Complexity & Algorithms, Spring 2026

Greedy

Approximation

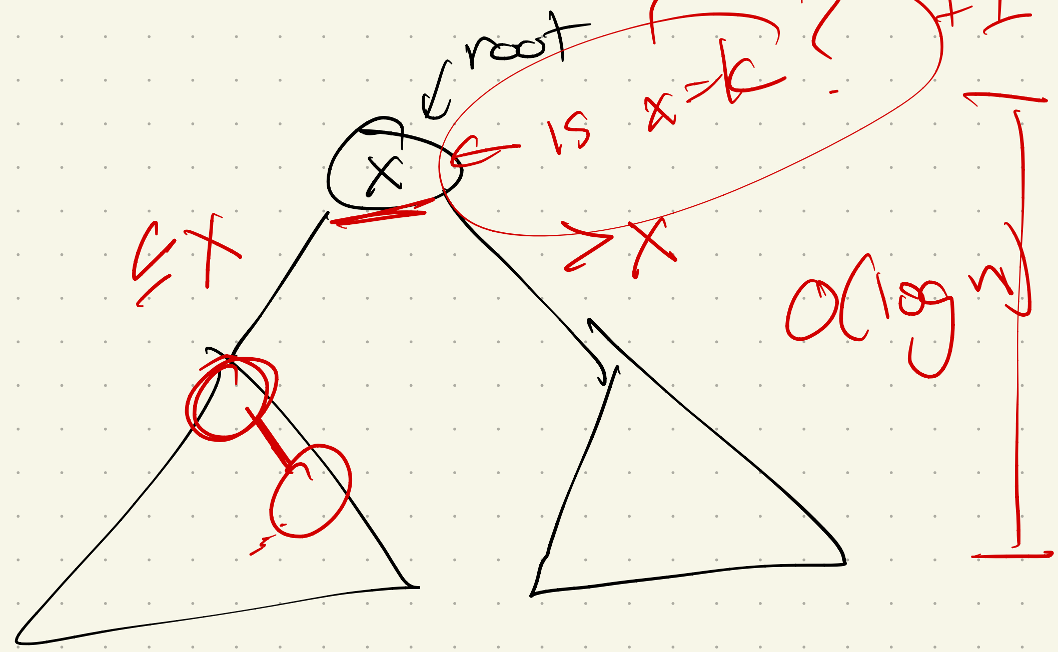
Recap

- HW1: due Thursday
- Reading this time: thoughts?
- Today: Finish dynamic programming, & on to greed!

Optimal Binary search trees

Recall: BSTs.

n nodes



Time to search for a value k in T:

O(depth of K in T)

Goal: If I know how many times you'll look up each value in T, can I build the perfect BST?

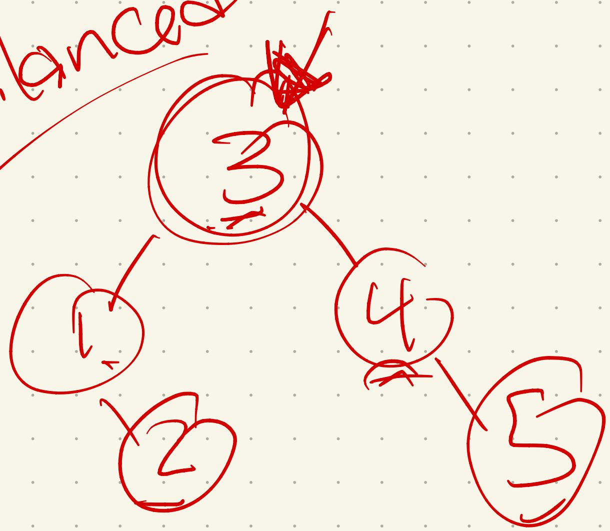
Question: Why not balanced?

