ساختمان داده و الگوریتم ها (CE203)

جلسه بیست و یکم:
نمونه های دیگر الگوریتمهای حریصانه

سجاد شیرعلی شهرضا

پاییز 1401

دوشنبه، 12 دی 1401

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 16

زمانبندی

یک مسئله پیچیده ترکه راه حل حریصانه دارد!

SCHEDULING: THE TASK

Input: A set of n jobs. Job **i** takes $\mathbf{t_i}$ hours. For every hour until job **i** is done, pay $\mathbf{c_i}$.
Output: An order of jobs to complete s.t. you minimize the cost.

Homework	**Time:** 5 hours.	**Cost:** 1 units/hr until it's done
Sleep	**Time:** 8 hours.	**Cost:** 5 units/hr until it's done
Laundry	**Time:** 4 hours.	**Cost:** 2 units/hr until it's done

SCHEDULING: THE TASK

Input: A set of n jobs. Job **i** takes $\mathbf{t_i}$ hours. For every hour until job **i** is done, pay $\mathbf{c_i}$.
Output: An order of jobs to complete s.t. you minimize the cost.

Homework	**Time:** 5 hours.	**Cost:** 1 units/hr until it's done
Sleep	**Time:** 8 hours.	**Cost:** 5 units/hr until it's done
Laundry	**Time:** 4 hours.	**Cost:** 2 units/hr until it's done

Homework | Sleep | Laundry — costs $\mathbf{(5 \cdot 1) + (13 \cdot 5) + (17 \cdot 2) = 104}$ **units**

Sleep | Homework | Laundry — costs $\mathbf{(8 \cdot 5) + (13 \cdot 1) + (17 \cdot 2) = 87}$ **units**

Sleep | Laundry | Homework — costs $\mathbf{(8 \cdot 5) + (12 \cdot 2) + (17 \cdot 1) = 81}$ **units**

SCHEDULING: THE TASK

Input: A set of n jobs. Job **i** takes $\mathbf{t_i}$ hours. For every hour until job **i** is done, pay $\mathbf{c_i}$.
Output: An order of jobs to complete s.t. you minimize the cost.

This problem has an optimal substructure!

Suppose this is an optimal schedule:

Job A **Job B** **Job C** **Job D**

This must be the optimal schedule on just jobs B, C, and D.

(If not, then rearranging B, C, D could make a better overall schedule than (A,B,C,D)!

SCHEDULING: THE TASK

Input: A set of n jobs. Job **i** takes $\mathbf{t_i}$ hours. For every hour until job **i** is done, pay $\mathbf{c_i}$.
Output: An order of jobs to complete s.t. you minimize the cost.

This problem has an optimal substructure!

Suppose this is an optimal schedule:

Job A	Job B	Job C	Job D

This must be the optimal schedule on just jobs B, C, and D.

(If not, then rearranging B, C, D could make a better overall schedule than (A,B,C,D)!

A greedy algorithm could greedily commit to the "best" job to do first, and then move on, repeatedly picking the next "best" job. **What would be the "best" job to do first?**

SCHEDULING: EASIER VERSION #1

What would be the "best" job to do first?

Thinking about time lengths & costs together feels a bit complicated… To get some intuition about how they relate to each other, let's brainstorm with a simpler version of the scheduling problem first:

SIMPLIFIED VERSION #1

Input: A set of n tasks. Each task takes 1 hour. For every hour until task **i** is done, pay $\mathbf{c_i}$.

Output: An order of tasks to complete s.t. you minimize the cost.

Job A	Job B	Job C	Job D
Cost/hr: **5**	Cost/hr: **2**	Cost/hr: **1**	Cost/hr: **10**

Which jobs should we do first?

A) Do higher-cost jobs first

B) Do lower-cost jobs first

SCHEDULING: EASIER VERSION #1

What would be the "best" job to do first?

Thinking about time lengths & costs together feels a bit complicated… To get some intuition about how they relate to each other, let's brainstorm with a simpler version of the scheduling problem first:

SIMPLIFIED VERSION #1

Input: A set of n tasks. Each task takes 1 hour. For every hour until task **i** is done, pay $\mathbf{c_i}$.

Output: An order of tasks to complete s.t. you minimize the cost.

Job A	Job B	Job C	Job D
Cost/hr: **5**	Cost/hr: **2**	Cost/hr: **1**	Cost/hr: **10**

Which jobs should we do first?

A) Do higher-cost jobs first

B) Do lower-cost jobs first

SCHEDULING: EASIER VERSION #1

What would be the "best" job to do first?

Thin... how

Do higher-cost jobs first! Why?

Suppose **A** costs $\mathbf{c_A}$/hr and **B** costs $\mathbf{c_B}$/hr, and $\mathbf{c_A \geq c_B}$ (**A** is higher-cost).
Then **cost(A then B) =** $\mathbf{1c_A + 2c_B}$, and **cost(B then A) =** $\mathbf{1c_B + 2c_A}$.
Since $\mathbf{c_A \geq c_B}$, then we know **cost(A then B) ≤ cost(B then A)**,
so it's cheaper to go with **A** (the higher cost job) before **B**.

Delaying expensive jobs is a bad idea, and it'll be better to get them out of the way first. So if we save the cheapest jobs for last, then even though there are more hours that go by before they get completed, the rate we pay for that delay is lower.

D) Do lower-cost jobs first

SCHEDULING: EASIER VERSION #2

What would be the "best" job to do first?

Here's a different but still simpler version of the scheduling problem:

SIMPLIFIED VERSION #2

Input: A set of n tasks. Task **i** takes $\mathbf{t_i}$ hours. For every hour until task **i** is done, pay **1** unit.

Output: An order of tasks to complete s.t. you minimize the cost.

Job A	Job B	Job C	Job D
Cost/hr: **1**	Cost/hr: **1**	Cost/hr: **1**	Cost/hr: **1**

Which jobs should we do first?

A) Do longer jobs first

B) Do shorter jobs first

SCHEDULING: EASIER VERSION #2

What would be the "best" job to do first?

Here's a different but still simpler version of the scheduling problem:

SIMPLIFIED VERSION #2

Input: A set of n tasks. Task **i** takes $\mathbf{t_i}$ hours. For every hour until task **i** is done, pay **1** unit.

Output: An order of tasks to complete s.t. you minimize the cost.

Job A	Job B	Job C	Job D
Cost/hr: **1**	Cost/hr: **1**	Cost/hr: **1**	Cost/hr: **1**

Which jobs should we do first?

A) Do longer jobs first

B) Do shorter jobs first

SCHEDULING: EASIER VERSION #2

What would be the "best" job to do first?

Do shorter jobs first! Why?

Suppose **A** takes $\mathbf{t_A}$ hours and **B** takes $\mathbf{t_B}$ hours, and $\mathbf{t_A \geq t_B}$ (**A** is longer).
Then **cost(A then B) =** $\mathbf{t_A + (t_A + t_B)}$, and **cost(B then A) =** $\mathbf{t_B + (t_B + t_A)}$.
Since $\mathbf{t_A \geq t_B}$, then we know **cost(A then B) ≥ cost(B then A)**,
so it's cheaper to go with **B** (the shorter job) before **A**.

Basically, doing longer jobs first is a bad idea. A longer job would delay every job that comes after it by a longer amount, so this is why shorter jobs are more attractive here — the shortest jobs adds on a minimal delay for each subsequent job.

SCHEDULING: THE "BEST" JOB

What would be the "best" job to do first?

Since both time & cost can vary in this actual problem, we'd like to combine the best of both versions...

It seems like we prefer **higher-cost** & **shorter** jobs.

So if **A** is higher-cost *and* shorter than **B** (i.e. $c_A \geq c_B$ and $t_A \leq t_B$), then **A** is "better".

But what if neither **A** nor **B** are both higher-cost *and* shorter than the other? Then it's not immediately obvious what the "better" job would be…

We need some way of assigning a "score" to each job, and then we can choose the job with the best score. Higher cost should increase a job's score, while longer time lengths should decrease a job's score.

SCHEDULING: THE “BEST” JOB

We need some way of assigning a “score” to each job, and then we can choose the job with the best score.

Higher cost should increase a job’s score, while longer time lengths should decrease a job’s score.

