CS 344

LECTURE 10

MORE GREEDY ALGORITHMS

- Huffman encoding
- Entropy
- Horn formulae
- Set cover

HUFFMAN ENCODING

MP3 AUDIO

- Sample a song to get real numbers $s_1, s_2, \ldots, s_T$
  - e.g., 44,100 samples per second
- Quantize each sample s_i
  - approximate each sample from a finite set Γ
- Convert string of length T over alphabet Γ to binary

Last step requires a mapping – Huffman encoding!

For example, let $T = 130$ million, $\Gamma = \{A, B, C, D\}$

One possible encoding:

Symbol	Codeword
A	00
B	01
C	10
D	11

Requires 260 million bits (260 megabits or 260 Mb)

Suppose not all symbols appear equally often:

Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

Could we use a variable-length encoding to make A shorter, at the expense of longer encodings for B, C, and D?

First attempt:

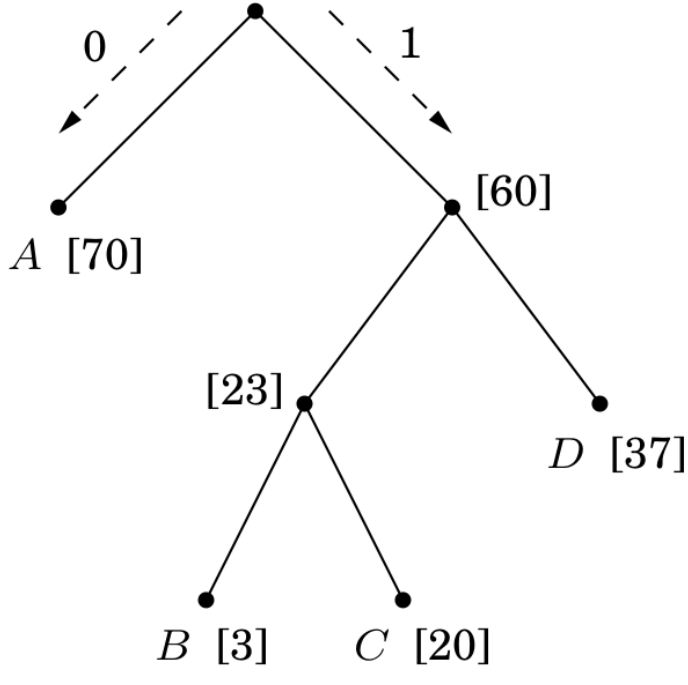
Symbol	Codeword
A	0
B	01
C	11
D	001

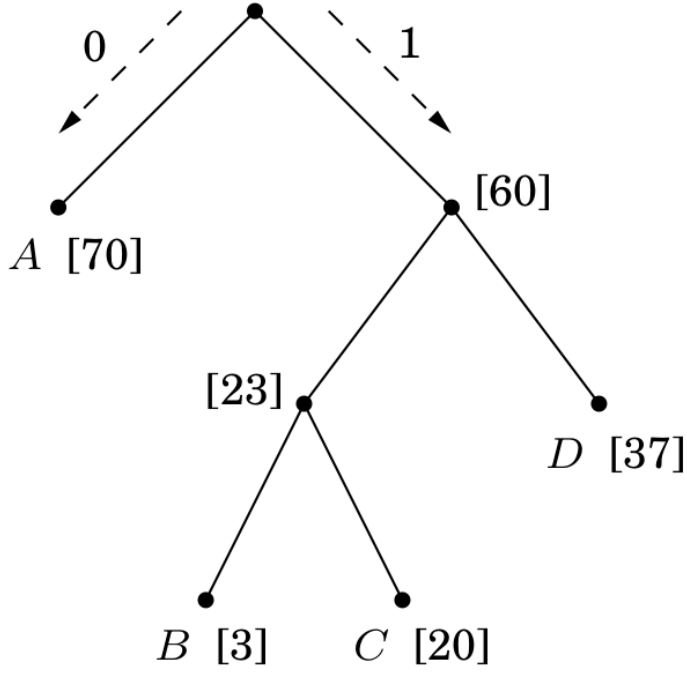
Then suppose we have to decode "001"...

We require that the encoding be **prefix-free**: no codeword can be a prefix of another.