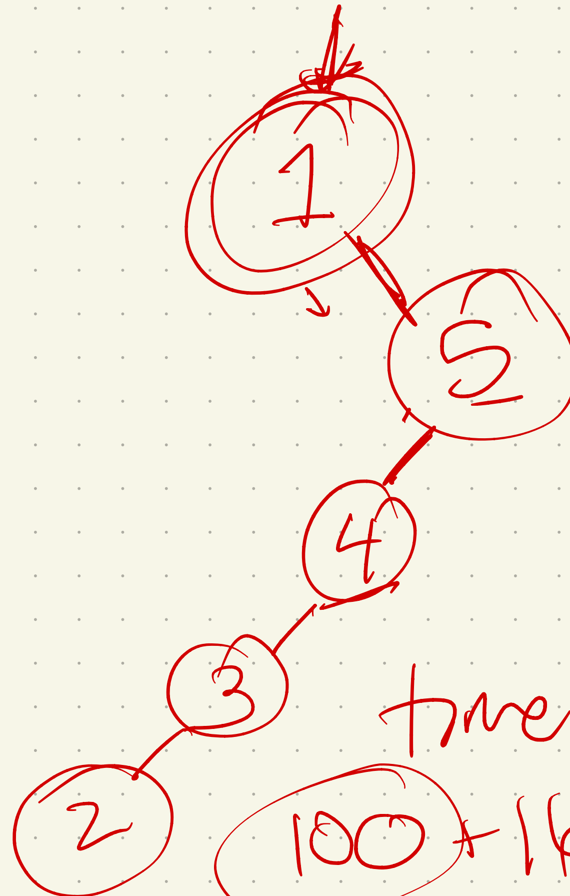
50

F	100	1	1	2	8
X	1	2	3	4	5

freq.
values

balanced





time:

$200 + 3 + 1 + 4 + 24$

$= 232$

time:

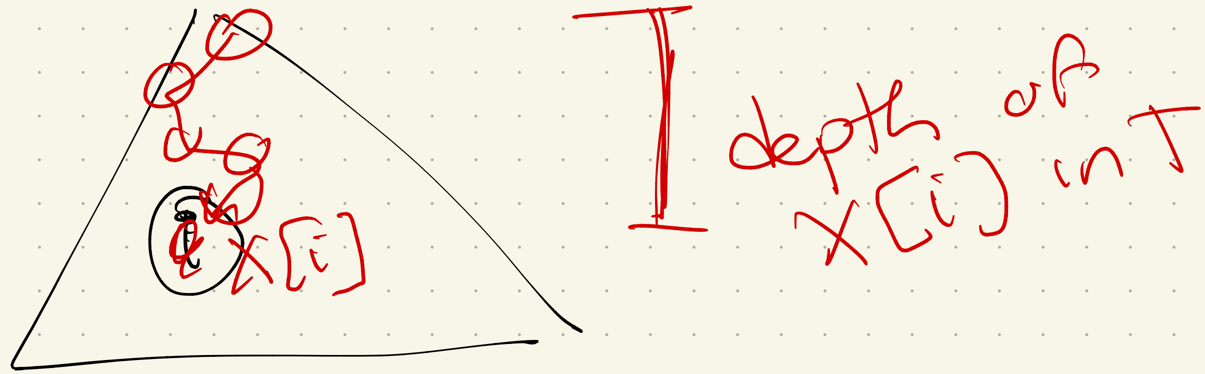
$100 + 16 + 6 +$
$4 + 5$

$= 131 < 232$

General problem: Given $X[1..n]$ + $F[1..n]$, where $X[i]$ has $F[i]$ searches, compute optimal BST:

minimum cost $= \sum_i F[i] \cdot (\text{depth in } T)$

Why?



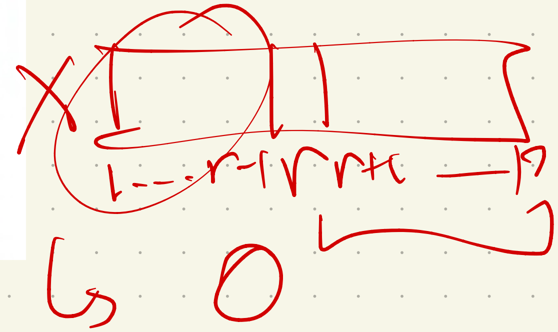
Intuition: put max freq. on top

↳ GREED!

