CS 161 – Section 7

CA: [Name of CA]

Agenda

- Greedy Algorithms
  - Activity selection
  - Scheduling
  - Huffman Encoding
- Minimum Spanning Trees
  - Prim’s
  - Kruskal’s

Greedy Algorithms

Common strategy
for greedy algorithms

- Make a **series of choices**.
- Show that, at each step, our choice **won't rule out all optimal solutions** at the end of the day.
- After we've made all our choices, there is an optimal solution we haven't ruled out, **so we must have found one.**

Common strategy (formally)

for greedy algorithms

- Inductive Hypothesis:
  - After greedy choice t, you haven't ruled out success.
- Base case:
  - Success is possible before you make any choices.
- Inductive step:
  - If you haven't ruled out success after choice t, then you won't rule out success after choice t+1.
- Conclusion:
  - If you reach the end of the algorithm and haven't ruled out success then you must have succeeded.

"Success" here means "finding an optimal solution."

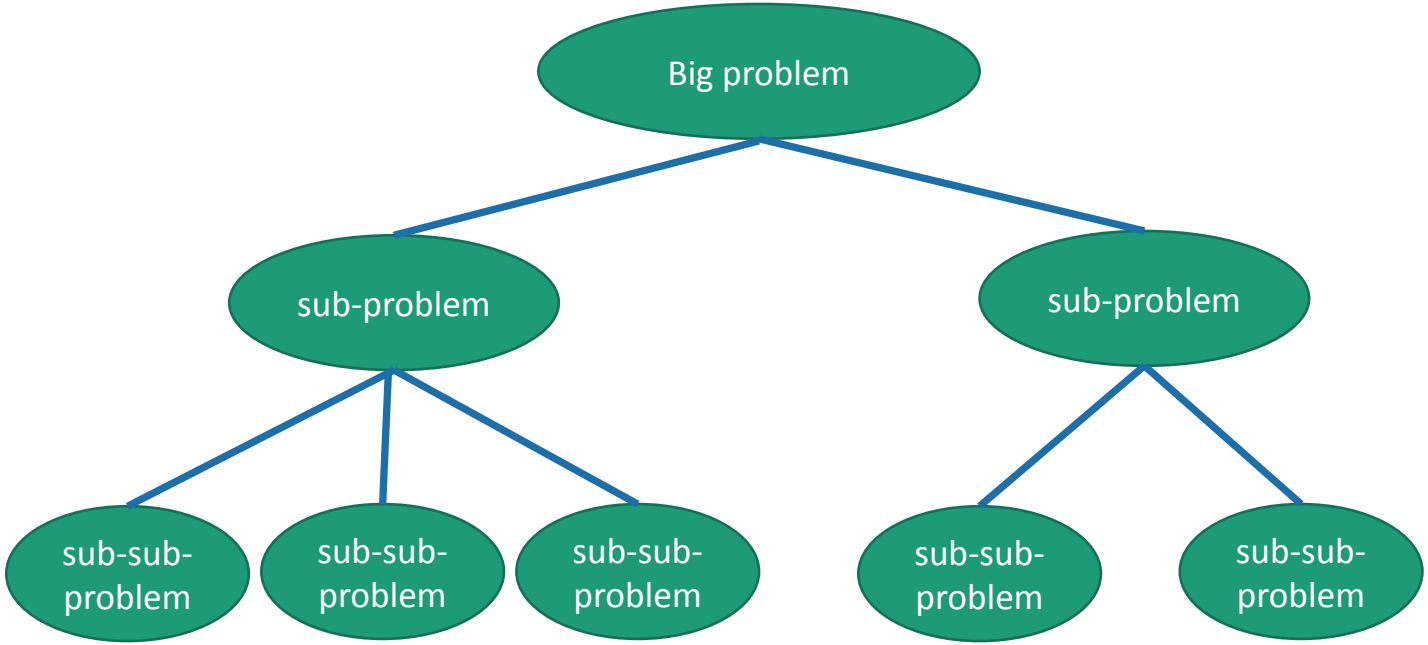
Common strategy

for showing we don't rule out success

- Suppose that you're on track to make an optimal solution T*.
  - Eg, after you've picked activity i, you're still on track.
- Suppose that T* *disagrees* with your next greedy choice.
  - Eg, it *doesn't* involve activity k.
- Manipulate T* in order to make a solution T that's not worse but that *agrees* with your greedy choice.
  - Eg, swap whatever activity T* did pick next with activity k.

