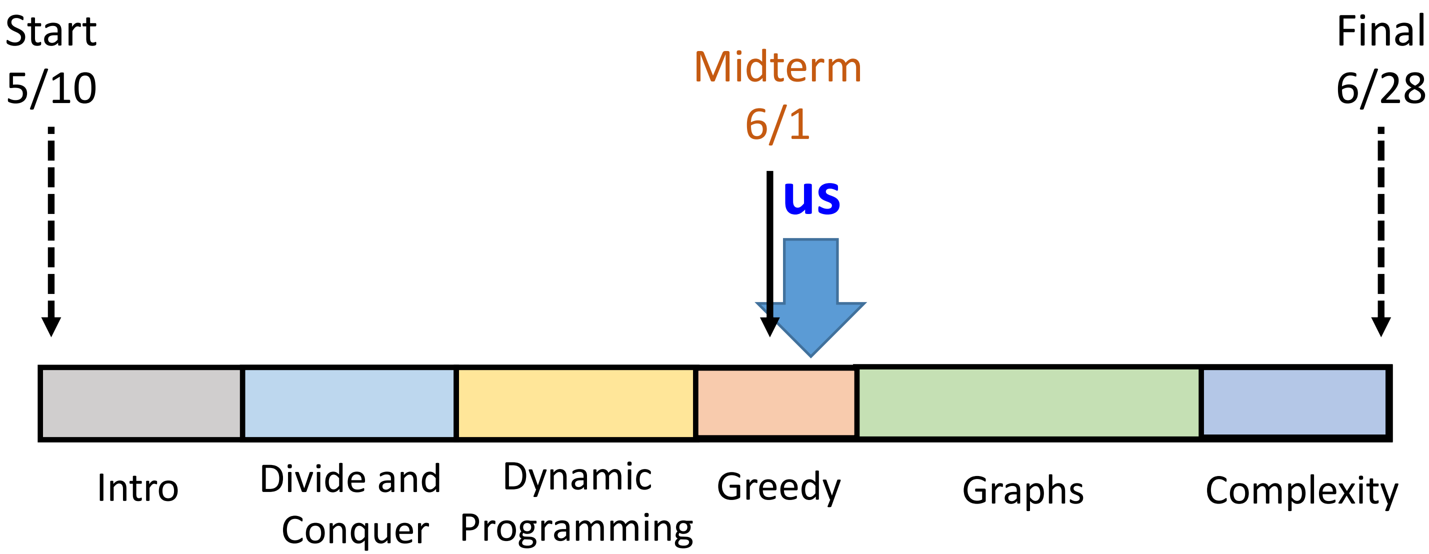
HW 3 due 6/6

# CS3000: Algorithms & Data
# Drew van der Poel

Lecture 13
- Finish Greedy: Exchange Argument
- Huffman Codes

June 2, 2021

# Outline




**Last class:** midterm review

**Next class:** graphs – terminology, BFS

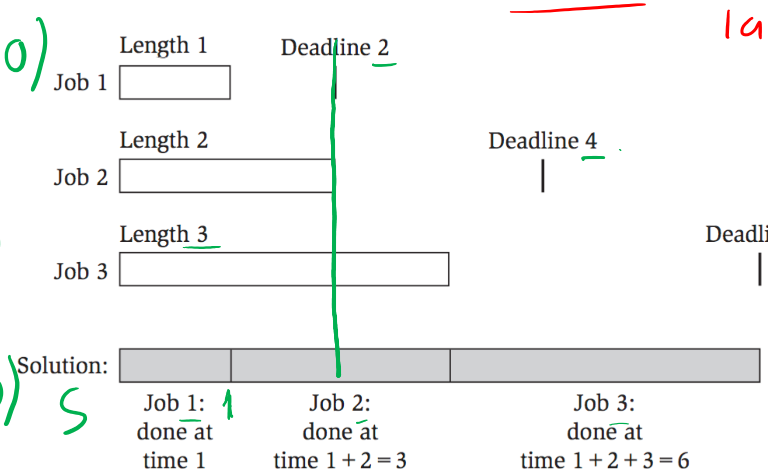
# Minimum Lateness Scheduling

- Input: $n$ jobs with length $t_\mathrm{i}$ and deadline $d_\mathrm{i}$
  - Simplifying assumption: all deadlines are distinct
- Output: a minimum--lateness schedule for the jobs
  - Can only do one job at a time, no overlap
  - The lateness of job $i$ is $\max\{f_\mathrm{i} - d_\mathrm{i}, 0\}$
  - The lateness of a schedule is $\max_{\mathrm{i}}\{\underbrace{\max\{f_\mathrm{i} - d_\mathrm{i}, 0\}}_{\text{lateness of } i}\}$