REASONABLE ATTEMPT #1?

score for job **i** = $\mathbf{cost_i - time_i}$

(higher cost increases score,
longer times decreases score)

SCHEDULING: THE "BEST" JOB

We need some way of assigning a "score" to each job, and then we can choose the job with the best score.

Higher cost should increase a job's score, while longer time lengths should decrease a job's score.

REASONABLE ATTEMPT #1?

score for job **i** = $\mathbf{cost_i - time_i}$

(higher cost increases score,
longer times decreases score)

REASONABLE ATTEMPT #2?

score for job **i** = $\mathbf{cost_i / time_i}$

(higher cost increases score,
longer times decreases score)

Which one works?

SCHEDULING: THE “BEST” JOB

We need some way of assigning a “score” to each job, and then we can choose the job with the best score.

Higher cost should increase a job’s score, while longer time lengths should decrease a job’s score.

REASONABLE ATTEMPT #1?

score for job **i** = $\mathbf{cost_i - time_i}$

(higher cost increases score,
longer times decreases score)

REASONABLE ATTEMPT #2?

score for job **i** = $\mathbf{cost_i / time_i}$

(higher cost increases score,
longer times decreases score)

Consider this example:

Cost/hr: **5**

Job A

time: **3 hours**

Cost/hr: **2**

Job B

time: **1 hour**

SCHEDULING: THE "BEST" JOB

We need some way of assigning a "score" to each job, and then we can choose the job with the best score.

Higher cost should increase a job's score, while longer time lengths should decrease a job's score.

WRONG SCORING SCHEME!

score for job **i** = $\mathbf{cost_i - time_i}$

This says we should do **Job A then Job B**.

This gives us cost: (3 · 5) + (4 · 2) = **23**

PROMISING SCORING SCHEME!

score for job **i** = $\mathbf{cost_i / time_i}$

This says we should do **Job B then Job A**.

This gives us cost: (1 · 2) + (4 · 5) = **22**

Consider this example:

Cost/hr: **5**

Job A

time: **3 hours**

Cost/hr: **2**

Job B

time: **1 hour**

SCHEDULING: THE "BEST" JOB

We need som… he best score.

Higher cost … **job's score.**

WRO… HEME!

score f… time_i

This says … n Job A.

This g…) = **22**

Cons…

exa…

…ur

Why does the ratio matter? For any two tasks A and B:

$$\text{cost}(A\ B) = (t_A \cdot c_A) + ((t_A + t_B) \cdot c_B)$$

$$\text{cost}(B\ A) = (t_B \cdot c_B) + ((t_A + t_B) \cdot c_A)$$

A B is better than B A when $c_B / t_B \leq c_A / t_A$:

$$(t_A \cdot c_A) + ((t_A + t_B) \cdot c_B) \leq (t_B \cdot c_B) + ((t_A + t_B) \cdot c_A)$$

$$(t_A \cdot c_A) + (t_A \cdot c_B) + (t_B \cdot c_B) \leq (t_B \cdot c_B) + (t_A \cdot c_A) + (t_B \cdot c_A)$$

$$t_A \cdot c_B \leq t_B \cdot c_A$$

$$c_B / t_B \leq c_A / t_A$$

SCHEDULING: “PSEUDOCODE”

Our greedy choice: always choose the job with the next biggest ratio:

cost (per hour until finished)

time it takes

SCHEDULING: “PSEUDOCODE”

Our greedy choice: always choose the job with the next biggest ratio:

$$\frac{\textbf{cost (per hour until finished)}}{\textbf{time it takes}}$$

SCHEDULING(n jobs with times & costs):

    Compute cost/time ratios for all jobs
    Sort jobs in descending order of cost/time ratios
    Return sorted jobs!

SCHEDULING: “PSEUDOCODE”

Our greedy choice: always choose the job with the next biggest ratio:

$$\frac{\textbf{cost (per hour until finished)}}{\textbf{time it takes}}$$

SCHEDULING(n jobs with times & costs):
    Compute cost/time ratios for all jobs
    Sort jobs in descending order of cost/time ratios
    Return sorted jobs!

Runtime: ?

SCHEDULING: "PSEUDOCODE"

Our greedy choice: always choose the job with the next biggest ratio:

$$\frac{\textbf{cost (per hour until finished)}}{\textbf{time it takes}}$$

SCHEDULING(n jobs with times & costs):

    Compute cost/time ratios for all jobs
    Sort jobs in descending order of cost/time ratios
    Return sorted jobs!

Runtime: O(n log n)

SCHEDULING: CORRECTNESS

Let's follow our framework from before:

Prove that after each choice, you're not ruling out success.
(i.e. you're not ruling out finding an optimal solution)

- **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 show that you won't rule out success after choice t+1 (let's elaborate on this!)
- **CONCLUSION:** If you reach the end of the algorithm and haven't ruled out success then you must have succeeded

SCHEDULING: CORRECTNESS

Let's follow our framework from before:

Prove that after each choice, you're not ruling out success.
(i.e. you're not ruling out finding an optimal solution)

- **INDUCTIVE HYPOTHESIS:** After greedy choice t, you haven't ruled out success
-
- at
  you won't rule out success after choice t+1 (let's elaborate on this!)

Our greedy choice in Scheduling: **choosing the highest ratio job**

- **CONCLUSION:** If you reach the end of the algorithm and haven't ruled out success then you must have succeeded

SCHEDULING: CORRECTNESS

Inductive Step (sketch): Suppose we've already chosen k jobs, and we haven't ruled out success. Then, if our greedy algorithm chooses the next $(k+1)^{st}$ job to be the one left that maximizes the cost/time ratio, we need to show that we still won't rule out success.

Job E | Job D | Job P | Job B | Job M | Job G | Job C

Already chose some jobs…

CASE 1: T* and our greedy algo. both choose Job B

Suppose there's still some optimal ordering of jobs **T*** that extends the k jobs we already chose.

Say our greedy algorithm chooses some Job B next.
If **T*** also chooses Job B as it's $(k+1)^{st}$ job, then great! **T*** is therefore an optimal solution that extends our k+1 choices, so we haven't ruled out our chances of arriving at an optimal solution.

SCHEDULING: CORRECTNESS

Inductive Step (sketch): Suppose we've already chosen k jobs, and we haven't ruled out success. Then, if our greedy algorithm chooses the next (k+1)st job to be the one left that maximizes the cost/time ratio, we need to show that we still won't rule out success.

On the other hand, if **T*** doesn't choose Job B as it's (k+1)st job, then we'll construct a solution **T** that is also optimal and *does* have B as the next job. Since B has the highest cost/time ratio, then we know we can safely swap it one-by-one with each job before it without increasing the cost of **T*** until Job B ends up as the (k+1)st job. This gives us another optimal solution **T**.

So **T** is an optimal solution that extends our k+1 choices!

SCHEDULING: WHAT DID WE LEARN?

The scheduling problem does have a greedy solution that works!
Always choose the job with the next biggest ratio:

$$\frac{\textbf{cost (per hour until finished)}}{\textbf{time it takes}}$$

Note: This is a harder greedy algorithm to come up with compared to the activity selection algorithm. If it helps, I'd classify the activity selection algorithm as something we'd want you to be able to come up with on HW/exams without many hints, but something like this scheduling problem would not show up as a standard question on an exam. It could be a homework problem, but we'd probably guide you through some ideas or give some hints, since realizing that this ratio is important is a bigger jump to make.

سوال؟

کد هافمن

یک نمونه دیگر از حل حریصانه مسئله

OPTIMAL CODES: THE TASK

ASCII can be pretty wasteful for English sentences, where letters have varying frequencies. If **e** shows up so often, maybe we should have a more efficient way of representing it (e.g. use less bits to represent **e**)!