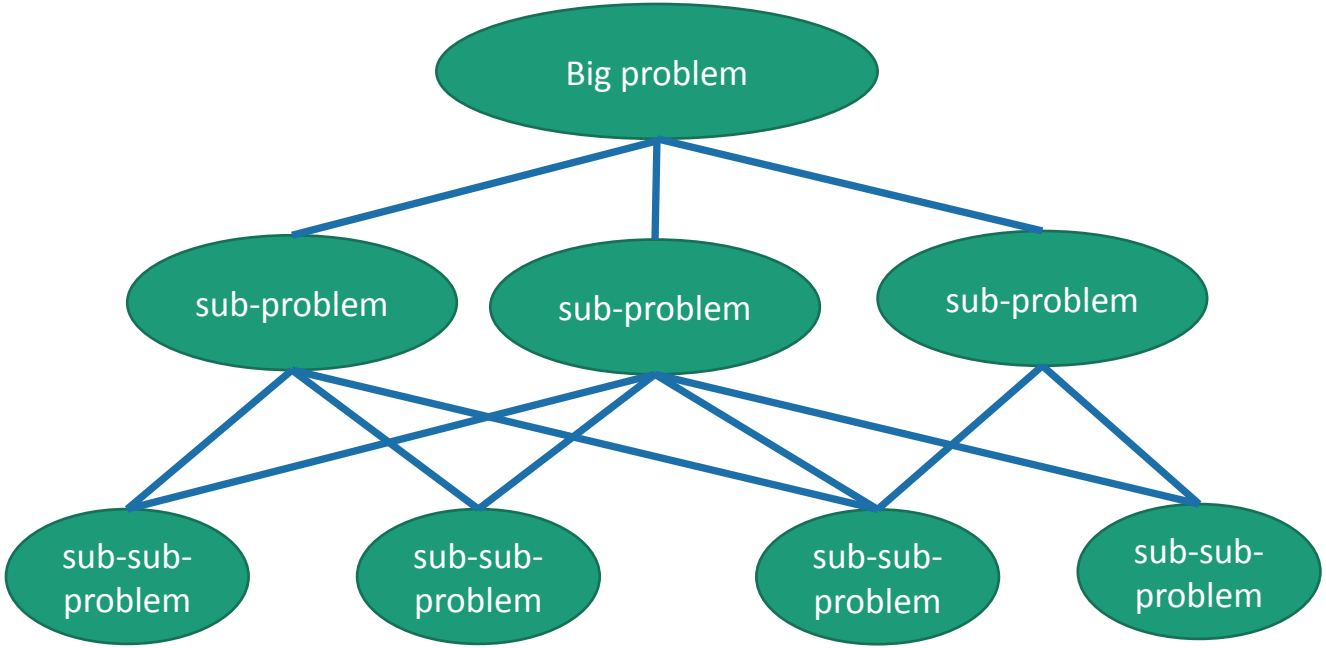
Sub-problem graph view

- Divide-and-conquer:



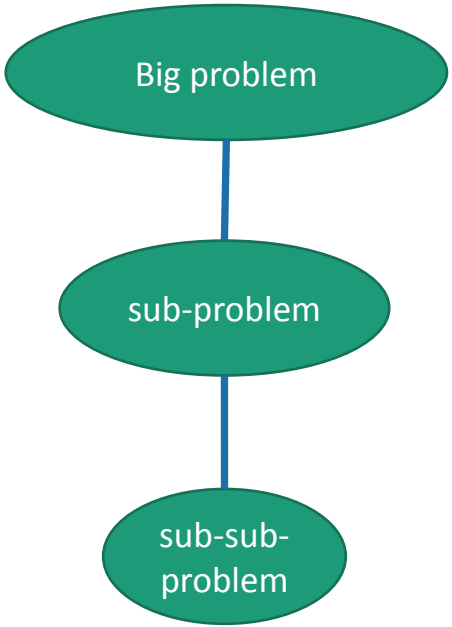
Sub-problem graph view

- Dynamic Programming:



Sub-problem graph view

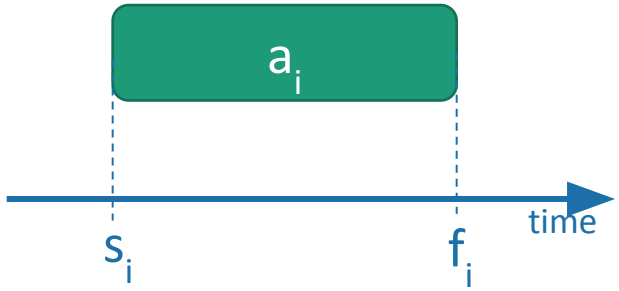
- Greedy algorithms:



- Not only is there **optimal sub-structure:**
  - optimal solutions to a problem are made up from optimal solutions of sub-problems
- but each problem **depends on only one sub-problem**.

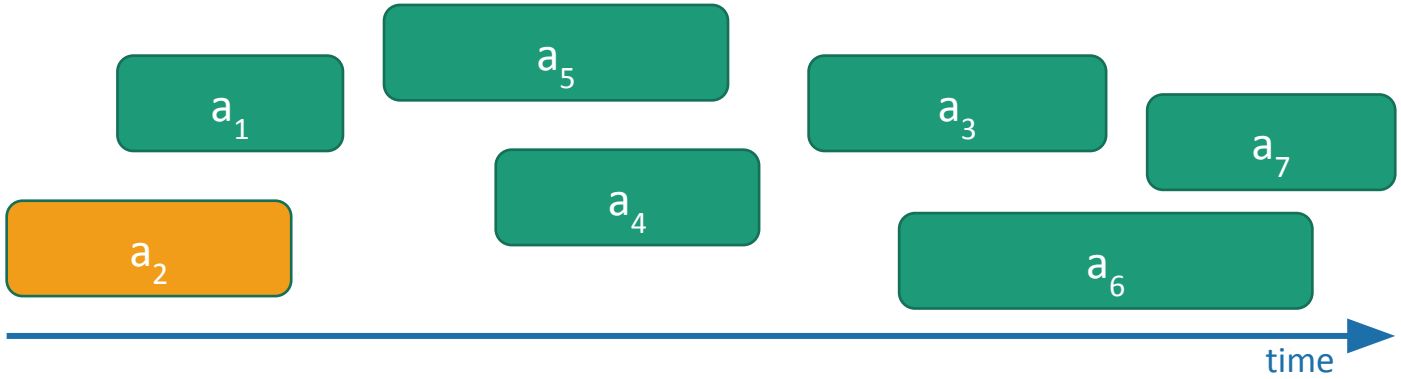
Social Activity selection

- Input:
  - Activities $a_1, a_2, \ldots, a_n$
  - Start times $s_1, s_2, \ldots, s_n$
  - Finish times $f_1, f_2, \ldots, f_n$



- Output:
  - A way to maximize the number of activities you can do today.