Lateness(1) = max(1-2, 0)
= 0

Lateness(2) = max(3-4, 0)
= 0

Lateness(3) = max(6-6, 0)
= 0

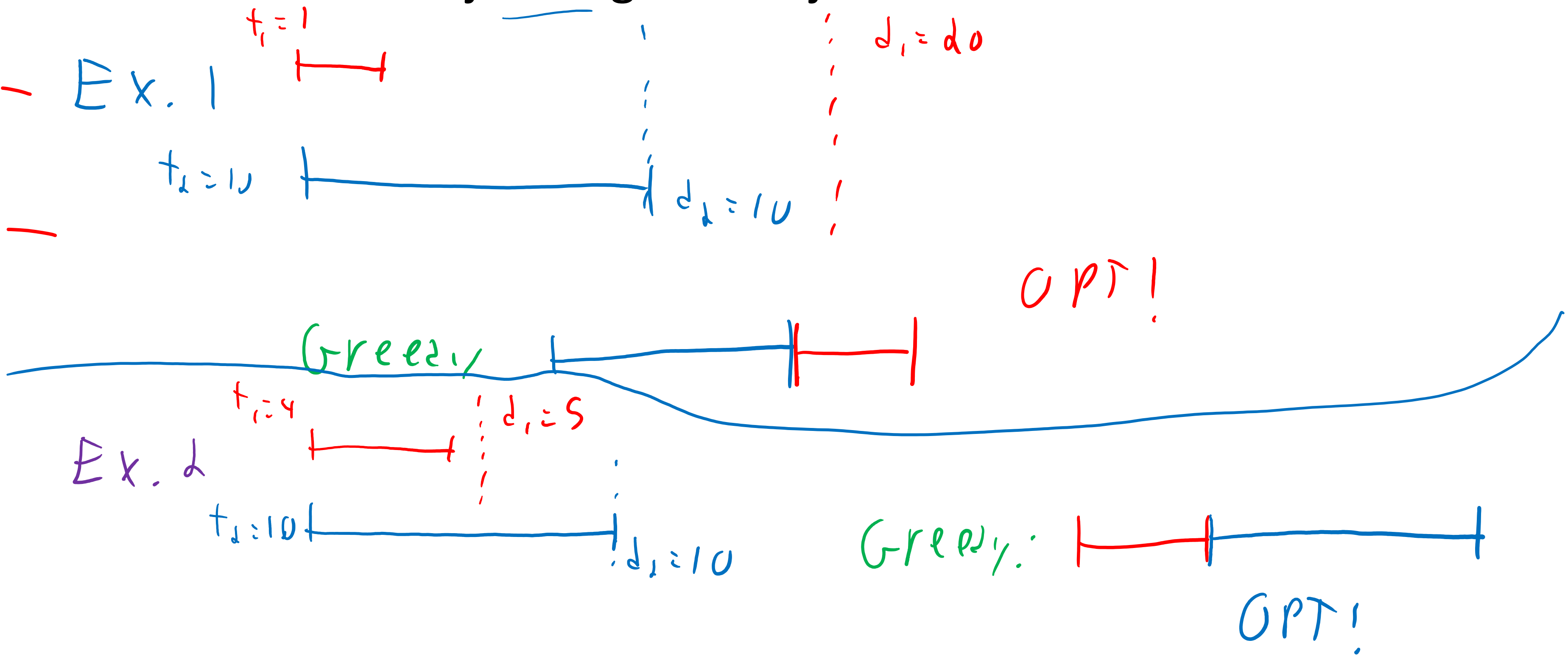



S

lateness(S) = 0

# Greedy Algorithm: Earliest Deadline First

- Sort jobs so that $d_1 \leq d_2 \leq \cdots \leq d_n$
- For $i = 1, \ldots, n$:
  - Schedule job $i$ right after job $i-1$ finishes



# Exchange Argument

- $G$ = greedy schedule, $O$ = (supposedly) optimal schedule

- Exchange Argument:
  - We can transform $O$ to $G$ by exchanging pairs of jobs
  - Each exchange only reduces the lateness of $O$
  - Therefore the lateness of $G$ is at most that of $O$

$O$: $j_1, j_2, \ldots, j_n$

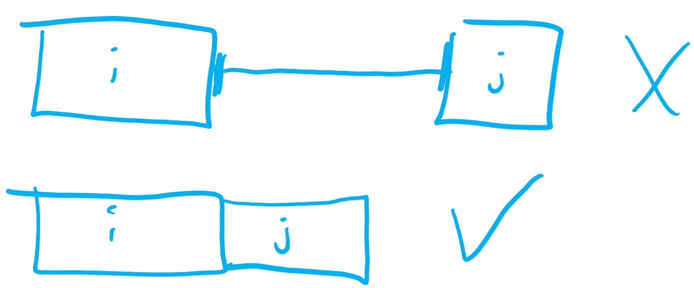
swap

$O \to$ swap 1 $\to$ swap 2 $\to \ldots \to G$

# Exchange Argument

- $G$ = greedy schedule, $O$ = (supposedly) optimal schedule

- Observation: ~~the~~ any optimal schedule has no gaps
  - A schedule is just an ordering of the jobs, with jobs scheduled back–to–back