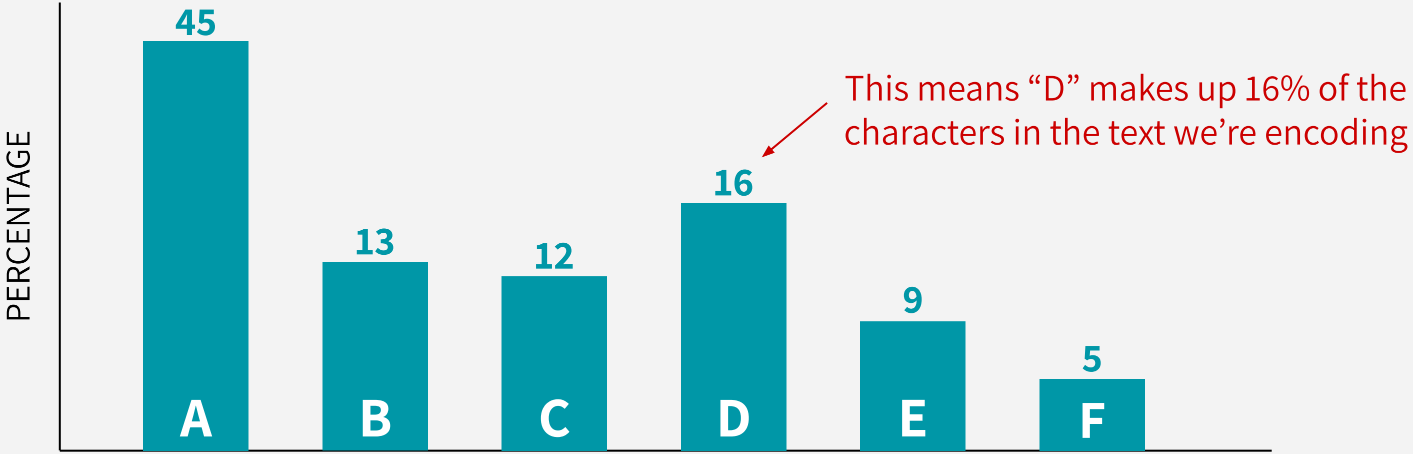
`everyday english sentence`

```
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
```

Input: Some distribution on characters (frequencies of characters)
Output: A way to encode the characters as efficiently* as possible

OPTIMAL CODES: THE TASK

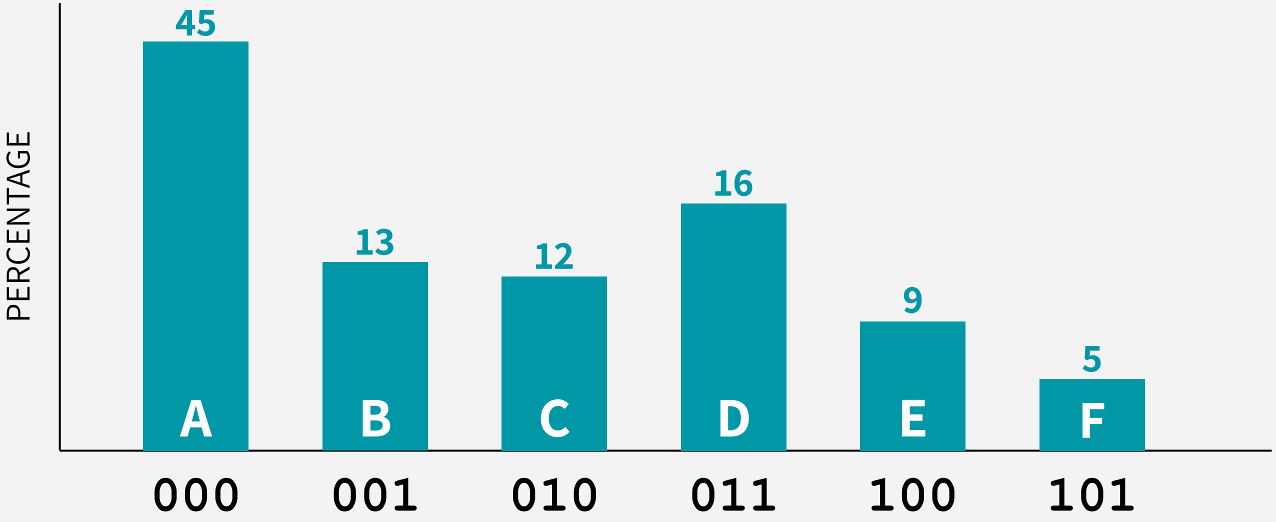
Input: Some distribution on characters (frequencies of characters)
Output: A way to encode the characters as efficiently* as possible



Goal: Minimize the average number of bits used to encode a symbol (with symbols weighted according to their frequencies)

OPTIMAL CODES: ATTEMPT 0

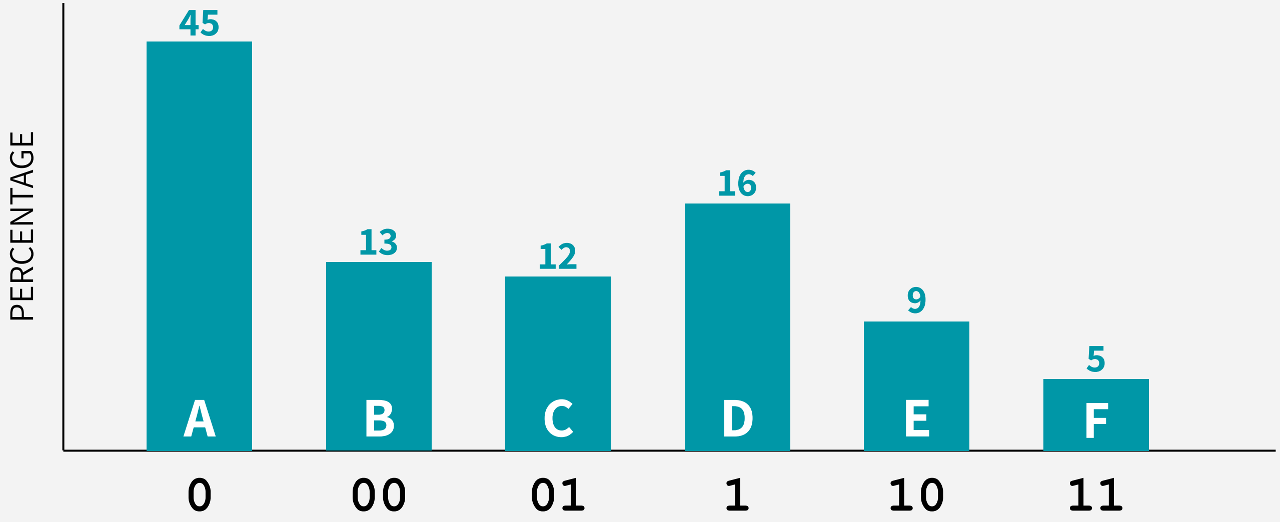
ATTEMPT 0: Use a fixed length code (the **i**th character gets coded as **i** in binary)



We should really try to get away with fewer bits for our more common symbols...

OPTIMAL CODES: ATTEMPT 1

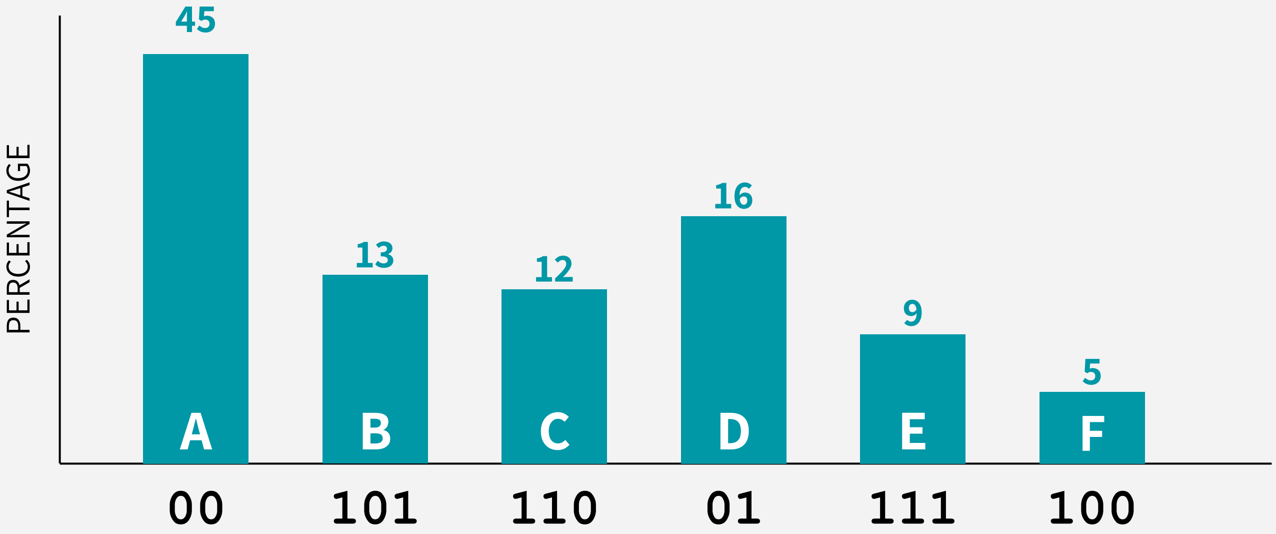
ATTEMPT 1: Use a *variable* length code (shorter codes for common characters)



What is 001? Could be **AC**, or it could be **BD**, or **AAD**… we've introduced ambiguity!