Symbol	Codeword
A	0
B	100
C	101
D	11

Any prefix-free encoding can be drawn as a **full** binary tree (every node has 0 or 2 children)





How do we decode "0100010101101000"?

0 100 0 101 0 11 0 100 0

A B A C A D A B A

When every codeword had length 2, it took 260 Mb to encode 130 million symbols.

With the new encoding:

```
A: 70 million * 1 bit  = 70 Mb
B:  3 million * 3 bits =  9 Mb
C: 20 million * 3 bits = 60 Mb
D: 37 million * 2 bits = 74 Mb
```

Total: 213 Mb!

How do we find an optimal coding tree?

Minimize cost:

$$\sum_{i=1}^{n} f_i \cdot (\text{depth of } i\text{th symbol in tree})$$

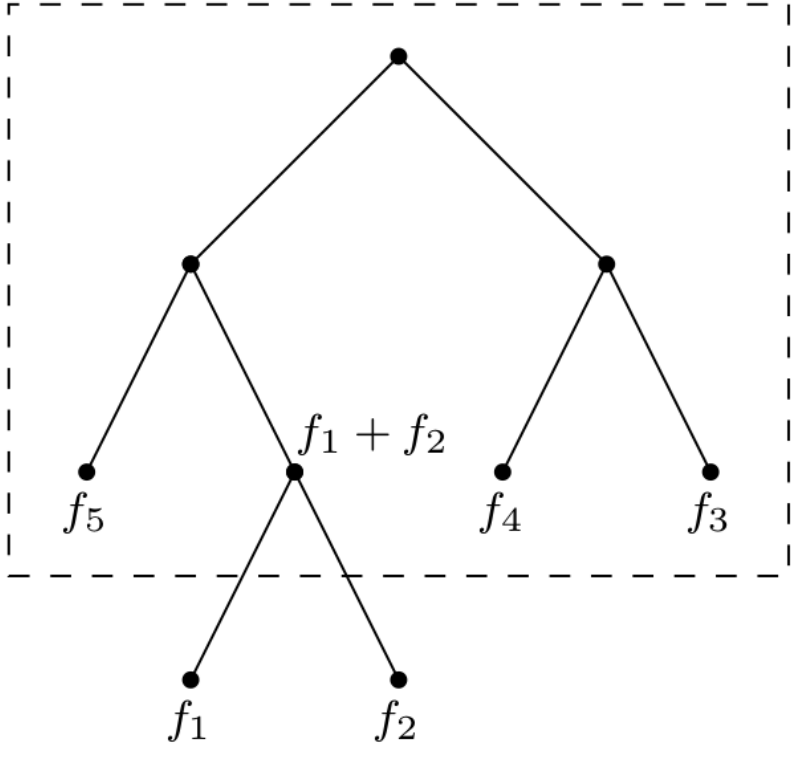
Another way to look at this:

- the frequency of each internal node is the sum of its children's frequencies
- each time we traverse an edge in the tree, we output a bit
- cost of a tree is the sum of the frequency of all non-root nodes

Observation

The two least frequent symbols must be at the bottom of the tree

This suggests a greedy algorithm:



- Take two smallest with frequencies f_i and f_j
- Make them children of a new node
- New node has frequency $f_i + f_j$
- Repeat

```
procedure Huffman(f)
Input:  An array f[1···n] of frequencies
Output:  An encoding tree with n leaves

let H be a priority queue of integers, ordered by f
for i = 1 to n:  insert(H, i)
for k = n + 1 to 2n − 1:
  i = deletemin(H),  j = deletemin(H)
  create a node numbered k with children i, j
  f[k] = f[i] + f[j]
  insert(H, k)
```

How long does it take to run Huffman?