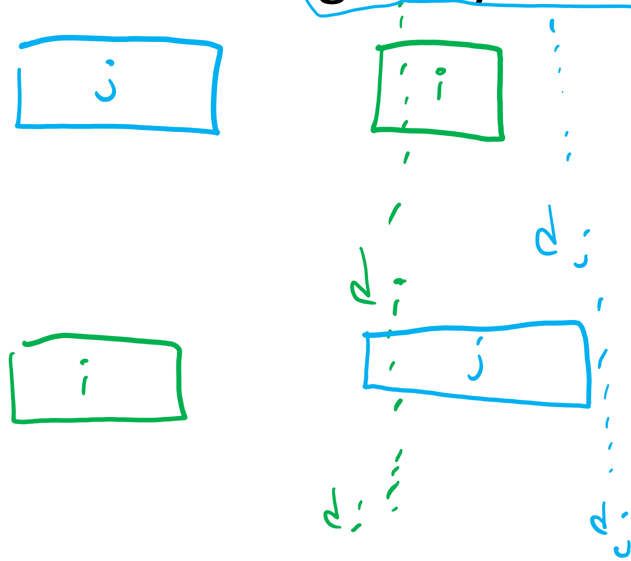


# Exchange Argument

any other schedule

- $G$ = greedy schedule, $O$ = (supposedly) optimal schedule

- We say that two jobs $i, j$ are inverted in $O$ if $d_i < d_j$ but $j$ comes before $i$ in the schedule
  - Observation: greedy has no inversions