OPTIMAL CODES: ATTEMPT 2

ATTEMPT 2: Use a variable length *prefix-free* code, so that no character's encoding is a prefix of another character's encoding (sometimes called a *prefix code*)

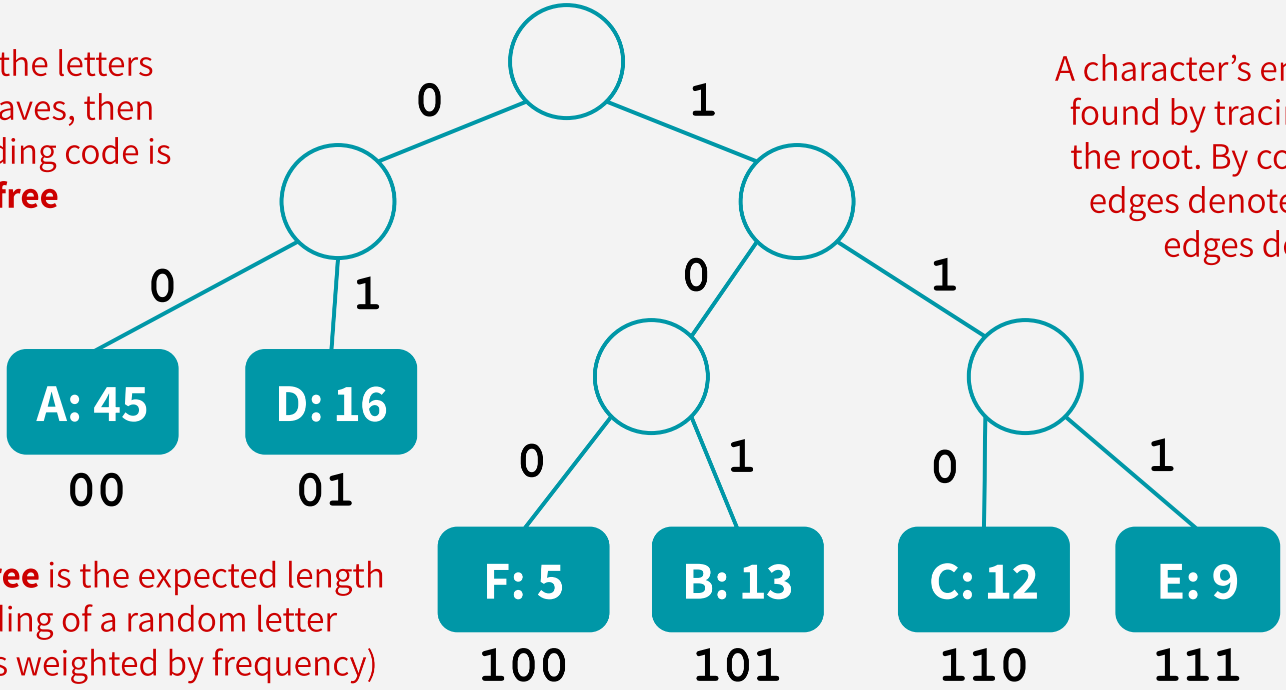


What is 0011001? **A C D**

A PREFIX-FREE CODE IS A TREE

We can think of a prefix-free code as a tree:

As long as all the letters show up as leaves, then the corresponding code is **prefix-free**



A character's encoding can be found by tracing down from the root. By convention, left edges denote 0, and right edges denote 1.

The **cost of a tree** is the expected length of the encoding of a random letter (randomness is weighted by frequency)

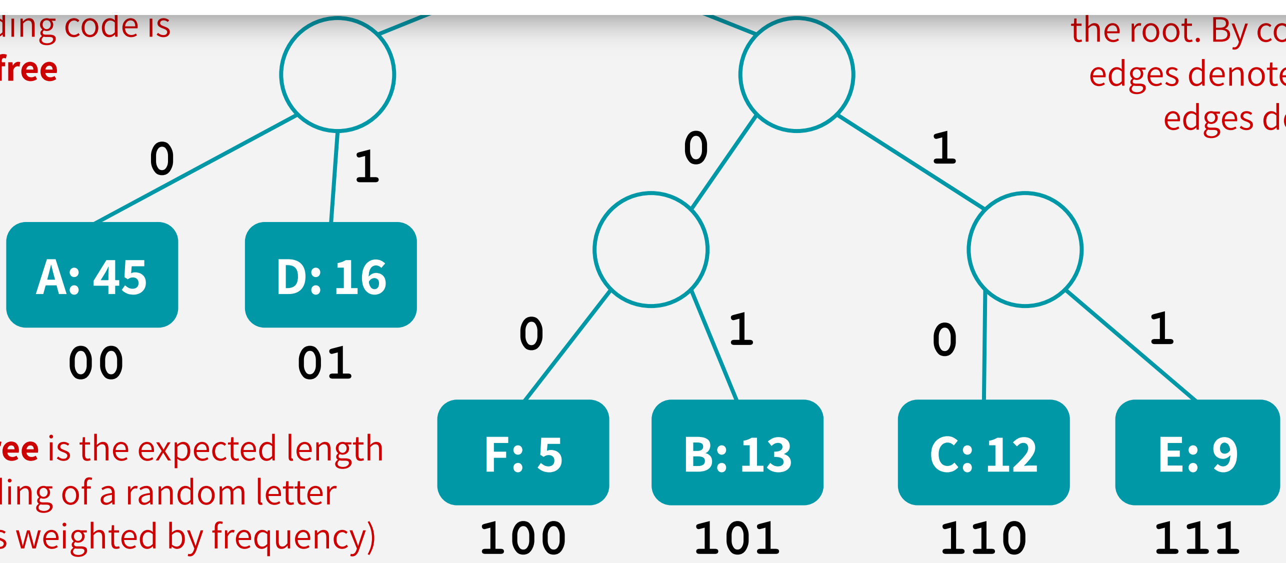
A PREFIX-FREE CODE IS A TREE

Cost of this tree (average leaf depth): $\sum_{\text{leaves } x} P(x) \cdot \text{depth}(x)$

$(2 \cdot 0.45) + (2 \cdot 0.16) + (3 \cdot 0.05) + (3 \cdot 0.13) + (3 \cdot 0.12) + (3 \cdot 0.09) = \mathbf{2.39}$

As lor
show
the corresponding code is
prefix-free

can be
n from
the root. By convention, left
edges denote 0, and right
edges denote 1.



The **cost of a tree** is the expected length of the encoding of a random letter (randomness is weighted by frequency)

A PREFIX-FREE CODE IS A TREE

Cost of this tree (average leaf depth): $\sum_{\text{leaves } x} P(x) \cdot \text{depth}(x)$

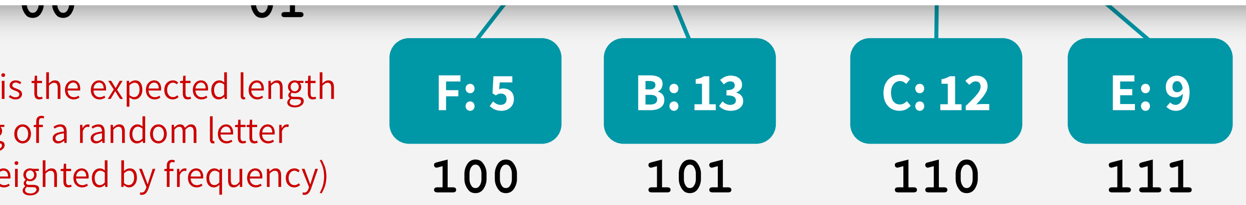
$(2 \cdot 0.45) + (2 \cdot 0.16) + (3 \cdot 0.05) + (3 \cdot 0.13) + (3 \cdot 0.12) + (3 \cdot 0.09) = \mathbf{2.39}$

As lo… show… the corresponding code is **prefix-free**

…can be …n from the root. By convention, left edges denote 0, and right edges denote 1.