If we use a binary heap,

- n inserts: $n \cdot O(\log n)$
- n iterations of k loop: $n(3 \cdot O(\log n))$

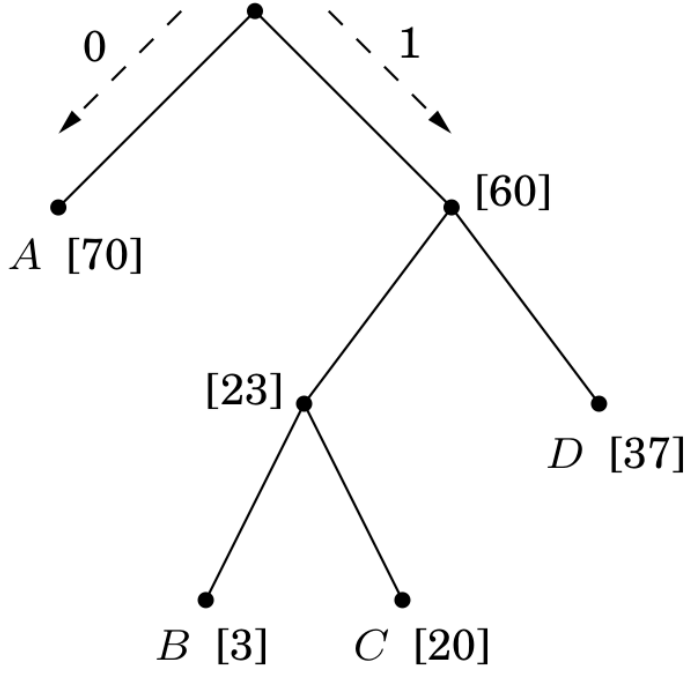
Total: $O(n \log n)$

Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

B + C = 23 million

B/C + D = 60 million

B/C/D + A = 130 million



B + C = 23 million

B/C + D = 60 million

B/C/D + A = 130 million

ASIDE: ENTROPY

From 200 horse races, we determine these probabilities:

Outcome	Aurora	Whirlwind	Phantasm
first	0.15	0.30	0.20
second	0.10	0.05	0.30
third	0.70	0.25	0.30
other	0.05	0.40	0.20

Which is most predictable?

Consider listening to two communications channels:

```
Channel 1: 00000001000000001001000000000...
Channel 2: 10101000101011011010111010...
```

Which is more predictable?

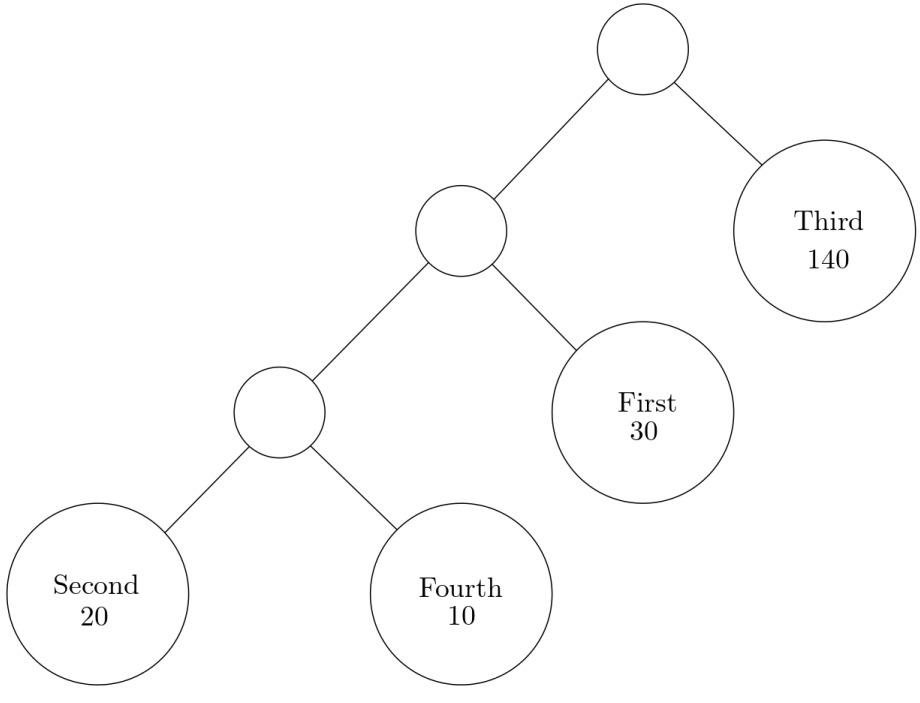
More compressible = less random = more predictable

There were 200 races, so e.g., Aurora has

Outcome	Percent	Count
first	0.15	30
second	0.10	20
third	0.70	140
other	0.05	10

How many bits would it take to encode the sequence of Aurora's wins?

