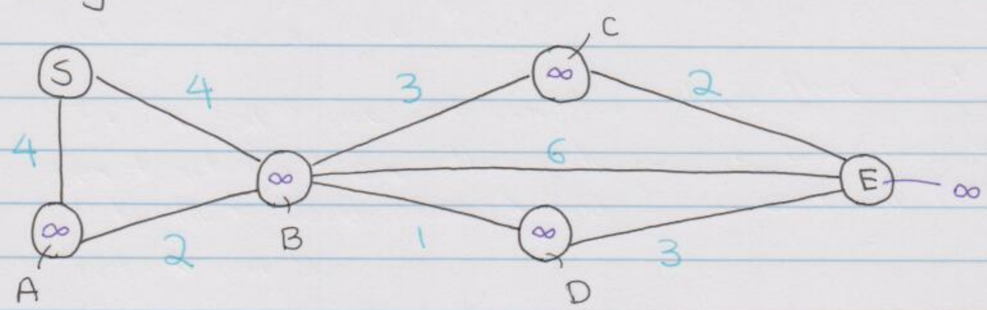
$L(T') \leq L(O')$

Dijkstra's Algo:

- Dijkstra's algo allows us to find the shortest path between any 2 vertices of a graph.

- Uses BFS and greedy algo.

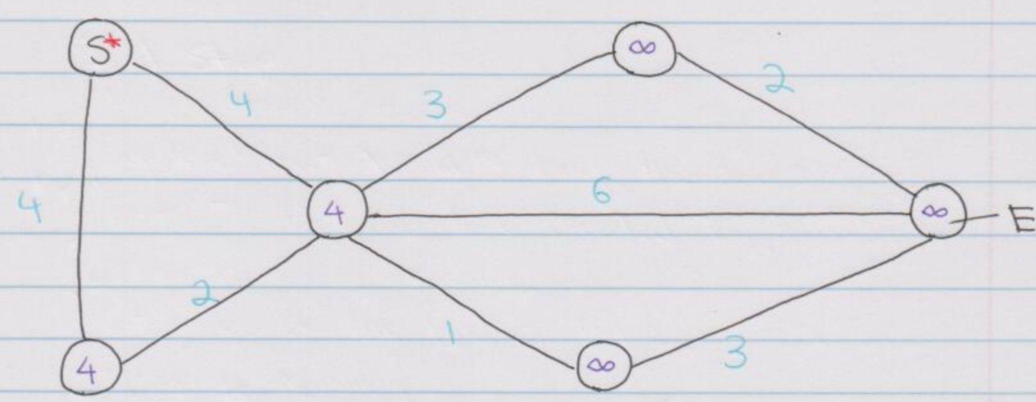
- E.g.



1. Start at S and visit all of its unvisited neighbour/adjacent nodes to update their distance from S if the new dist is less than the current dist. Then, mark S as visited.

Updated graph:

* means visited

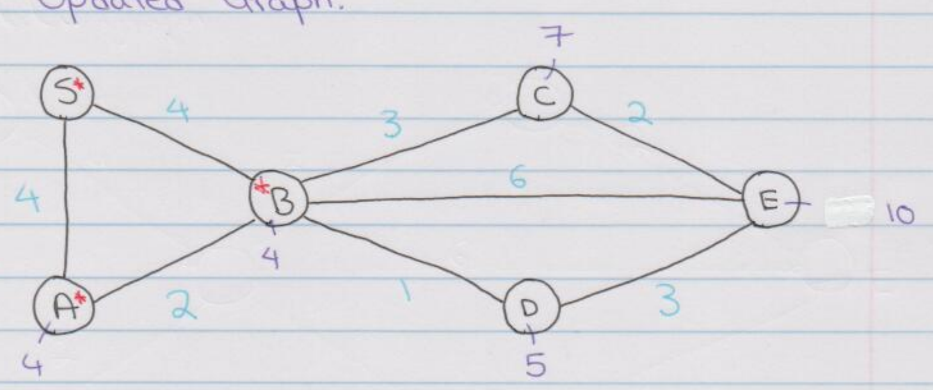


2. Go to node A. Then visit all of its unvisited adj nodes and update their distance if the new dist is less than their current dist. Then, mark A as visited.
   Note: When we calculate the dist from A to any of its unvisited adj nodes, we need to add the dist from S to A to the dist from A to that node. This is bc we're calculating the dist from S to a node.