Our goal (rephrased in terms of this tree):

Minimize the (weighted) average leaf depth of this binary tree!



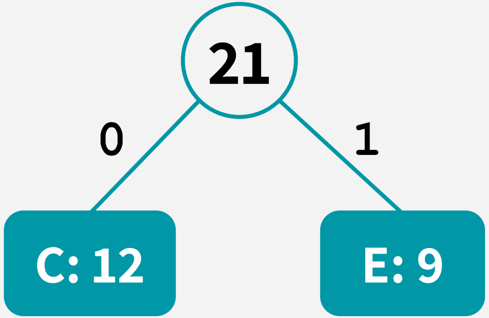
The **cost of a tree** is the expected length of the encoding of a random letter (randomness is weighted by frequency)

HUFFMAN CODING: THE IDEA

IDEA: Greedily build sub-trees from the bottom up, where the "greedy goal" is to have less frequent letters further down in the tree.

To ensure that less frequent letters are further down in the tree, we'll greedily build subtrees, by "merging" the 2 node with the smallest frequency count, and then repeating until we've merged everything!

A **"merge"** between 2 nodes creates a common parent node whose key is the sum of those 2 nodes frequencies:



HUFFMAN CODING: EXAMPLE

Greedily build subtrees by merging, starting with the 2 most infrequent letters.

A: 45

B: 13

C: 12

D: 16

E: 9

F: 5

HUFFMAN CODING: EXAMPLE

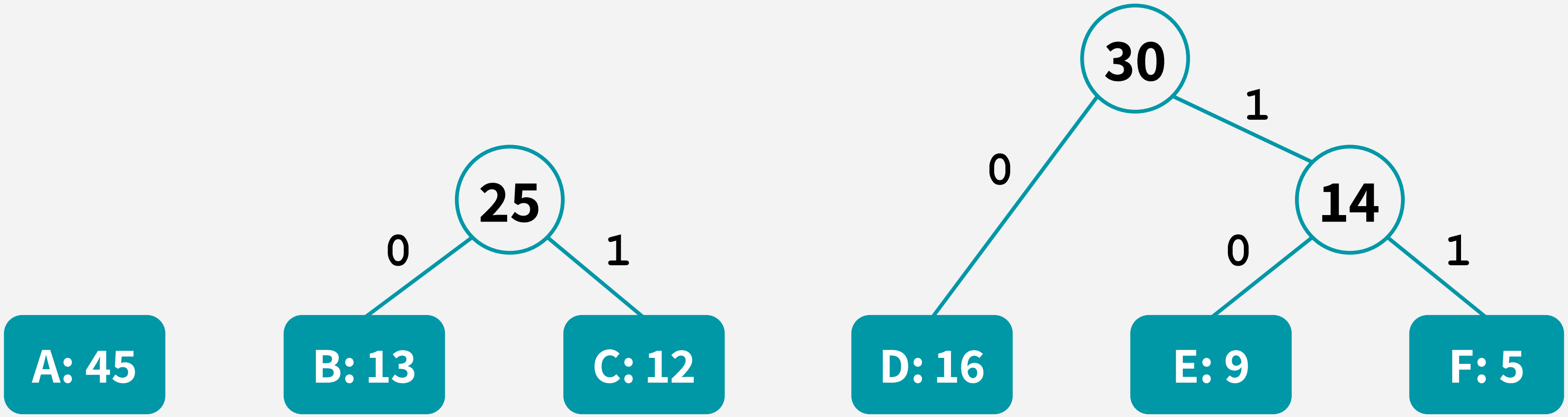
Greedily build subtrees by merging, starting with the 2 most infrequent letters.

HUFFMAN CODING: EXAMPLE

Greedily build subtrees by merging, starting with the 2 most infrequent letters.

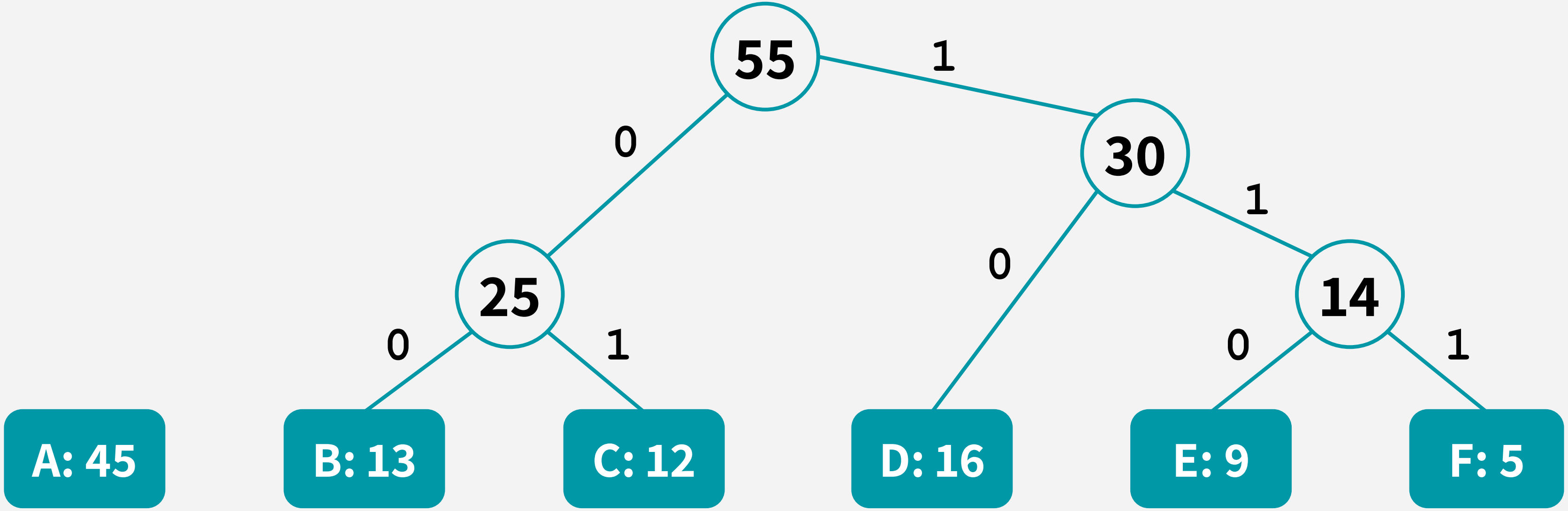
HUFFMAN CODING: EXAMPLE

Greedily build subtrees by merging, starting with the 2 most infrequent letters.