Use Huffman encoding!



Second+Fourth = 30

First+2nd/4th = 60

Third = 140

```
30 first  * 2 bits = 60 bits
20 second * 3 bits = 60 bits
140 third * 1 bit  = 140 bits
10 fourth * 3 bits = 30 bits
```

Total: 290 bits for Aurora

Doing the same for the other two horses, we get

Aurora	290
Whirlwind	380
Phantasm	420

So Aurora is the most predictable.

If there are n outcomes with probabilities

$$p_1, p_2, \ldots, p_n,$$

and we draw m (a large number) of values, then the ith will appear roughly mp_i times.

Then the number of bits needed to encode the sequence is:

$$\sum_{i=1}^{n} mp_i \lg(1/p_i)$$

And the average number of bits for a single value is:

$$\sum_{i=1}^{n} p_i \lg\left(\frac{1}{p_i}\right)$$

This is called the distribution's **entropy**.

For example, flipping a fair coin:

$$\frac{1}{2}\lg 2 + \frac{1}{2}\lg 2 = 1$$

But for a biased coin flip (75% heads, 25% tails):

$$\frac{3}{4}\lg\frac{4}{3}+\frac{1}{4}\lg 4\approx 0.81$$

Fewer bits are required, since this sequence is more predictable.

HORN FORMULAE

Horn formulas are made up of

- Boolean variables
- Horn clauses
  - Implication
  - Negative clause

Implication:

$$(z \wedge w) \Rightarrow u$$

Facts are written with an empty LHS:

$$\Rightarrow u$$

Negative clauses:

$$(\bar{u} \vee \bar{v} \vee \bar{y})$$

Given a set of clauses, find an assignment of true/false values to variables that makes all clauses true.

(called a **satisfying** assignment)

Notice that:

- Implications want things to be true
- Negative clauses want things to be false

A greedy strategy:

- set everything to false
- change some variables to true
  (if necessary for some implication)
- once implications are satisfied, check negative clauses

Greedy algorithm for Horn formulae:

```
set all variables to false

while there is an implication that is not satisfied:
    set the right-hand variable of the implication to true

if all pure negative clauses are satisfied:
    return the assignment
else:
    return "formula is not satisfiable"
```

For example, suppose we have these clauses

$$(w \wedge y \wedge z) \Rightarrow x,$$
$$(x \wedge z) \Rightarrow w,$$
$$x \Rightarrow y,$$
$$\Rightarrow x,$$
$$(x \wedge y) \Rightarrow w,$$
$$(\bar{w} \vee \bar{x} \vee \bar{z}),$$
$$(\bar{z})$$

Initially, everything is false

- x must be true due to $\Rightarrow x$
- y must be true due to $x \Rightarrow y$
- w must be true due to $(x \wedge y) \Rightarrow w$

Result: $\{w : T, x : T, y : T, z : F\}$

$$(w \wedge y \wedge z) \Rightarrow x,$$
$$(x \wedge z) \Rightarrow w,$$
$$x \Rightarrow y,$$
$$\Rightarrow x,$$
$$(x \wedge y) \Rightarrow w,$$
$$(\bar{w} \vee \bar{x} \vee \bar{y}),$$
$$(\bar{z})$$

Initially, everything is false

- x must be true due to $\Rightarrow x$
- y must be true due to $x \Rightarrow y$
- w must be true due to $(x \wedge y) \Rightarrow w$

Result: cannot be satisfied

Why does this work?

Invariant:

If the algorithm sets some variables to true, then they must be true in any satisfying assignment.