(NO)

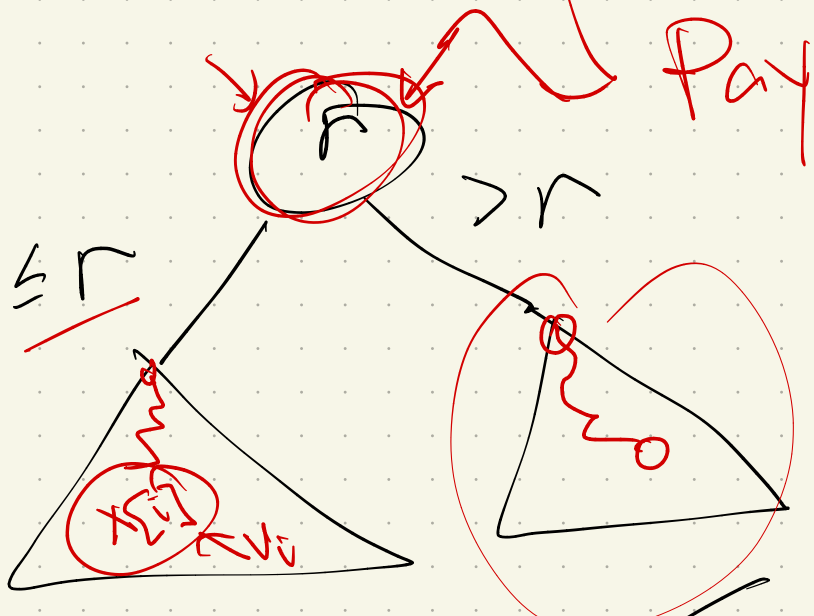
Last Chapter Assume X is sorted.

$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] + \sum_{i=1}^{r-1} f[i] \cdot \#\text{ancestors of } v_i \text{ in } left(T) + \sum_{i=r+1}^{n} f[i] \cdot \#\text{ancestors of } v_i \text{ in } right(T)$$



Why? Let root be r:

Pay for the root on every query



Essentially regrouping: $\sum_i F[i] \cdot \text{depth}$

$= \sum_{\text{levels } k}$ (frequencies of nodes at level ≥ k)

Recursive (backtracking) strategy

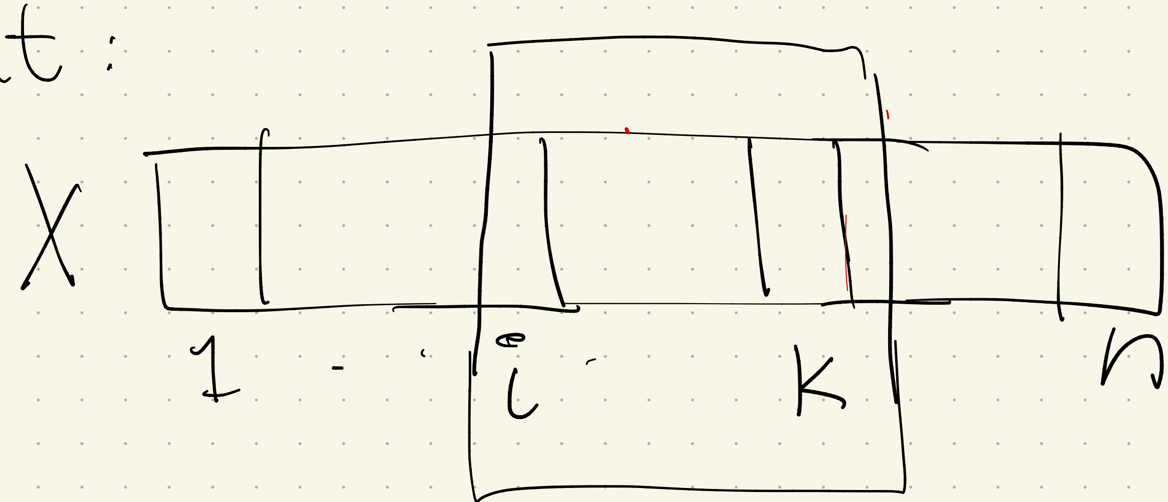
$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] + \sum_{i=1}^{r-1} f[i] \cdot \#\text{ancestors of } v_i \text{ in } left(T) + \sum_{i=r+1}^{n} f[i] \cdot \#\text{ancestors of } v_i \text{ in } right(T)$$

$$\Rightarrow \quad OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \displaystyle\sum_{j=i}^{k} f[i] + \underbrace{\min_{i \le r \le k}}_{\text{Choose best root!}} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$

How to memoize?