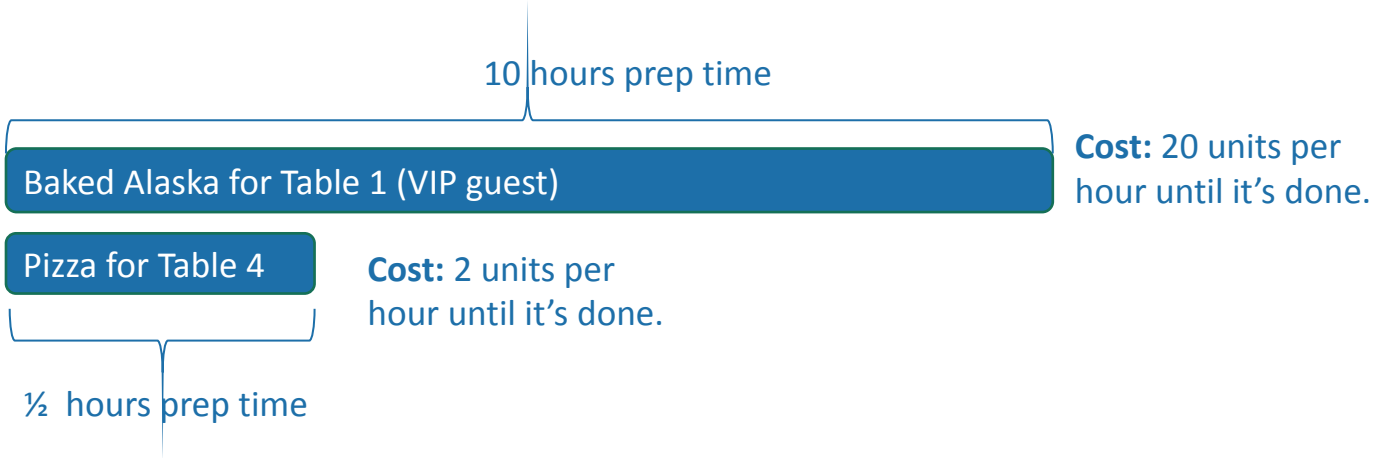
Greedy algorithm



- Pick activity you can add with the smallest finish time.
- Repeat.

Scheduling

- n jobs
- Job i takes t_i hours
- For every hour that passes until job i is done, pay c_i



- Baked Alaska, then Pizza: costs $10 \cdot 20 + (10 + ½) \cdot 2 = 221$
- Pizza, then Baked Alaska: costs $½ \cdot 2 + (½ + 10) \cdot 20 = 211$

Greedy algorithm

- Choose the job with the biggest $\frac{\text{cost of delay}}{\text{time it takes}}$ ratio.

Claim 1:

If $\frac{\text{A.cost}}{\text{A.time}} \geq \frac{\text{B.cost}}{\text{B.time}}$ then **AB** is better than **BA**

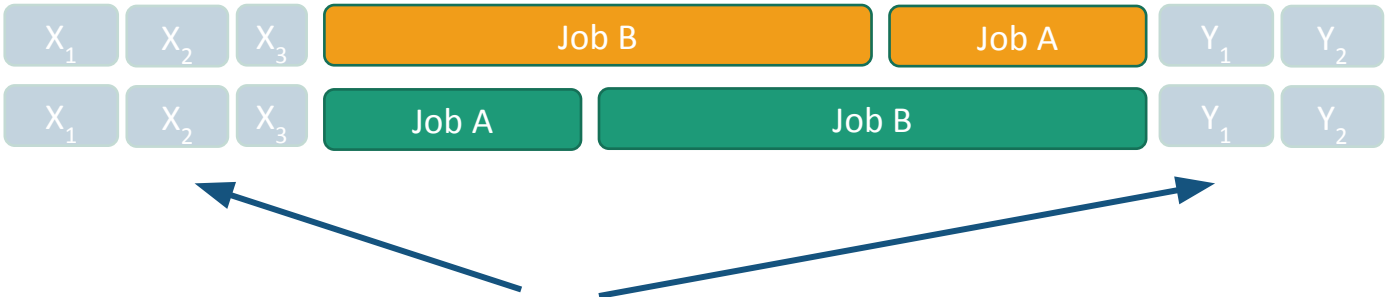
Proof: we just did this…

AB is better than **BA** when:

$$xz + (x+y)w \leq yw + (x+y)z$$
$$xz + xw + yw \leq yw + xz + yz$$
$$wx \leq yz$$
$$\frac{w}{y} \leq \frac{z}{x}$$

Claim 2:

If $\frac{\text{A.cost}}{\text{A.time}} \geq \frac{\text{B.cost}}{\text{B.time}}$ then $X_1X_2\ldots\mathbf{AB}Y_1Y_2\ldots$ is better than $X_1X_2\ldots\mathbf{BA}Y_1Y_2\ldots$



Proof: Other jobs wait the same amount of time!

(And we know **AB** is better than **BA** by Claim 1)