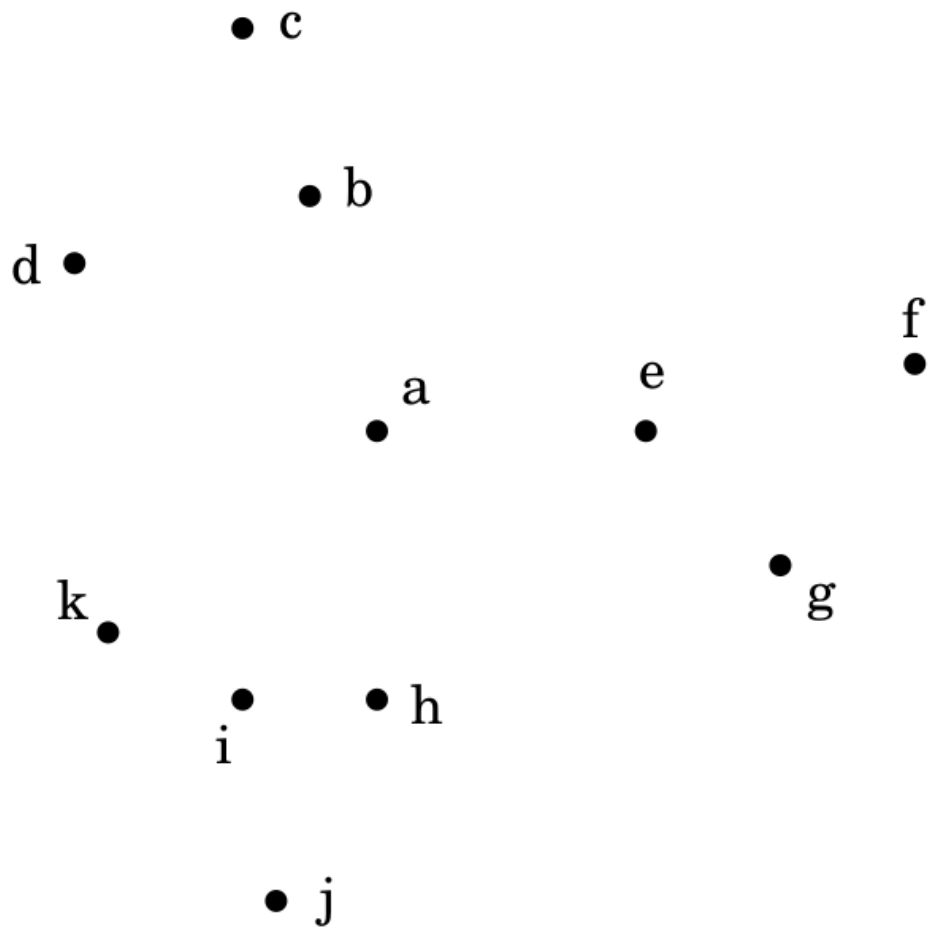
If the negative clauses aren't satisfied with that truth assignment, there cannot be any satisfying truth assignment.

SET COVERS

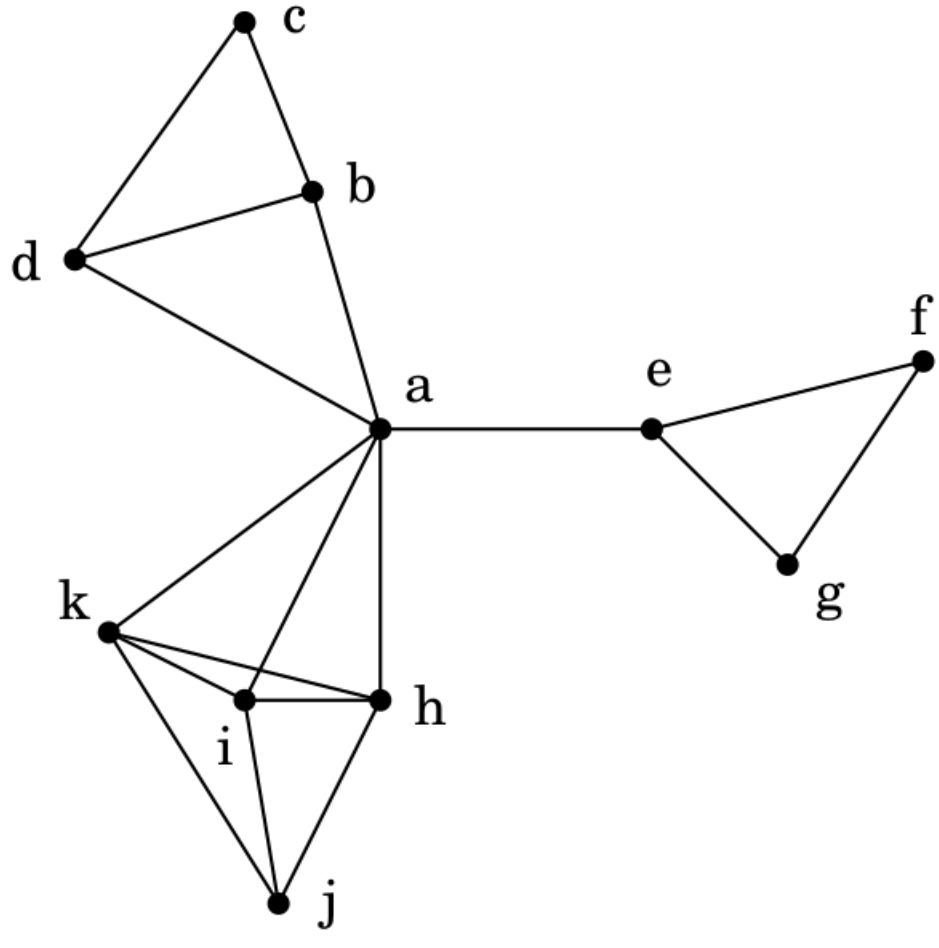
Suppose we want to build schools in some towns, so that all people are within 30 miles of a school.