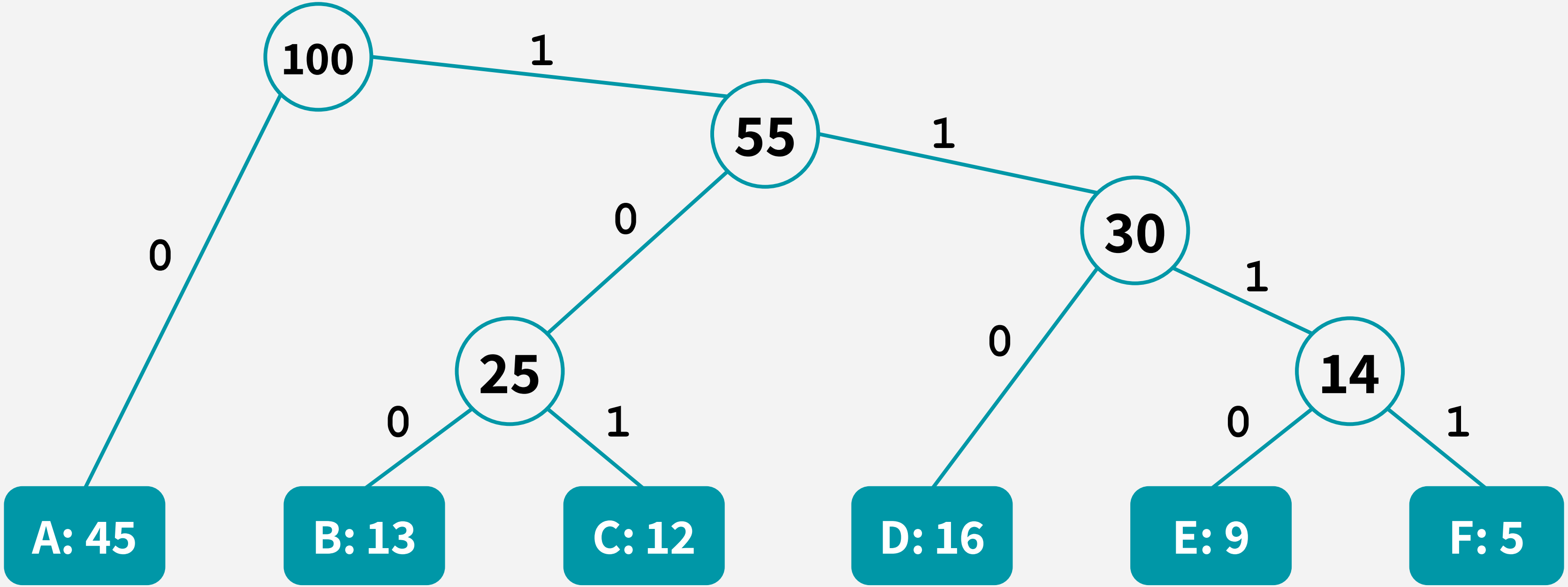
HUFFMAN CODING: EXAMPLE

Greedily build subtrees by merging, starting with the 2 most infrequent letters.



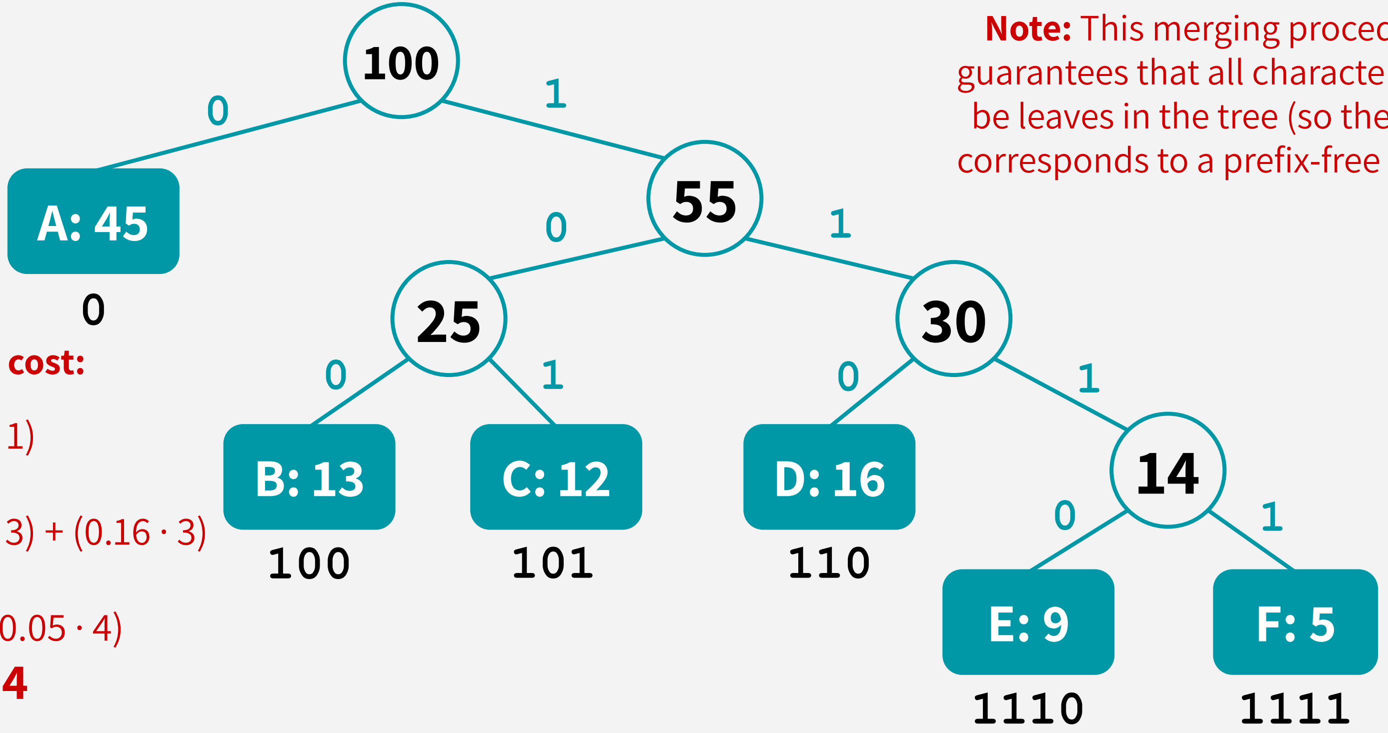
HUFFMAN CODING: EXAMPLE

Greedily build subtrees by merging, starting with the 2 most infrequent letters.



HUFFMAN CODING: EXAMPLE

Greedily build subtrees by merging, starting with the 2 most infrequent letters.



Note: This merging procedure guarantees that all characters will be leaves in the tree (so the tree corresponds to a prefix-free code)

Expected cost:

$$(0.45 \cdot 1) + (0.13 \cdot 3) + (0.12 \cdot 3) + (0.16 \cdot 3) + (0.09 \cdot 4) + (0.05 \cdot 4) = \mathbf{2.24}$$

HUFFMAN CODING: PSEUDOCODE

```
HUFFMAN_CODING(Characters C, Frequencies F):
    Create a node for each character (key is its frequency)
    CURRENT = {set of all these nodes}
    while len(CURRENT) > 1:
        X and Y ← the 2 nodes in CURRENT with the smallest keys
        Create a new node Z with Z.key = X.key + Y.key
        Z.left = X, Z.right = Y
        Add Z to CURRENT, and remove X and Y from CURRENT
    return CURRENT[0]
```

This depends on how we store our set of CURRENT nodes! We need to find minimum nodes and insert nodes.

Runtime: O(n · runtime per iteration)

HUFFMAN CODING: PSEUDOCODE

```
HUFFMAN_CODING(Characters C, Frequencies F):
    Create a node for each character (key is its frequency)
    CURRENT = {set of all these nodes}
    while len(CURRENT) > 1:
        X and Y ← the 2 nodes in CURRENT with the smallest keys
        Create a new node Z with Z.key = X.key + Y.key
        Z.left = X, Z.right = Y
        Add Z to CURRENT, and remove X and Y from CURRENT
    return CURRENT[0]
```

Using a heap data structure or balanced BST to store CURRENT (O(log n) find min & insert)

Runtime: O(n log n)

HUFFMAN CODING: PSEUDOCODE

```
HUFFMAN_CODING(Characters C, Frequencies F):
    Create a node for each character (key is its frequency)
    CURRENT = {set of all these nodes}
    while len(CURRENT) > 1:
        X and Y ← the 2 nodes in CURRENT with the smallest keys
        Create a new node Z with Z.key = X.key + Y.key
        Z.left = X, Z.right = Y
        Add Z to CURRENT, and remove X and Y from CURRENT
    return CURRENT[0]
```

Pre-sorting frequencies using MERGESORT and maintaining 2 queues (can you figure this out?)

Runtime: O(n log n)

HUFFMAN CODING: PSEUDOCODE

```
HUFFMAN_CODING(Characters C, Frequencies F):
    Create a node for each character (key is its frequency)
    CURRENT = {set of all these nodes}
    while len(CURRENT) > 1:
        X and Y ← the 2 nodes in CURRENT with the smallest keys
        Create a new node Z with Z.key = X.key + Y.key
        Z.left = X, Z.right = Y
        Add Z to CURRENT, and remove X and Y from CURRENT
    return CURRENT[0]
```

Pre-sorting frequencies using RADIXSORT (if frequencies are appropriate!!!) and using 2 queues (can you figure this out?)

Runtime: O(n)

HUFFMAN CODING: CORRECTNESS

Let's use our framework! We'll sketch the proof here.

Prove that after each choice, you're not ruling out success.
(i.e. you're not ruling out finding an optimal solution)

- **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 show that you won't rule out success after choice t+1 (let's elaborate on this!)
- **CONCLUSION:** If you reach the end of the algorithm and haven't ruled out success then you must have succeeded

HUFFMAN CODING: CORRECTNESS

Let's use our framework! We'll sketch the proof here.