Huffman Encoding

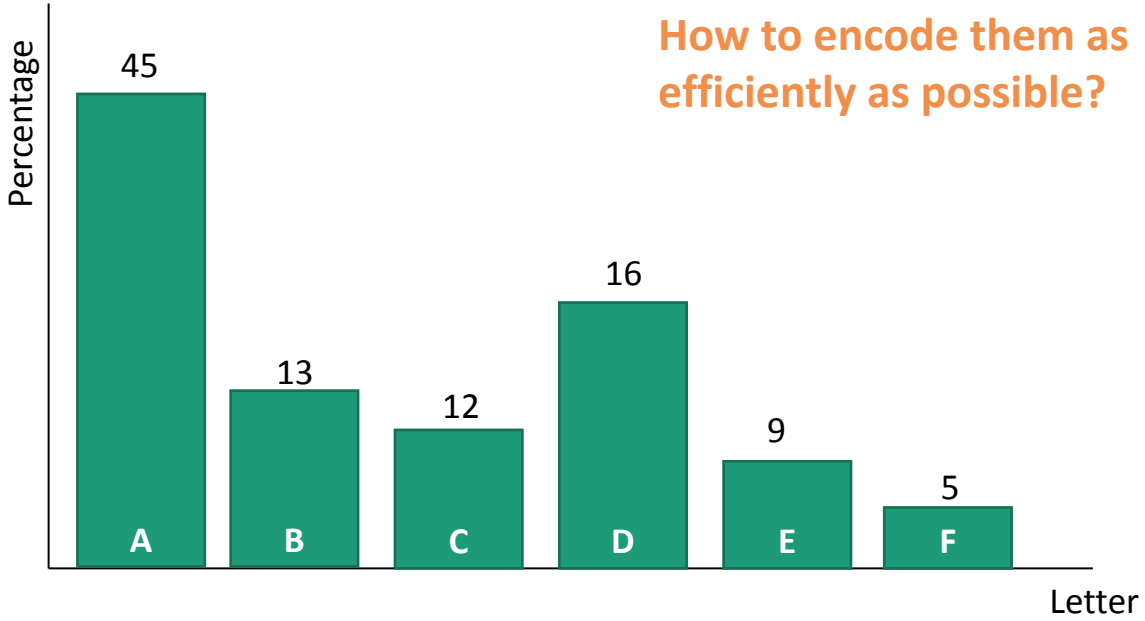
ASCII is pretty wasteful for English sentences. If **e** shows up so often, we should have a more parsimonious way of representing it!

- everyday english sentence
- **01100101** 01110110 **01100101** 01110010 01111001 01100100 01100001 01111001 00100000 **01100101** 01101110 01100111 01101100 01101001 01110011 01101000 00100000 01110011 **01100101** 01101110 01110100 **01100101** 01101110 01100011 **01100101**

- qwertyui_opasdfg+hjklzxcv
- 01110001 01110111 **01100101** 01110010 01110100 01111001 01110101 01101001 01011111 01101111 01110000 01100001 01110011 01100100 01100110 01100111 00101011 01101000 01101010 01101011 01101100 01111010 01111000 01100011 01110110

Suppose we have some distribution on characters

For simplicity, let's go with this made-up example



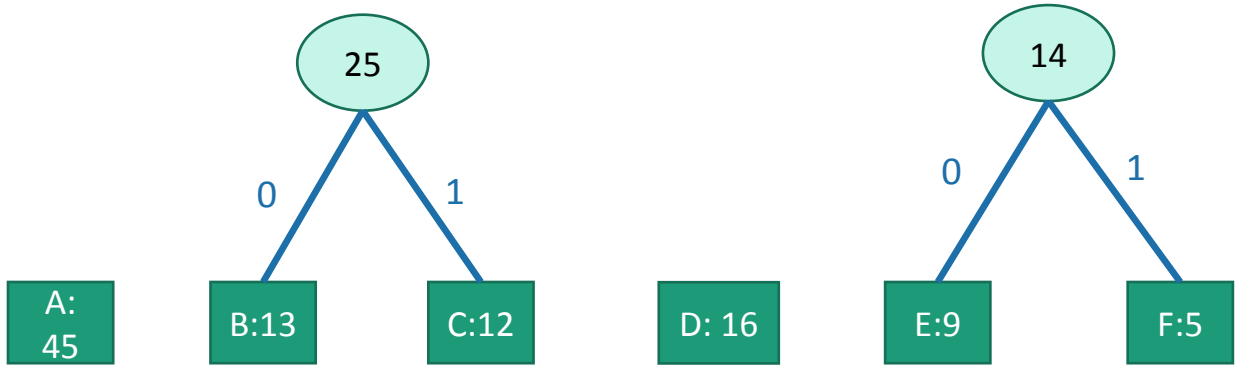
How to encode them as efficiently as possible?