What is the minimum number of schools we need?

Suppose we had these towns:



Connect the towns within 30 miles of each other:



Then the problem becomes finding a set of vertices that **cover** all vertices in the graph.

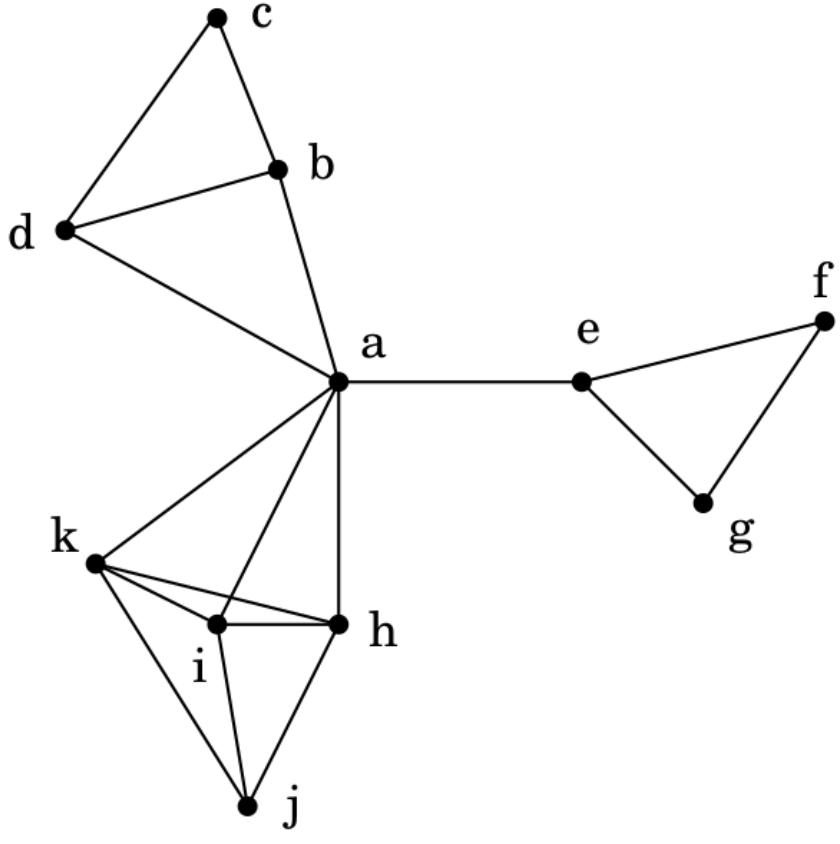
SET COVERING

Let S_v be the set of vertices adjacent to vertex v (including v itself).