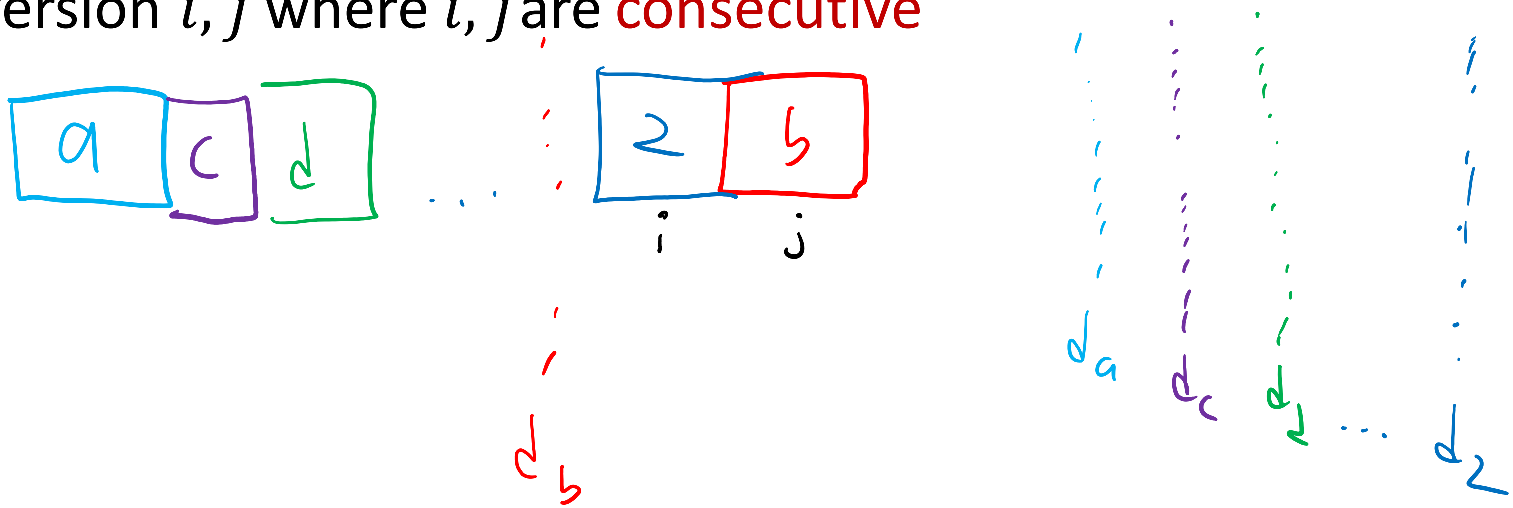

Inversion

No inversion

# Exchange Argument

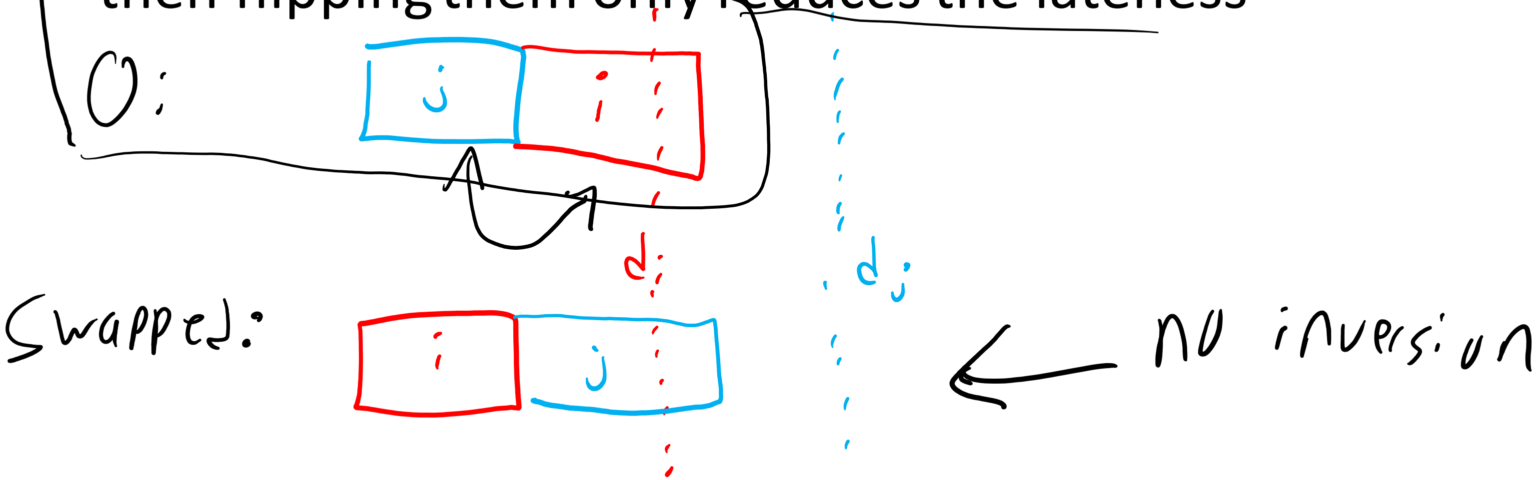
- We say that two jobs $i, j$ are inverted in $O$ if $d_i < d_j$ but $j$ comes before $i$
- Claim: an optimal schedule has no inversions
  - Step 1: suppose $O$ has an inversion, then it has an (a+b) inversion $i, j$ where $i, j$ are consecutive

a + b are inverted



# Exchange Argument

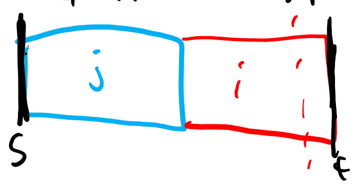
- We say that two jobs $i, j$ are inverted in $O$ if $d_{\mathrm{i}} < d_{\mathrm{j}}$ but $j$ comes before $i$
- Claim: an optimal schedule has no inversions
  - Step 1: suppose $O$ has an inversion, then it hasan inversion $i, j$ where $i, j$ are consecutive
  - Step 2: if $i, j$ are a consecutive jobs that are inverted then flipping them only reduces the lateness



# Exchange Argument

- If $i, j$ are a consecutive jobs that are inverted then flipping them only reduces the lateness

Inversion

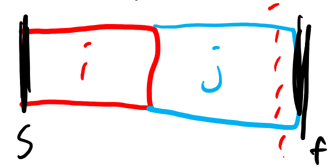



$d_i$

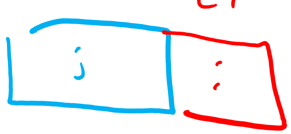
$\boxed{d_i < d_j}$ ✓

$d_j$

No Inversion




$d_i$ $d_j$




max. lateness(j)

0

$s + t_j - d_j$

lateness(i)

max lateness → 0

$\boxed{s + t_j + t_i - d_i}$

$s + t_j - d_j < s + t_j + t_i - d_i$

$d_i < d_j + t_i$

Case

$d_j > f = s + t_i + t_j$

$d_i < s + t_i + t_j$

max. lateness(j)

0

① $s + t_i + t_j - d_j$

max. lateness(i)

0

② $s + t_i - d_i$

① $s + t_j + t_i - d_i > s + t_i + t_j - d_j$

$d_j > d_i$ ✓

② $s + t_j + t_i - d_i > s + t_i - d_i$

$t_j > 0$ ✓

# Exchange Argument

- We say that two jobs $i, j$ are inverted in $O$ if $d_i < d_j$ but $j$ comes before $i$
- Claim: an optimal schedule has no inversions
  - Step 1: suppose $O$ has an inversion, then it has an inversion $i, j$ where $i, j$ are consecutive
  - Step 2: if $i, j$ are consecutive jobs that are inverted then flipping them only reduces the lateness

$O \longrightarrow$ swap 1 $\longrightarrow \ldots \longrightarrow G$

- $G$ is the unique schedule with no inversions, lateness(G) $\leq$ lateness(O)

- Problems: counting students, stable matching, sorting, n-digit multiplication, array searching, selection, weighted interval scheduling, segmented least squares, knapsack

- Alg. techniques: divide & conquer, dynamic programming, greedy

- Analysis: asymptotic analysis, recursion trees, Master Thm.