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \displaystyle\sum_{j=i}^{k} f[i] + \min_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$

Remember input:



build best tree here

Everyone searches at root

↳ precompute

Let $F[i][k] = \sum_{j=i}^{k} f[j]$

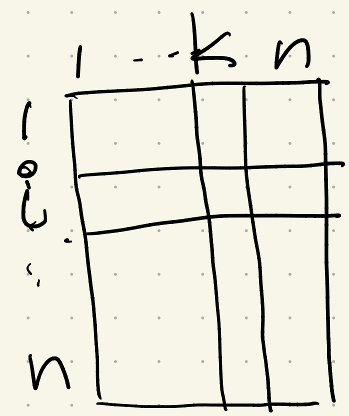
Now:

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[i] + \min\limits_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$

$$\Rightarrow OptCost(i,k) = \begin{cases} 0 \\ F[i][k] + \end{cases}$$

Memoize: $0 \le i \le k \le n$

So: 2d table!

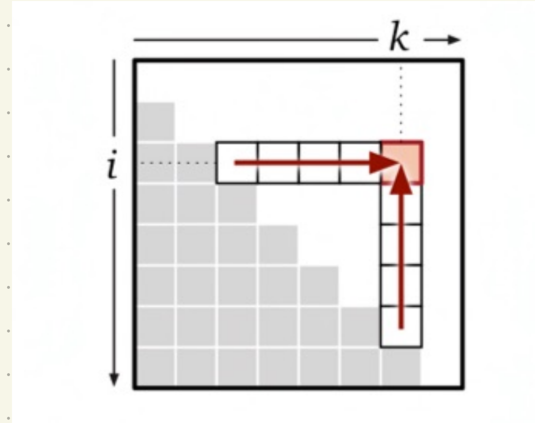


Each $O[i][k]$ needs:

- $F[i][k]$
- and

This picture (prettier):

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ F[i,k] + \min\limits_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$



So:

```
OptimalBST(f[1..n]):
  InitF(f[1..n])
  for i ← 1 to n+1
      OptCost[i, i−1] ← 0
  for d ← 0 to n−1
      for i ← 1 to n−d          ⟨⟨...or whatever⟩⟩
          ComputeOptCost(i, i+d)
  return OptCost[1, n]
```

Time:

Space:

Other ones in reading:

- Subset Sum: $O(nT)$

 <u>but</u>

- Independent Sets in trees:

 Not an array!

 For each node, need to store values

 ↳ Use the tree

Classical greedy algorithms

Some algorithms <u>can</u> be solved correctly (& fast) with a greedy approach.

<u>Ex</u>: Coins & making change

In the US: 1¢, 5¢, 10¢, 25¢

If I want to give 72¢ in change, how can I do it using fewest coins?

When greed seems to work, how to prove?

- Assume optimal is different than greedy
- Find the "first" place they differ.
- Argue that we can exchange the two without making optimal worse.

$\Longrightarrow$ there is no "first place" where they must differ, so greedy in fact is an optimal solution.

Proof techniques:

Coin example:

Suppose greedy $\neq$ opt.

opt:

greedy:

Dynamic Programming vs Greedy

Dyn. pro: try all possibilities
↳ but intelligently!

In greedy algorithms, we avoid building all possibilities

How?

Some part of the problem's structure lets us pick a local "best" and have it lead to a global best.

Doesn't always work!

Examples:

- Edit distance:

- Optimal BSTs:

Greedy approximation

While greed can work, it often fails!

- but - a useful heuristic!

Still need to find the <u>right</u> greedy strategy, though.