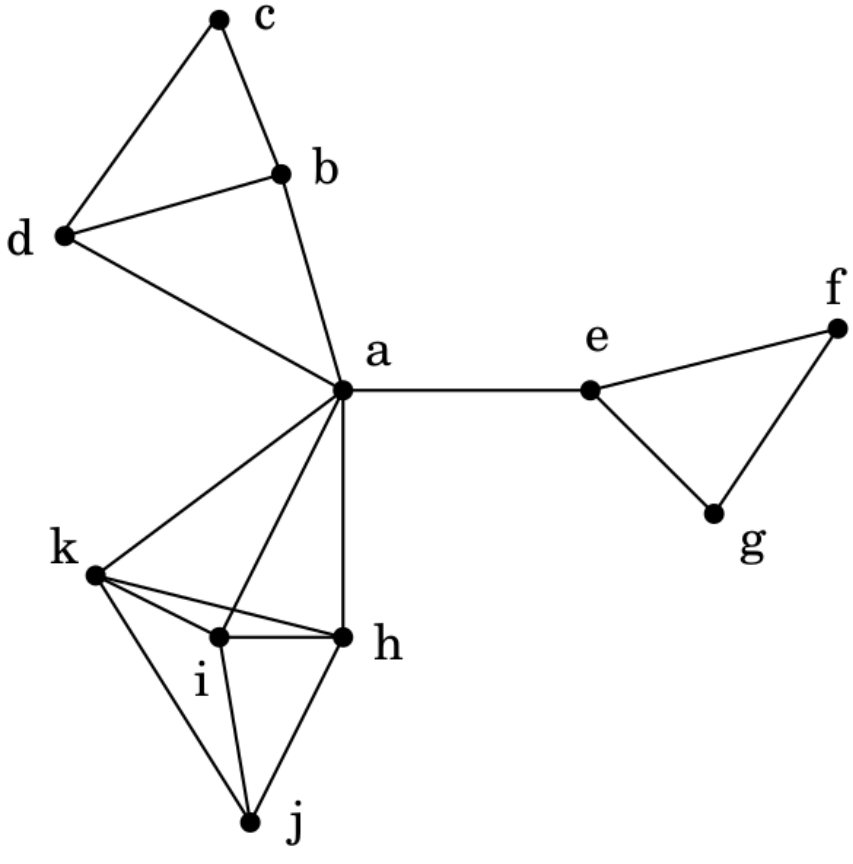
Given V and $S_1, S_2, \ldots, S_m$, find a minimal selection of S_i whose union is B

A greedy approach:

- Until everything in B is covered:
  - Pick the S_i with the largest number of uncovered elements