Solution

greedily build subtrees, starting with the infrequent letters

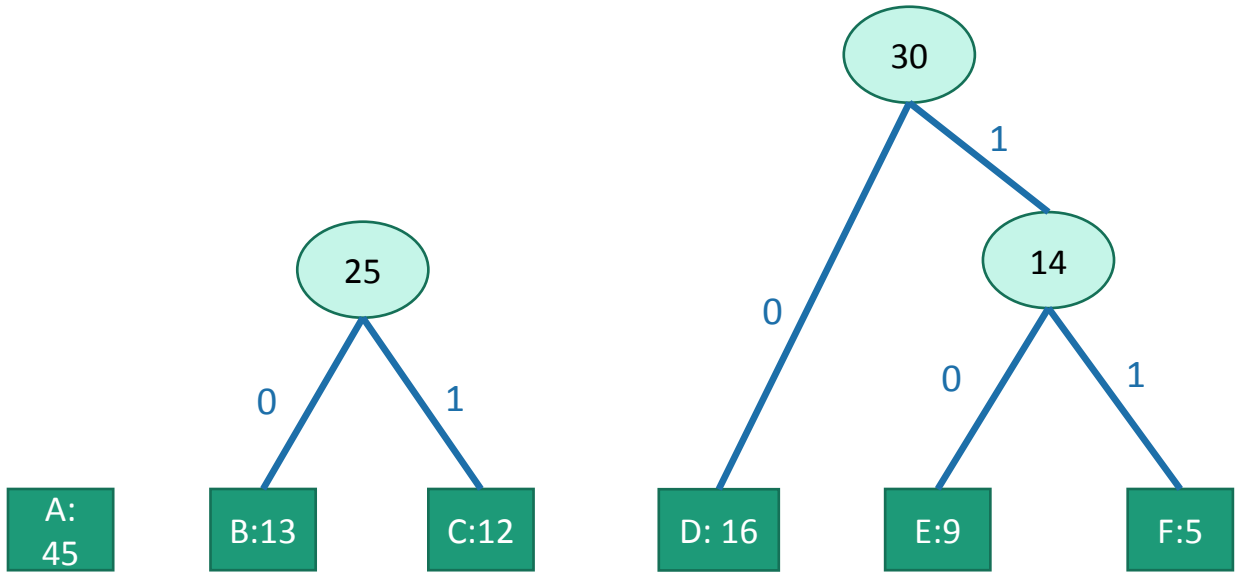
Solution

greedily build subtrees, starting with the infrequent letters



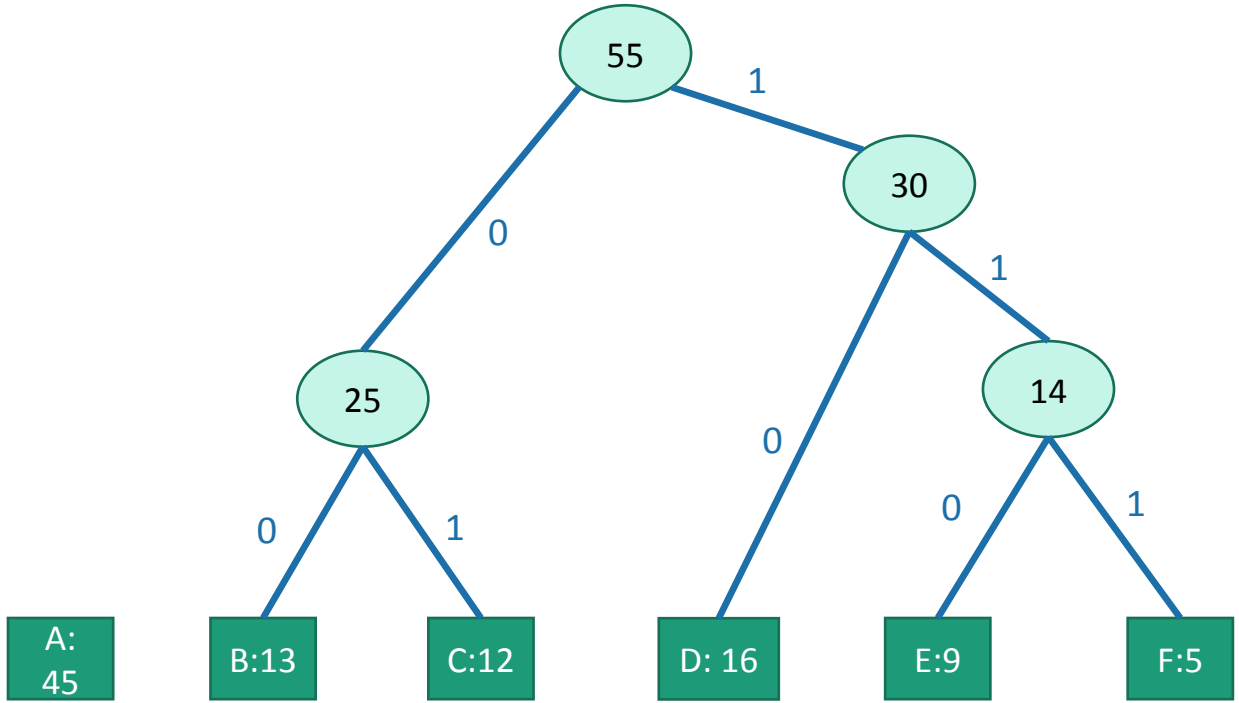
Solution

greedily build subtrees, starting with the infrequent letters