- Proof techniques: (strong) induction, contradiction, greedy stays ahead, **exchange argument**

# Data Compression

- How do we store strings of text compactly?

- A binary code is a mapping from alphabet $\Sigma \rightarrow \{0,1\}^*$
  - Simplest code: assign numbers $0, 1, \ldots, |\Sigma|-1$ to each symbol, map to binary numbers of $\lceil \log_2 |\Sigma| \rceil$ bits (**fixed-length**)

$\Sigma = \{a, b, c, d\}$

$|\Sigma| = 4$ 2-bits

$a \to 0$ 00

$b \to 1$ 01

$c \to 2$ 10

$d \to 3$ 11

  - Morse Code: (variable length)

| | | |
|---|---|---|
| A •– | J •––– | S ••• |
| B –••• | K –•– | T – |
| C –•–• | L •–•• | U ••– |
| D –•• | M –– | V •••– |
| E • | N –• | W •–– |
| F ••–• | O ––– | X –••– |
| G ––• | P •––• | Y –•–– |
| H •••• | Q ––•– | Z ––•• |
| I •• | R •–• | |

# Data Compression

- Letters have uneven frequencies!
  - Want to use short encodings for frequent letters, long encodings for infrequent letters -> smaller files/more compression

| | a | b | c | d | exp. enc. len. |
|---|---|---|---|---|---|
| **Frequency** | 1/2 | 1/4 | 1/8 | 1/8 | |
| **Encoding 1** | 00 | 01 | 10 | 11 | **2.0** |
| **Encoding 2** | 0 | 10 | 110 | 111 | **1.75** |

$$\frac{1}{2}\cdot 2 + \frac{1}{4}\cdot 2 + \frac{1}{8}\cdot 2 + \frac{1}{8}\cdot 2 = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{4}$$

$$= 2$$

$$\frac{1}{2}\cdot 1 + \frac{1}{4}\cdot 2 + \frac{1}{8}\cdot 3 + \frac{1}{8}\cdot 3 = 1 + \frac{6}{8} = 1.75$$

# Data Compression

- What properties would a good code of A-Z have?
  - The encoding is short on average

    $\leq$ 4 bits per letter
  - Easy to encode a string

    Encode(ALGO) = • − • − • • − − • − − −
  - Easy to decode a string? NO

    N G I

    Decode(− • − − • • •) =

    Y S

    K B

| | | |
|---|---|---|
| A •− | J •−−− | S ••• |
| B −••• | K −•− | T − |
| C −•−• | L •−•• | U ••− |
| D −•• | M −− | V •••− |
| E • | N −• | W •−− |
| F ••−• | O −−− | X −••− |
| G −−• | P •−−• | Y −•−− |
| H •••• | Q −−•− | Z −−•• |
| I •• | R •−• | |

# Prefix Free Codes

- Cannot decode if there are ambiguities
  - e.g. $\text{enc}(\text{“}E\text{”})$ is a prefix of $\text{enc}(\text{“}S\text{”})$ in Morse code

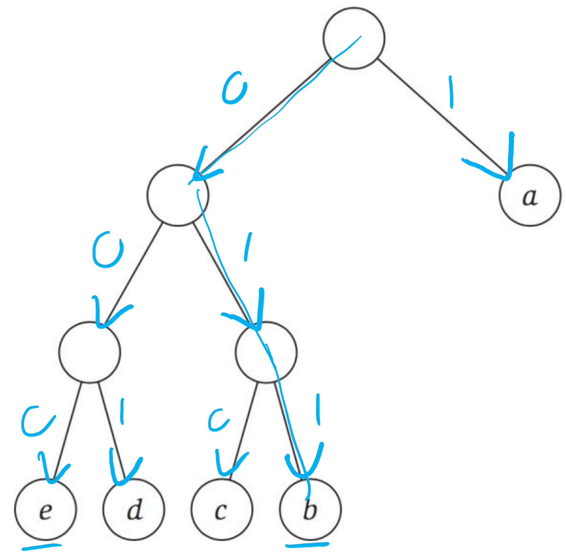
•                    • • •

E

- Prefix-Free Code:
  - A binary enc: $\Sigma \to \{0,1\}^*$ such that for every $x \neq y \in \Sigma$, $\text{enc}(x)$ is not a prefix of $\text{enc}(y)$
  - Any fixed-length code is prefix-free
  - Are all prefix-free codes fixed-length? NO