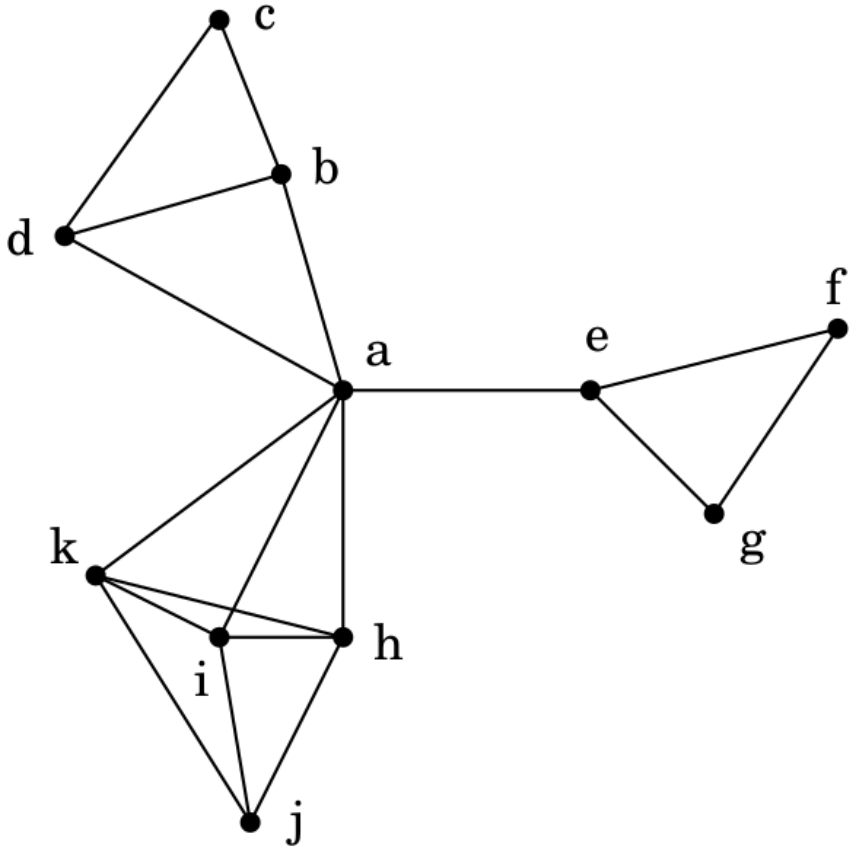


- $S_a = \{a, b, d, e, k, h, i\}$
- $S_b = \{a, b, c, d\}$
- $S_c = \{b, c, d\}$
- $S_d = \{a, b, c, d\}$
- $S_e = \{a, e, f, g\}$
- ...



We pick these towns:

a, c, j, and f (or g)



But consider these instead:

b, e, i

The greedy algorithm is not optimal!

But it's not too bad...

Claim:

- if B has n elements,
- and the optimal cover has k sets,
- then the greedy algorithm uses at most $k \ln n$ sets.

Why is this true?

After t iterations, say n_t elements are still not covered.

(and $n_0 = n$)

We know the optimal k sets cover these,

so there must be some set with at least n_t / k of these.
(Pigeonhole principle)

Therefore, we have that

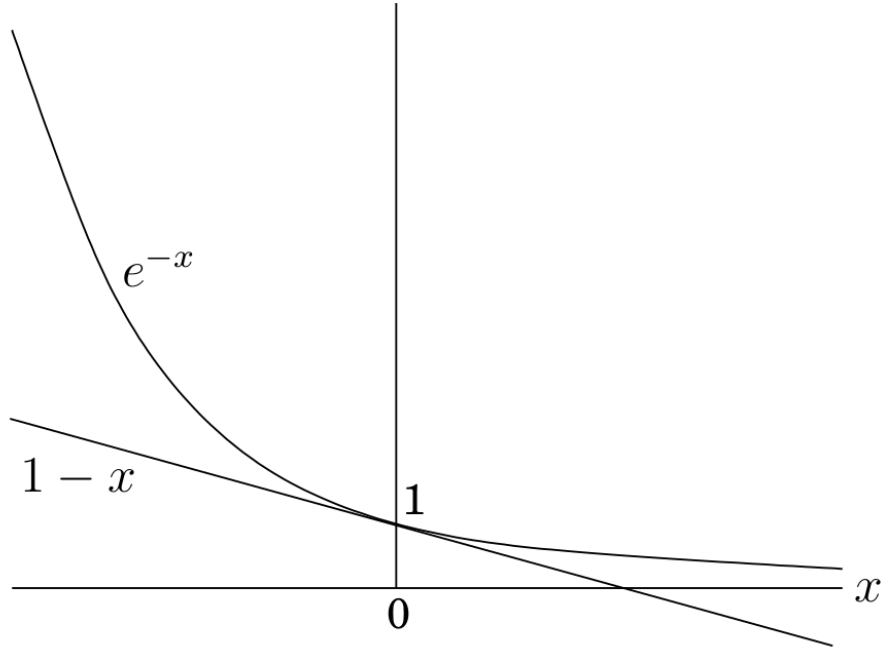
$$n_{t+1} \leq n_t - \frac{n_t}{k} = n_t \left(1 - \frac{1}{k}\right)$$

which, when repeated, implies

$$n_t \leq n_0 \left(1 - \frac{1}{k}\right)^t$$

Aside: a useful inequality

$1 - x \leq e^{-x}$ (and equal when $x = 0$)



Using the inequality $1 - x < e^{-x}$ (for $x \neq 0$), we get

$$\begin{aligned} n_t &\leq n_0 \left(1 - \frac{1}{k}\right)^t \\ &< n_0 (e^{-1/k})^t \\ &= n e^{-t/k} \end{aligned}$$

Given $n_t < ne^{-t/k}$, let $t = k \ln n$. Then

$$n_t < ne^{-\ln n} = \frac{n}{n} = 1$$

so no elements are left uncovered.