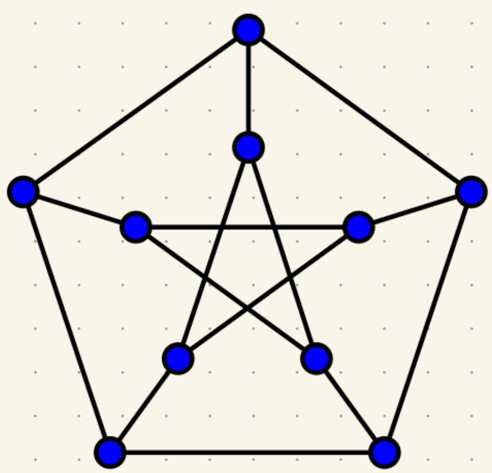
(and then some proof of approximation ratio)

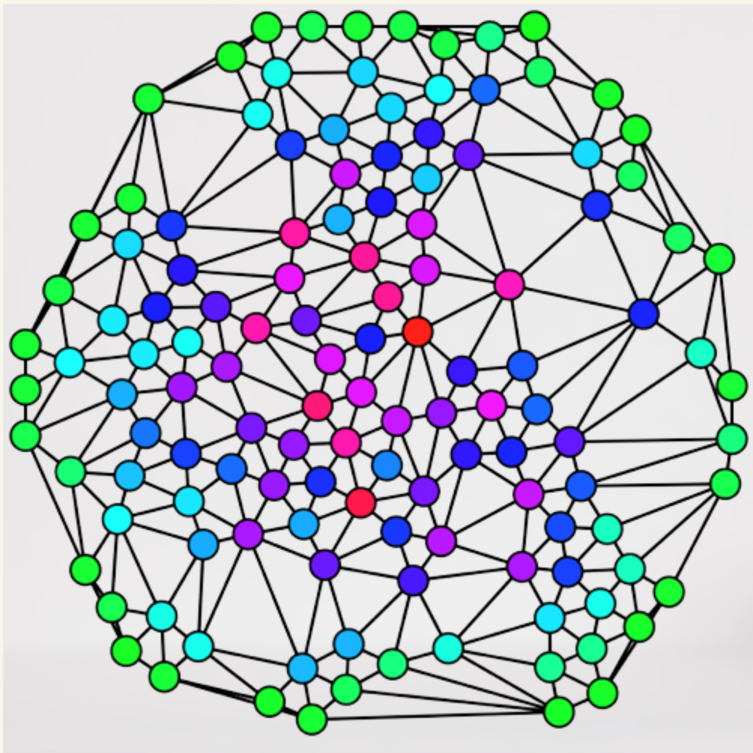
↳ Not obvious!

First example

Vertex cover: Given a graph $G=(V,E)$, choose a set of vertices $S \subseteq V$ such that every $e \in E$ is incident to some $v \in S$.

Examples:





How hard?

Easy to find a cover:

Challenge:

Note: In general, NP-Hard. (More later...)

One idea: Use vertices with high degree.

Why?

Greedy algorithm:

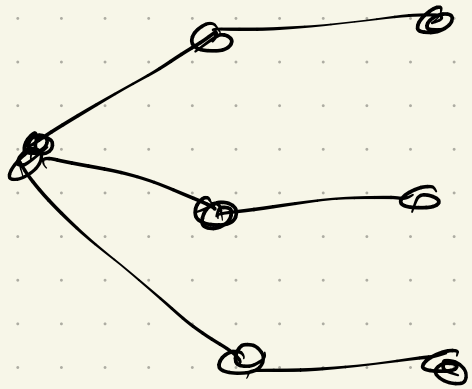
GREEDYVERTEXCOVER(G):
    $C \leftarrow \varnothing$
    while G has at least one edge
        $v \leftarrow$ vertex in G with maximum degree
        $G \leftarrow G \setminus v$
        $C \leftarrow C \cup v$
    return C

Why?

Question: does this ever give the min set?

Question: how to make it fail?

Need high degree vertices that are <u>not</u> optimal.



But:

Can we prove this is an approximation to optimal?

ie $|C| > |Opt|$ (see last slide)

but $|C| \leq \alpha \cdot |OPT|$?

Note: Nothing in our algorithm tells us what to aim for!

prev. example $\Rightarrow$

Let's check some notation here...

Dfn's for Approx:

Let $OPT(x)$ = value of optimal solution

$A(x)$ = value of solution computed by algorithm A

A is an $\alpha(n)$-approximation algorithm if.

1. $$\frac{OPT(x)}{A(x)} \le \alpha(n)$$

2. and $$\frac{A(x)}{OPT(x)} \le \alpha(n)$$

$\alpha(n)$ is called approximation factor.