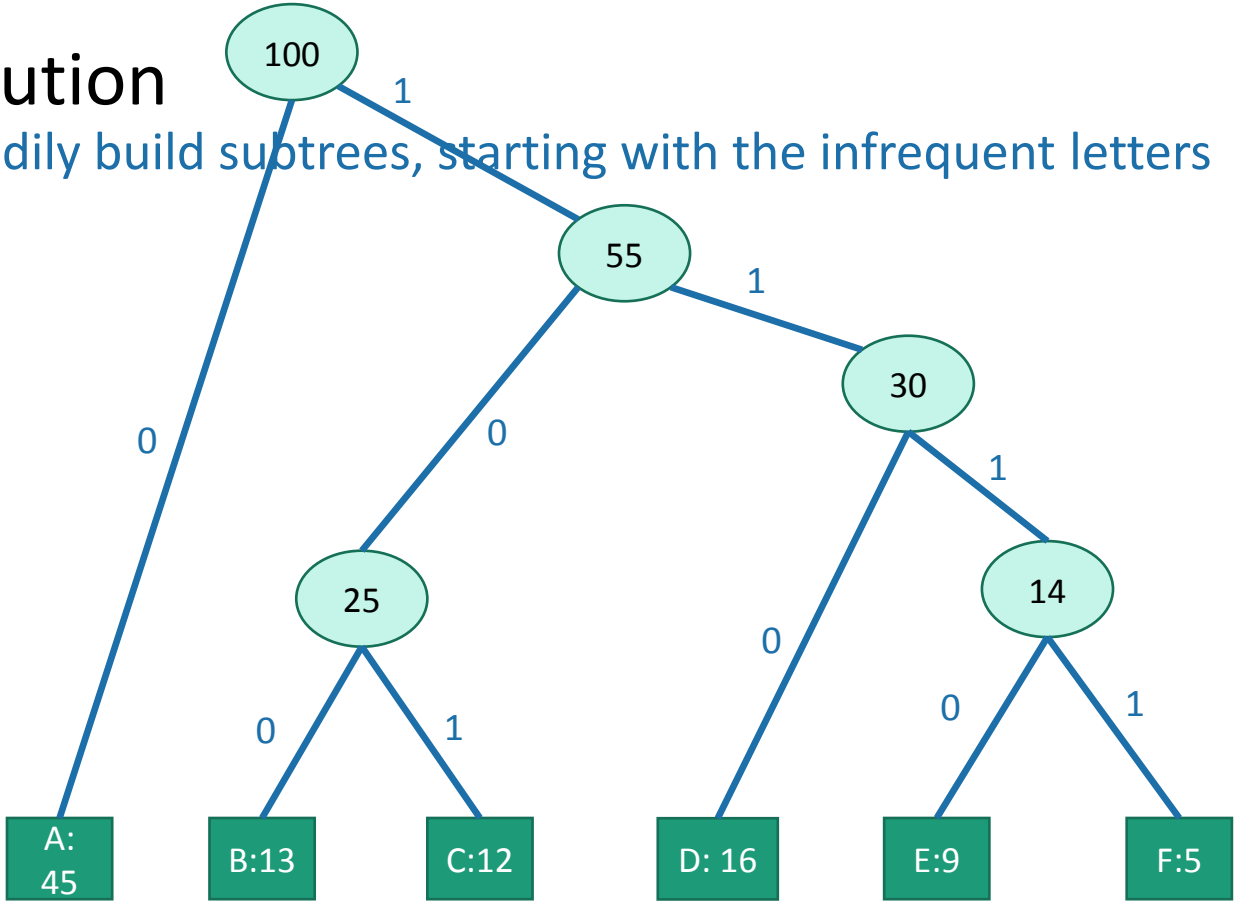
Solution

greedily build subtrees, starting with the infrequent letters



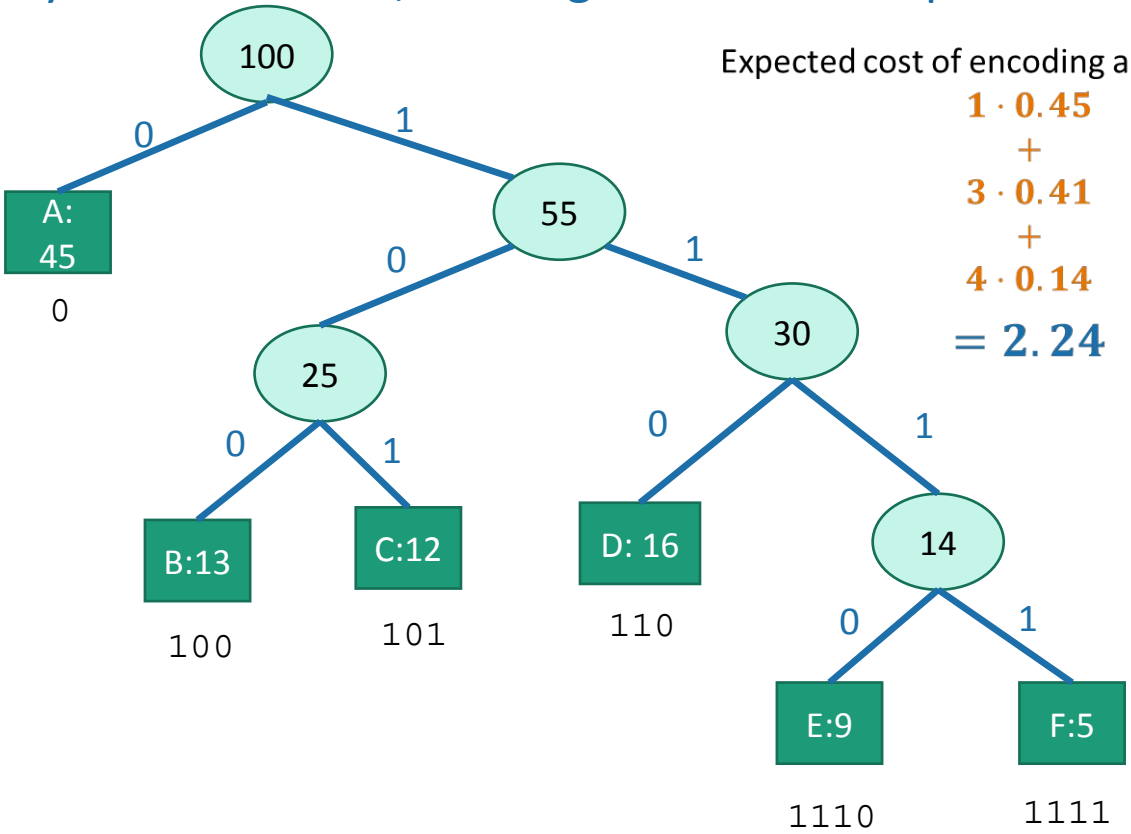
Solution

greedily build subtrees, starting with the infrequent letters



Solution

greedily build subtrees, starting with the infrequent letters



Expected cost of encoding a letter:

$$1 \cdot 0.45 + 3 \cdot 0.41 + 4 \cdot 0.14 = 2.24$$

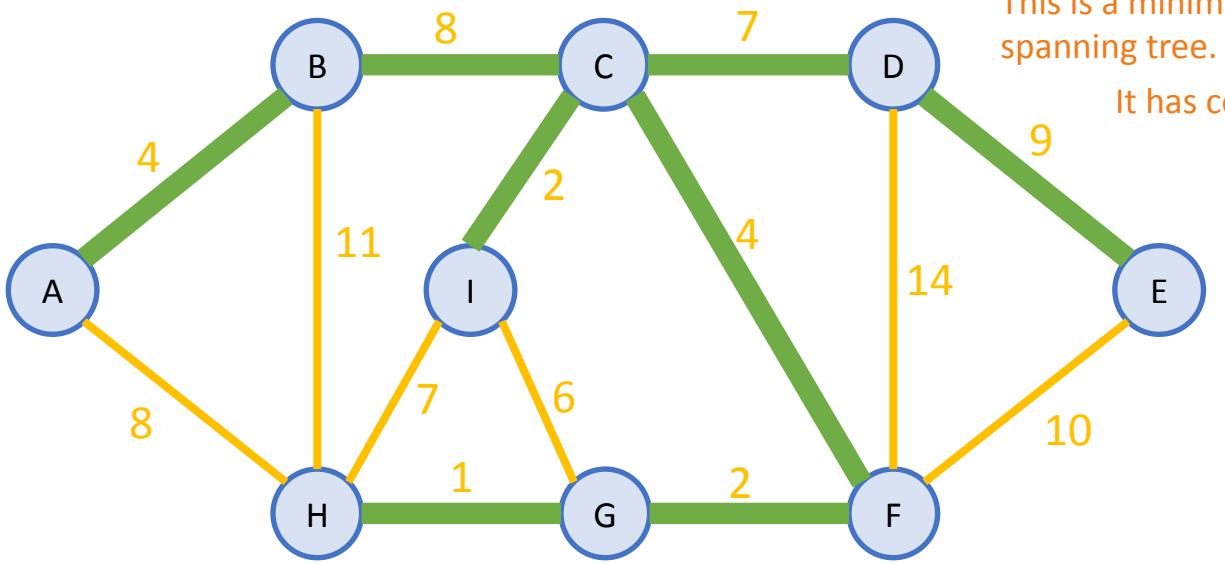
Recap

- Greedy algorithms!
- Often easy to write down
  - But may be hard to come up with and hard to justify
- The natural greedy algorithm may not always be correct.
- A problem is a good candidate for a greedy algorithm if:
  - it has optimal substructure
  - that optimal substructure is **REALLY NICE**
    - solutions depend on just one other sub-problem.

Minimum Spanning Tree

Minimum Spanning Tree

Say we have an undirected weighted graph



This is a minimum spanning tree.

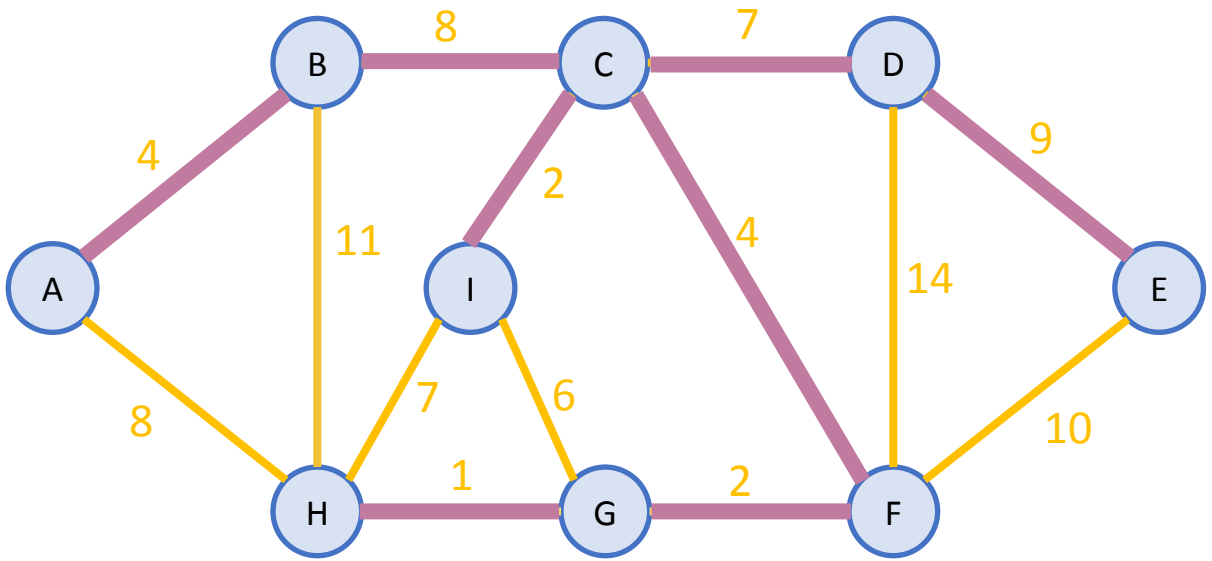
It has cost 37

minimum **of minimal cost**

A **spanning tree** is a **tree** that connects all of the vertices.

Algorithms for Minimum Spanning Tree

Plan 1: Start growing a tree, greedily add the shortest edge we can to grow the tree.



Prim's Algorithm

Prim(G = (V,E), starting vertex s):

- MST = {}
- **for each** vertex v:
  - k[v] = ∞
  - mark v as **disconnected**
- k[s] = 0; mark s as **connected**

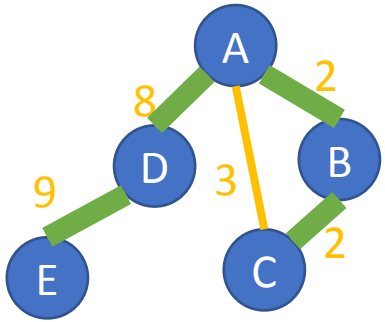
Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's unreached neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached**, and **add (p[u],u) to MST.**

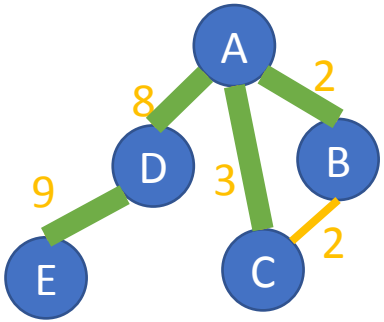
This should look pretty familiar…

Both grow a tree:

- Prim / MST



- Dijkstra / Shortest Path



But optimize different functions:

$$\sum_v w(\text{last edge before } v)$$

$$\sum_v w(\text{total path to } v)$$

w/ corresponding update rules:

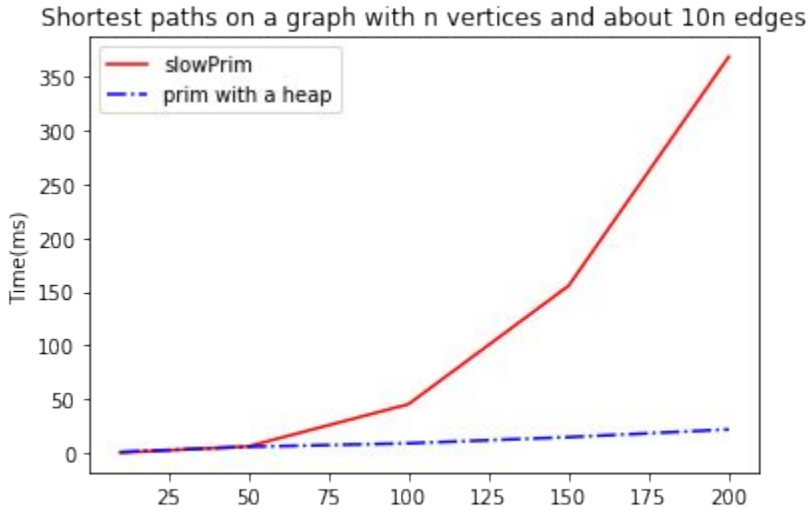
k[v] = min(k[v], w(u,v))

d[v] = min(d[v], d[u] + w(u,v))

One thing that is similar:

Running time

- Exactly the same as Dijkstra:
  - O(mlog(n)) using a Red-Black tree or binary heap as a priority queue.
  - O(m + nlog(n)) if we use a Fibonacci Heap*.



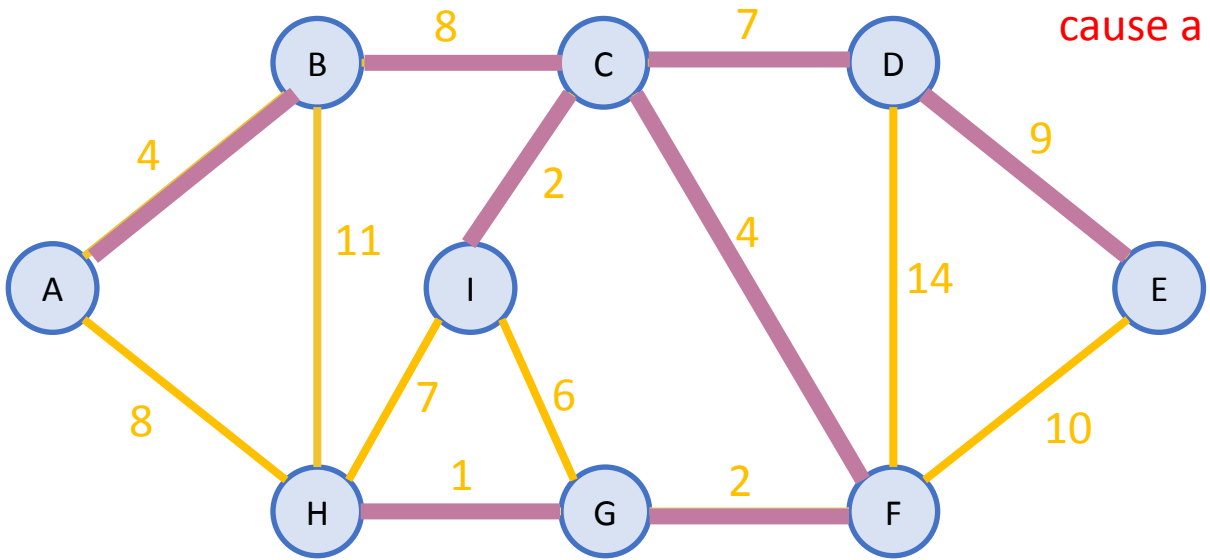
*See CS166

Idea 2

what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?

That won't cause a cycle



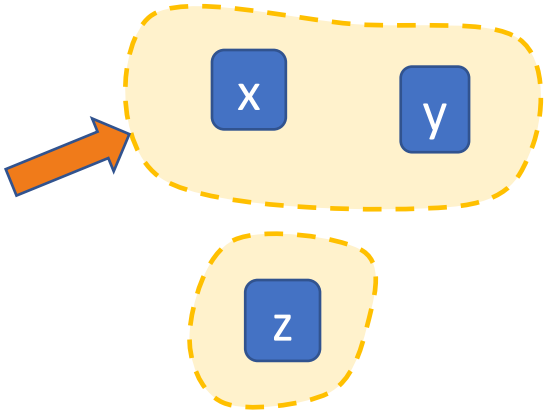
Union-find data structure
also called disjoint-set data structure

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)

find(x)
```



We've discovered Kruskal's algorithm!

- **kruskal**(G = (V,E)):
  - Sort E by weight in non-decreasing order
  - MST = {} // initialize an empty tree
  - **for** v in V:
    - **makeSet**(v) // put each vertex in its own tree in the forest
  - **for** (u,v) in E: // go through the edges in sorted order
    - **if find**(u) != **find**(v): // if u and v are not in the same tree
      - add (u,v) to MST
      - **union**(u,v) // merge u's tree with v's tree
  - **return** MST

Compare and contrast

- Prim:
  - Grows a tree.
  - Time O(mlog(n)) with a red-black tree
  - Time O(m + nlog(n)) with a Fibonacci heap
- Kruskal:
  - Grows a forest.
  - Time O(mlog(n)) with a union-find data structure
  - If you can do radixSort on the edge weights, O(mα(n))

Prim might be a better idea on dense graphs if you can't radixSort edge weights

Kruskal might be a better idea on sparse graphs if you can radixSort edge weights

Thank you