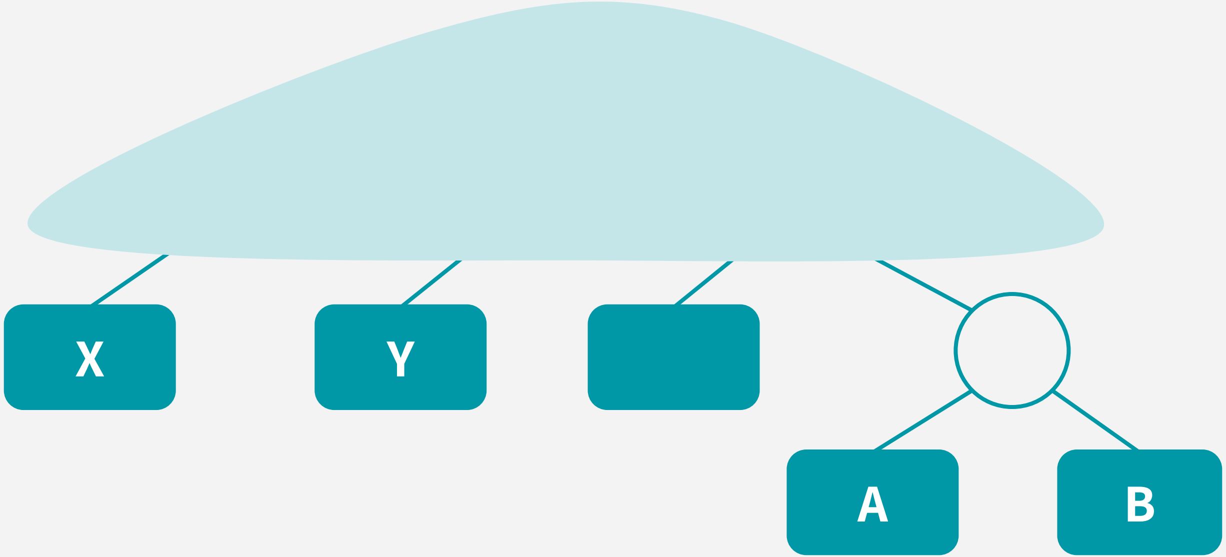
Prove that after each choice, you're not ruling out success.
(i.e. you're not ruling out finding an optimal solution)

- **INDUCTIVE HYPOTHESIS:** After greedy choice t, you haven't ruled out success
- **B**
- **I** … that you won't rule out success after choice t+1 (let's elaborate on this!)
- **CONCLUSION:** If you reach the end of the algorithm and haven't ruled out success then you must have succeeded

Our greedy choice in huffman coding: **merging two nodes**!

HUFFMAN CODING: CORRECTNESS

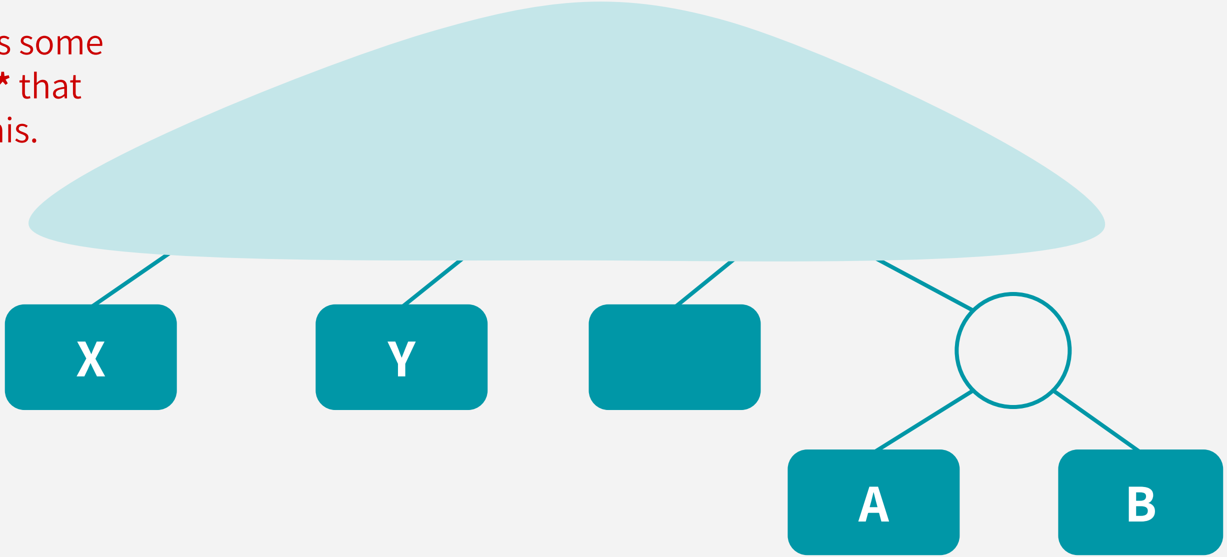
LEMMA (to be used in Inductive step): If X and Y are the two least-frequent letters, then there is an optimal tree where X and Y are *siblings*!
In other words, choosing to merge X and Y doesn't rule out success.



HUFFMAN CODING: CORRECTNESS

LEMMA (to be used in Inductive step): If X and Y are the two least-frequent letters, then there is an optimal tree where X and Y are *siblings*!
In other words, choosing to merge X and Y doesn't rule out success.

Suppose there is some optimal tree **T*** that looks like this.

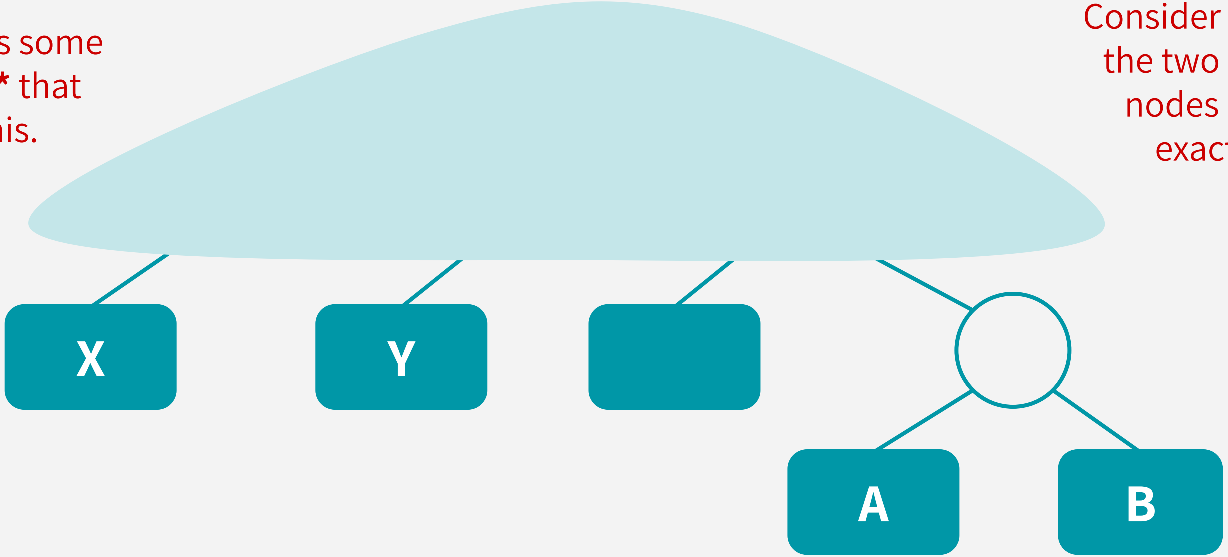


HUFFMAN CODING: CORRECTNESS

LEMMA (to be used in Inductive step): If X and Y are the two least-frequent letters, then there is an optimal tree where X and Y are *siblings*!
In other words, choosing to merge X and Y doesn't rule out success.

Suppose there is some optimal tree **T*** that looks like this.

Consider the case where the two lowest sibling nodes in **T*** are not exactly X and Y.