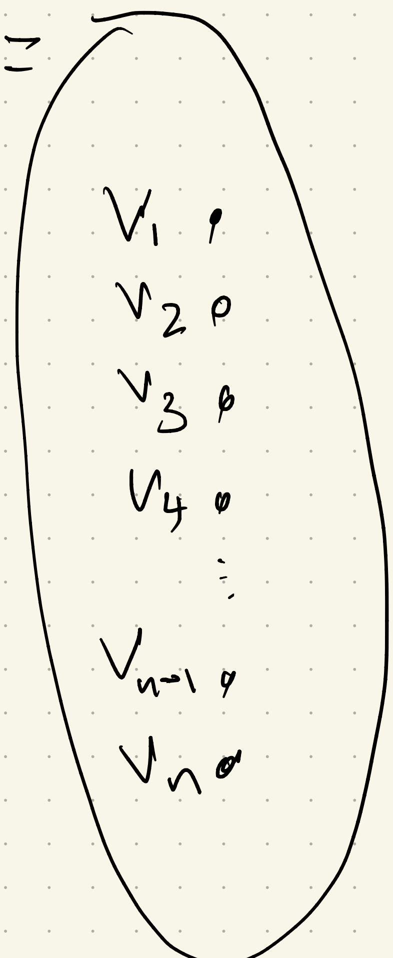
Back to VC

Question: is it a 2-approximation?

No. (But not obvious...)