# Prefix Free Codes

- Can represent a prefix-free code as a binary tree
  - Each leaf is a character from the alphabet




Code:
a→1
b→011
c→010
d→001
e→000

- Encode by going up the tree (or using a table)
  - b e a d → 011 000 1 001
- Decode by going down the tree
  - 0 1 1 0 0 0 1 0 0 1 0 1 0 1 0 1 1

b e a d c a b

# Huffman Codes

- (An algorithm to find) an **optimal** prefix-free code

$\downarrow$ len(T)

| | a | b | c | d | exp. enc. len. |
|---|---|---|---|---|---|
| **Frequency** | 1/2 | 1/4 | 1/8 | 1/8 | |
| **Encoding 1** | 00 | 01 | 10 | 11 | **2.0** |
| **Encoding 2** | 0 | 10 | 110 | 111 | **1.75** |

$$\text{len}(E2) = f_a \cdot \text{len}_{E2}(a) + f_b \cdot \text{len}_{E2}(b) + f_c \cdot \text{len}_{E2}(c) + f_d \cdot \text{len}_{E2}(d)$$

$$= \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \frac{1}{8} \cdot 3$$

# of bits for i in T

- **optimal** = $\min_{\text{prefix-free } T} \text{len}(T) = \Sigma_{i \in \Sigma} (f_i * \text{len}_T(\text{i}))$
  - Note, optimality depends on what you're compressing
  - H is the 8th most frequent letter in English (6.094%) but the 20th most frequent in Italian (0.636%)

- Problems: counting students, stable matching, sorting, n-digit mulitiplication, array searching, selection, weighted interval scheduling, segmented least squares, knapsack, **prefix-free encoding**

- Alg. techniques: divide & conquer, dynamic programming, greedy

- Analysis: asymptotic analysis, recursion trees, Master Thm.

- Proof techniques: (strong) induction, contradiction, greedy stays ahead, exchange argument

# Huffman Codes

**Exp. Code Length:**

$$\Sigma_{i \in \Sigma} \left( f_i * \mathrm{len}_T(i) \right)$$

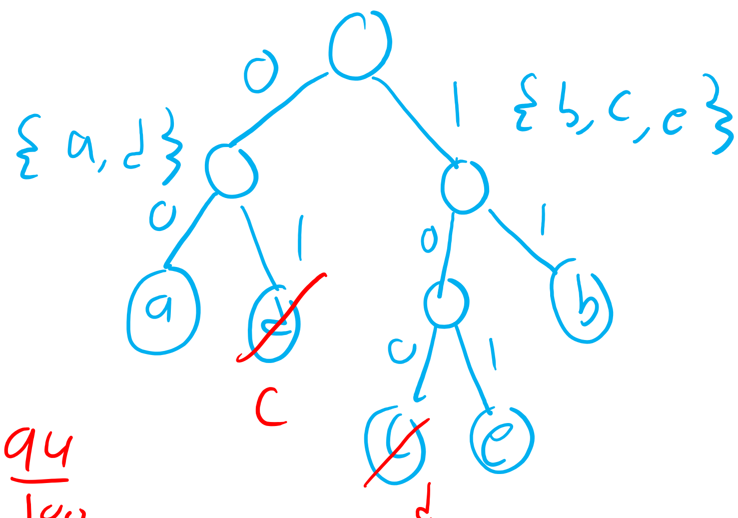
- Idea: Balanced binary trees should have low depth -> small lengths
- First Try: Split letters into two sets of roughly equal frequency and repeat (greedy!!)

| | a | b | c | d | e |
|---|---|---|---|---|---|
| $f(i)$ | .32 | .25 | .20 | .18 | .05 |