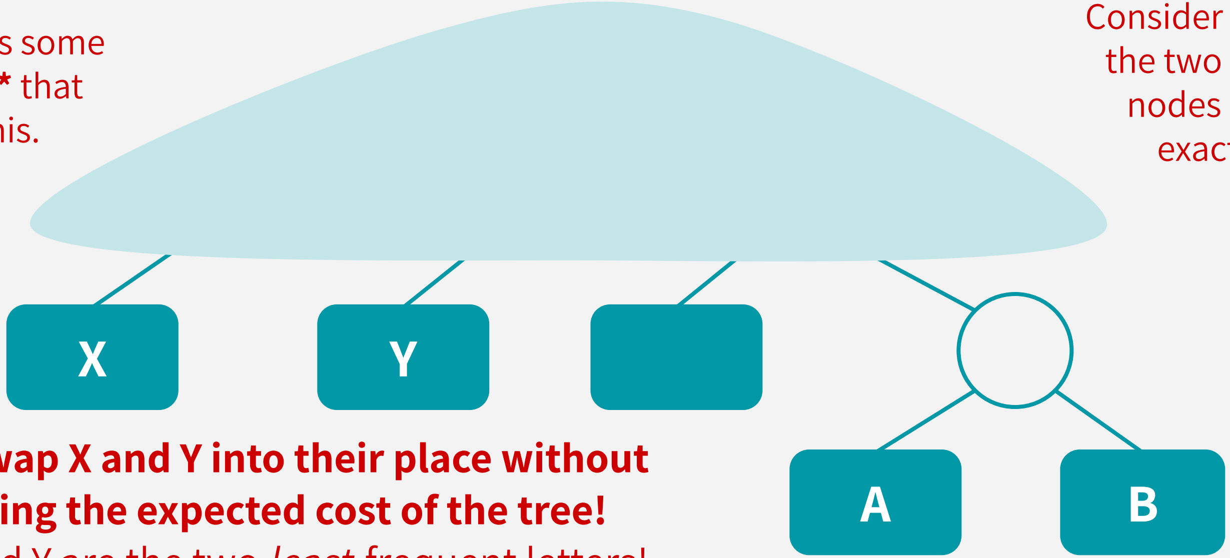


HUFFMAN CODING: CORRECTNESS

LEMMA (to be used in Inductive step): If X and Y are the two least-frequent letters, then there is an optimal tree where X and Y are *siblings*!
In other words, choosing to merge X and Y doesn't rule out success.

Suppose there is some optimal tree **T*** that looks like this.

Consider the case where the two lowest sibling nodes in **T*** are not exactly X and Y.



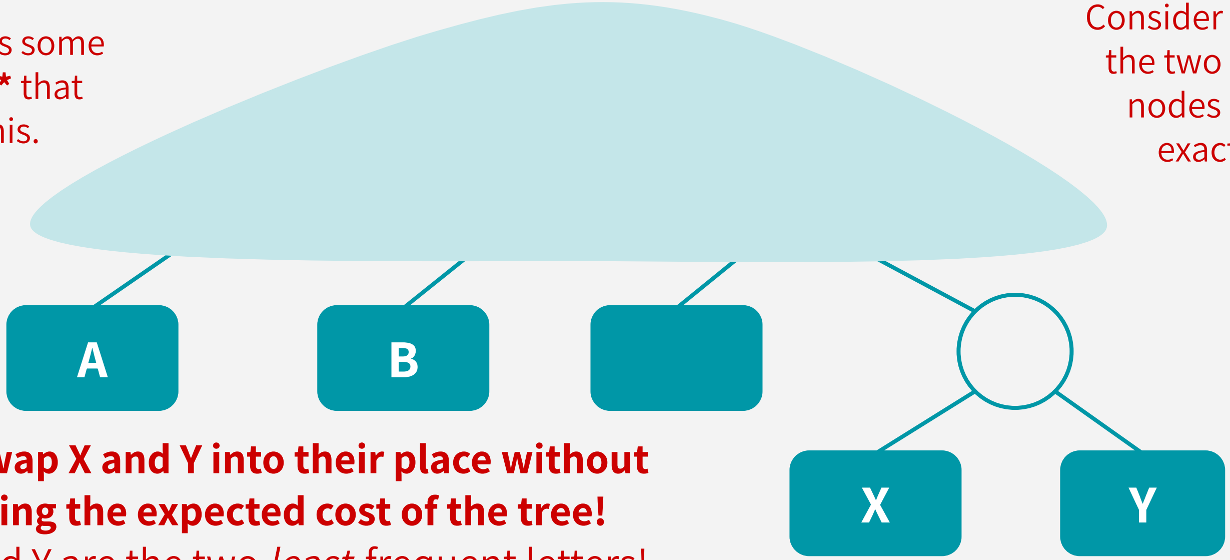
We can swap X and Y into their place without increasing the expected cost of the tree!
Why? X and Y are the two *least*-frequent letters!

HUFFMAN CODING: CORRECTNESS

LEMMA (to be used in Inductive step): If X and Y are the two least-frequent letters, then there is an optimal tree where X and Y are *siblings*!
In other words, choosing to merge X and Y doesn't rule out success.

Suppose there is some optimal tree **T*** that looks like this.

Consider the case where the two lowest sibling nodes in **T*** are not exactly X and Y.



We can swap X and Y into their place without increasing the expected cost of the tree!
Why? X and Y are the two *least*-frequent letters!

HUFFMAN CODING: CORRECTNESS

LEMMA (to be used in Inductive step): If X and Y are the two least-frequent letters, then there is an optimal tree where X and Y are *siblings*!

In other words, choosing to merge X and Y doesn't rule out success.

Suppose th
optimal t
looks l

ase where
st sibling
are not
and Y.

One problem… This seems to be enough to show that we don't rule out optimality on the first step. In later steps, we end up potentially merging nodes that were introduced, rather than just the original character nodes. Will everything still work?

We can swap X and Y into their place without increasing the expected cost of the tree!

Why? X and Y are the two *least*-frequent letters!

HUFFMAN CODING: CORRECTNESS

To show that we continue to not rule out optimality once we start merging nodes that aren't just the original character nodes...

We can basically treat the parent nodes that get created as leaves in a *new alphabet*!

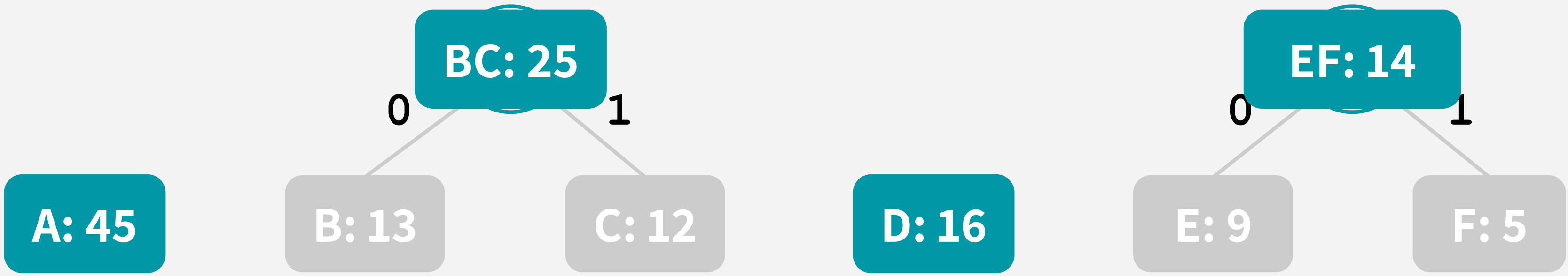
Suppose I've performed these merges so far...

HUFFMAN CODING: CORRECTNESS

To show that we continue to not rule out optimality once we start merging nodes that aren't just the original character nodes…

We can basically treat the parent nodes that get created as leaves in a *new alphabet*!

I can now consider "BC" or "DEF" as new letters in my new alphabet (with their own frequencies)!
This means that our lemma we just showed can still be applied.



HUFFMAN CODING: WHAT DID WE LEARN?

Huffman Coding is an optimal way to encode characters to minimize average number of bits needed to encode a character!

We greedily built subtrees & merging the 2 characters with the minimum total frequency (from the bottom up)

SUPER FUN FACT!!!

David Huffman came up with this as his term paper for an MIT class. His professor gave students the option to opt out of the final exam if they worked on a project to come up with optimal prefix code. Turns out that his professor, Robert Fano, had been working on coming up with a prefix code and had a more divide-and-conquer-y way to build a prefix tree, but it was suboptimal! Huffman didn't realize that the prefix code was an open problem (and later admitted that he wouldn't have tried it if he knew his professor had tried and couldn't get it), and he just managed to come up with this beautiful and optimal algorithm!

سوال؟