Construction: bipartite graph $G = (V, E)$ where $V = L \cup R$

$L =$

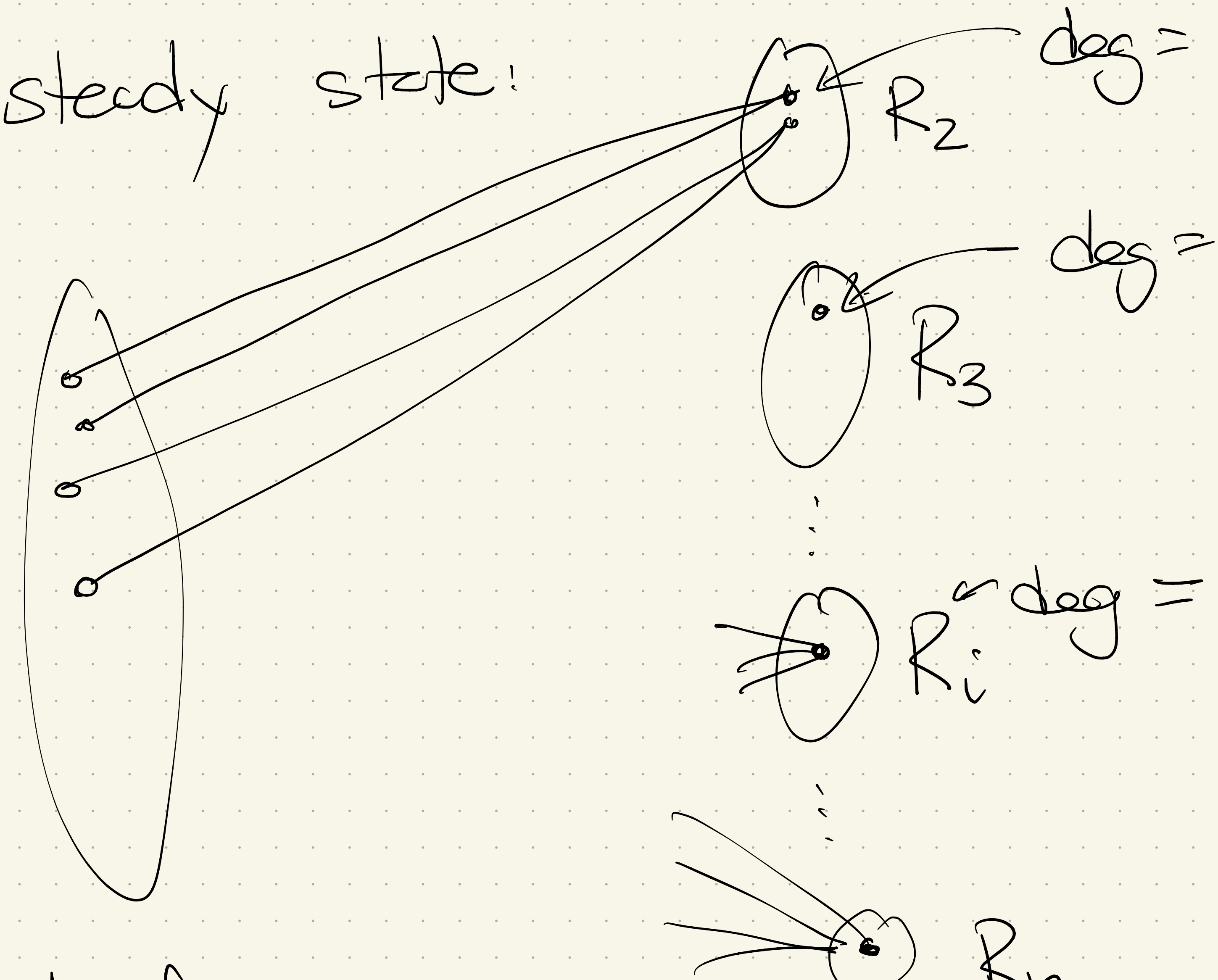


$R =$

For R: for each $i \in 2..n$, add $\lfloor \frac{n}{i} \rfloor$ vertices, each degree i, & connect to <u>different</u> vertices in L.

↳ call these $R_i \subseteq R$

In steady state:



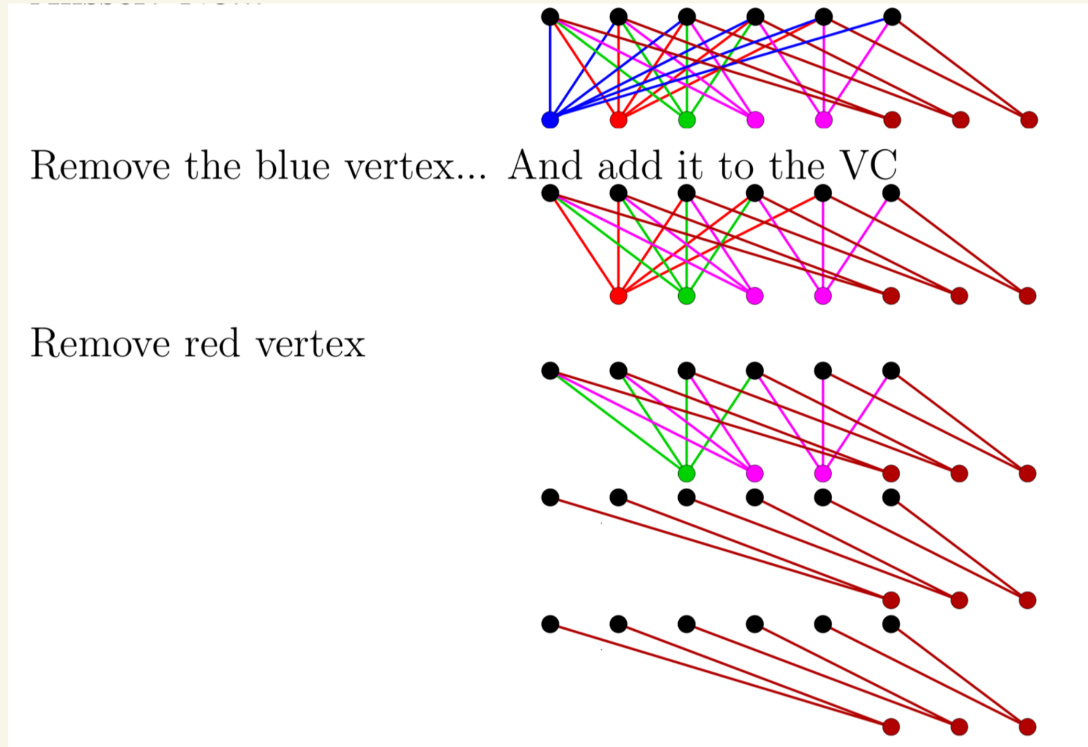
L of size n,

max degree ≤

R

What does our algorithm do?

```
GreedyVertexCover(G):
  C ← ∅
  while G has at least one edge
    v ← vertex in G with maximum degree
    G ← G \ v
    C ← C ∪ v
  return C
```



Highest degree vertex?