after swap:
$\mathrm{len}(c) = 2,\ f_c = 0.2$
$\mathrm{len}(d) = 3,\ f_d = 0.18$

$2 \cdot \frac{20}{100} + 3 \cdot \frac{18}{100}$

$= \frac{40 + 54}{100} = \frac{94}{100}$




Originally: $\mathrm{len}(d) = 2,\ f_d = .18$
$\mathrm{len}(c) = 3,\ f_c = .2$

$2 \cdot \frac{18}{100} + 3 \cdot \frac{20}{100} = \frac{36 + 60}{100}$

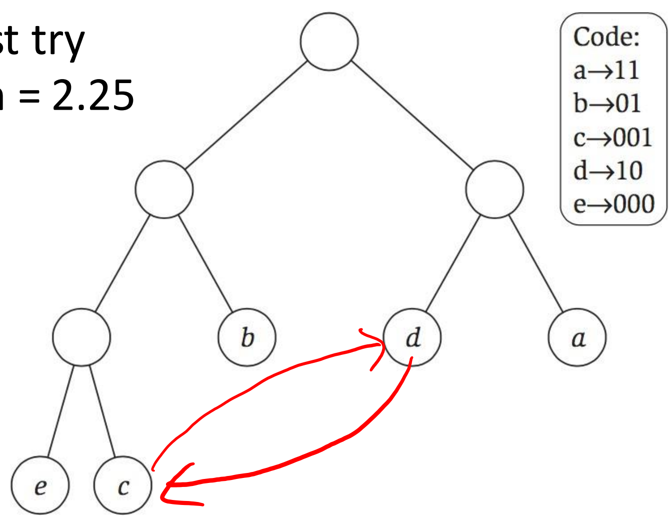
$= \frac{96}{100}$

# Huffman Codes

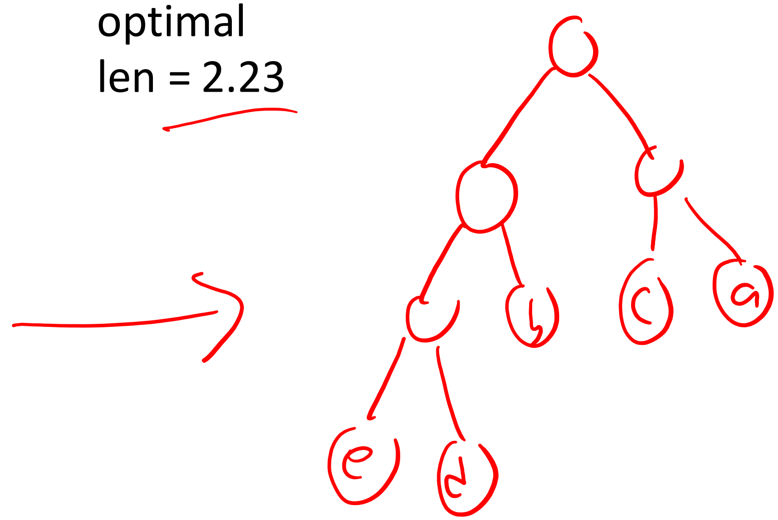
- First Try: Split letters into two sets of roughly equal frequency and repeat

| a | b | c | d | e |
|---|---|---|---|---|
| .32 | .25 | .20 | .18 | .05 |

first try
len = 2.25



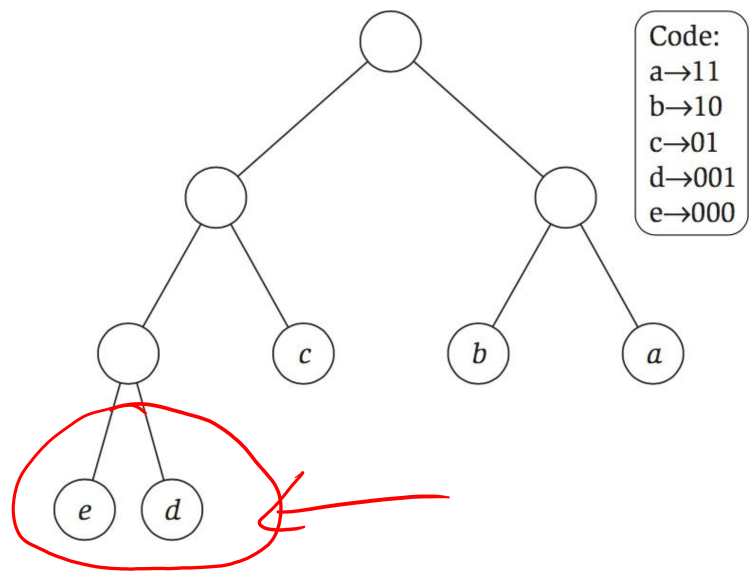

optimal
len = 2.23



# Huffman Codes

| a | b | c | d | e |
|---|---|---|---|---|
| .32 | .25 | .20 | .18 | .05 |

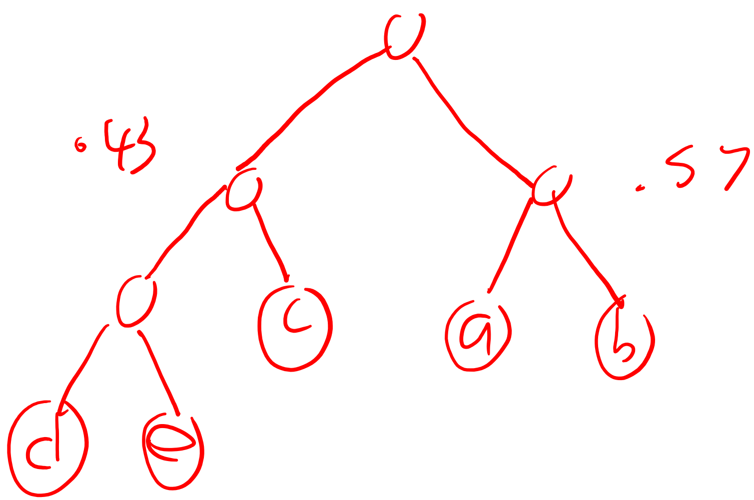
optimal
len = 2.23



# Huffman Codes

- Huffman's Algorithm: pair up the two letters with the lowest frequency and repeat