   Note: Because S is marked as visited, we won't check it. Furthermore, the dist from A to B is 4+2 or 6. Since 6>4, we don't update the dist for B. Hence, the only change we make to the graph is marking A as visited.

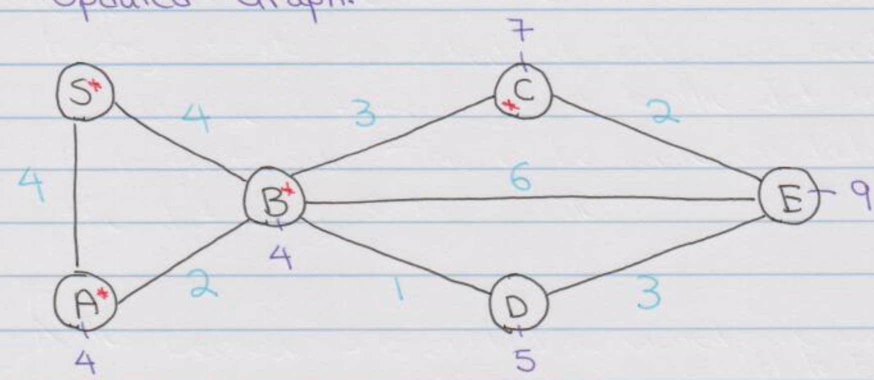
3. Go to node B. Then, visit all of its unvisited adj nodes, update their dist if needed and mark B as visited.

   Updated Graph:



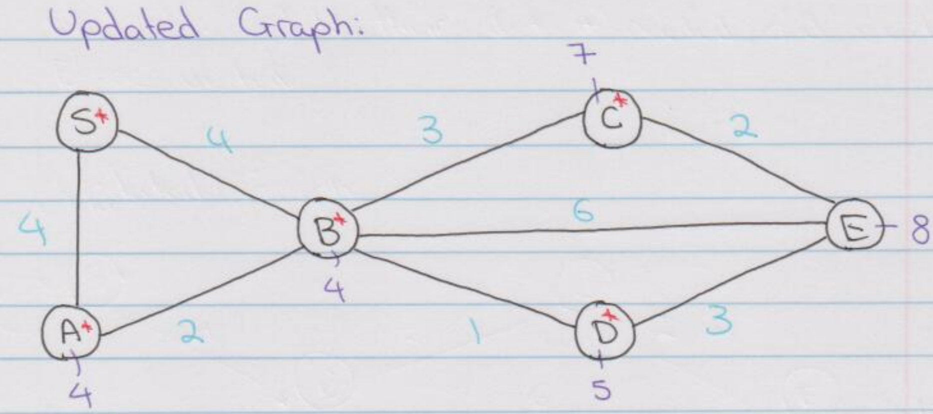
4. Go to node C, visit all of its unvisited adj nodes, update their dist if needed and mark C as visited.

Updated Graph:



5. Go to node D, visit all of its unvisited adj nodes, update their dist if needed and mark D as visited.

Updated Graph:



We're done

- Pseudo Code:

```
def Dijkstra(G, S):

    for each vertex v in G:
        if (v ≠ S):
            Set dist(v) = ∞
        else:
            set dist(v) = 0

    nodes = [S]
    visited = { }  ← Set
    for node in nodes:
        Remove node from nodes and add it to visited

        for each adj-node:
            if (adj-node not in visited):
                nodes.append(adj-node)
                dist(adj-node) = min( dist(adj-node),
                                      dist(node) + w(node, adj)
                                    )
                                                   ↑
                                      The weight
                                      of the edge
                                      connecting node
                                      and adj-node

    return distances
```

Final Words

- Assume that you have a problem that needs to be optimized (max or min) at a given point. A greedy algo makes the best choice at each stage.
  Note: You can't go back or reverse a decision.

- The first step to coming up with a greedy algo is to find the substructure of the problem.

- When proving that a greedy soln is opt, you can use either induction or contradiction. The idea is to compare your algo with a supposedly opt soln and show that your soln is no worse than the opt soln.