↳ in R, <u>one</u> of degree n.

When removed:

So, in end, all R vertices chosen.

What is $|R|$?

$$|R| = \sum_{i=2}^{n} |R_i| = \sum_{i=2}^{n} \left\lfloor \frac{n}{i} \right\rfloor$$

$$\geq$$

Recall that "cheat sheet":

Harmonic series:

$$H_n = \sum_{i=1}^{n} \frac{1}{i},$$

&

Harmonic numbers:

$$1,\ \frac{3}{2},\ \frac{11}{6},\ \frac{25}{12},\ \frac{137}{60},\ \frac{49}{20},\ \frac{363}{140},\ \frac{761}{280},\ \frac{71}{25}$$

$$\ln n < H_n < \ln n + 1,$$

$$H_n = \ln n + \gamma + O\left(\frac{1}{n}\right).$$

So, back to $\alpha(n)$ stuff:

$$|R| \geq n(H_n - 2)$$

$$|L| = n$$

so, greedy factor $\alpha(n) \geq \dfrac{|R|}{|L|}$

$$\& \quad \frac{|R|}{|L|} \geq$$

Note: lower bound! Can we show it always gets at least this?

Theorem Greedy algorithm <u>always</u> chooses a set of size $\leq (\log n) \cdot OPT$

To prove: Rewrite slightly:

GREEDYVERTEXCOVER(G):
$C \leftarrow \varnothing$
$G_0 \leftarrow G$
$i \leftarrow 0$
while G_i has at least one edge
  $i \leftarrow i + 1$
  $v_i \leftarrow$ vertex in G_{i-1} with maximum degree
  $d_i \leftarrow \deg_{G_{i-1}}(v_i)$
  $G_i \leftarrow G_{i-1} \setminus v_i$
  $C \leftarrow C \cup v_i$
return C

Let G_i = graph in i^{th} iteration.

Let d_i = max degree in G_i

Let C^* = optimal vertex cover in G (which must exist but which we don't know)

We do know that C^* is a vertex cover for each G_i.

So:

$$\sum_{v \in C^*} \text{degree of } v \text{ in } G_i \geq \# \text{ edges in } G_i$$

Why?

Since $\sum_{v \in C^*} \deg_{G_i}(v) \geq |E(G_i)|$

$\Rightarrow$ average degree in G_i of C^* is $\geq \frac{|E(G_i)|}{|C^*|}$

Why?

But: this means max degree in G_i is at least this size.

$$\Rightarrow d_i \geq \frac{|E(G_i)|}{|C^*|} = \frac{|E(G_i)|}{OPT}$$

Also: # of edges in G_i decreases

$$d_i \geq \frac{|E(G_i)|}{OPT} \geq \frac{|E(G_j)|}{OPT}$$

for $j \geq i$

Now, consider first OPT iterations of loop:

$$G_{\underline{1}} \to G_1 \to G_2 \to \cdots \to G_{OPT}$$

How many edges get removed?

$$\sum_{i=1}^{OPT} d_i \geq$$

So: $\sum_{i=1}^{OPT} d_i \geq |E(G_{OPT})|$

But: $|E(G_{OPT})| \geq |E(G)| - \sum_{i=1}^{OPT} d_i$

Why?

Crazy sums: $\sum_{i=1}^{OPT} d_i \geq |E(G)| - \sum_{i=1}^{OPT} d_i$

In other words:
OPT iterations removes at least half the edges.

$$|E| \rightarrow \frac{|E|}{2} \rightarrow$$

Keep going: OPT iterations more

How many times?
After $\log(|E|)$ rounds, done.
How many per round?

Runtime & space:

A different approximation – simpler idea:

- pick any edge + add its endpoints to the cover
- delete all "covered" edges
- Repeat

```
DUMBVERTEXCOVER(G):
  C ← ∅
  while G has at least one edge
      (u, v) ← any edge in G
      G ← G \ {u, v}
      C ← C ∪ {u, v}
  return C
```

Seems worse, right?

Theorem
Dumb vertex cover is a 2-approximation.

Proof Let C be greedy cover here,
& C^* be OPT.

For each edge $e = \{uv\}$:

Hub?