| a | b | c | d | e |
|---|---|---|---|---|
| .32 | .25 | .20 | .18 | .05 |



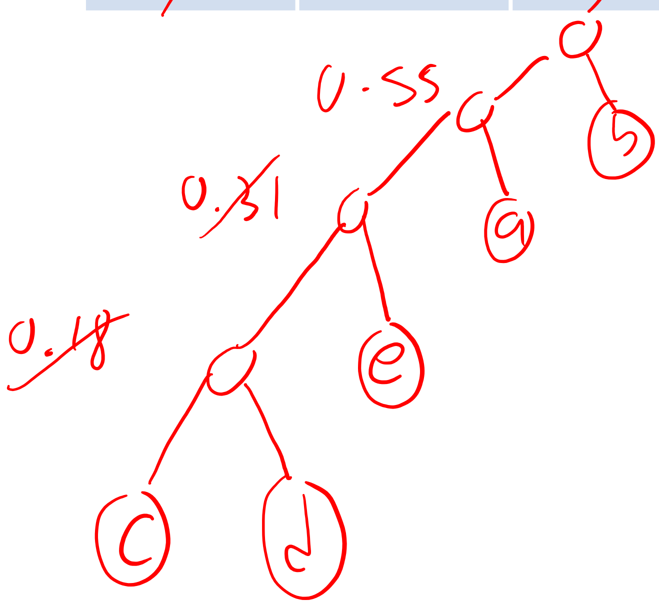

{e, d} .23

a → 10
b → 11
c → 01
d → 000
e → 001

# Now You Try!

- Huffman's Algorithm: pair up the two letters with the lowest frequency and repeat

| a | b | c | d | e |
|---|---|---|---|---|
| ~~.24~~ | .45 | ~~.11~~ | ~~.07~~ | ~~.13~~ |



# Huffman Codes

- Huffman's Algorithm: pair up the two letters with the lowest frequency and repeat
- Theorem: Huffman's Algorithm produces a prefix-free code of optimal length
  - We can prove the theorem using an exchange argument

# An Experiment

- Take the Dickens novel *A Tale of Two Cities*
    - File size is 799,940 bytes
- Build a Huffman code and compress - what letters have long codes?

| char | frequency | code |
|---|---|---|
| 'A' | 48165 | 1110 |
| 'B' | 8414 | 101000 |
| 'C' | 13896 | 00100 |
| 'D' | 28041 | 0011 |
| 'E' | 74809 | 011 |
| 'F' | 13559 | 111111 |
| 'G' | 12530 | 111110 |
| 'H' | 38961 | 1001 |

| char | frequency | code |
|---|---|---|
| 'I' | 41005 | 1011 |
| 'J' | 710 | 1111011010 |
| 'K' | 4782 | 11110111 |
| 'L' | 22030 | 10101 |
| 'M' | 15298 | 01000 |
| 'N' | 42380 | 1100 |
| 'O' | 46499 | 1101 |
| 'P' | 9957 | 101001 |
| 'Q' | 667 | 1111011001 |

| char | frequency | code |
|---|---|---|
| 'R' | 37187 | 0101 |
| 'S' | 37575 | 1000 |
| 'T' | 54024 | 000 |
| 'U' | 16726 | 01001 |
| 'V' | 5199 | 1111010 |
| 'W' | 14113 | 00101 |
| 'X' | 724 | 1111011011 |
| 'Y' | 12177 | 111100 |
| 'Z' | 215 | 1111011000 |

- File size is now 439,688 bytes

| | Raw | Huffman |
|---|---|---|
| **Size** | 799,940 | 439,688 |

# But Wait!

- Take the Dickens novel *A Tale of Two Cities*
  - File size is 799,940 bytes
- Build a Huffman code and compress

| char | frequency | code |
|---|---|---|
| ‘A’ | 48165 | 1110 |
| ‘B’ | 8414 | 101000 |
| ‘C’ | 13896 | 00100 |
| ‘D’ | 28041 | 0011 |
| ‘E’ | 74809 | 011 |
| ‘F’ | 13559 | 111111 |
| ‘G’ | 12530 | 111110 |
| ‘H’ | 38961 | 1001 |

| char | frequency | code |
|---|---|---|
| ‘I’ | 41005 | 1011 |
| ‘J’ | 710 | 1111011010 |
| ‘K’ | 4782 | 11110111 |
| ‘L’ | 22030 | 10101 |
| ‘M’ | 15298 | 01000 |
| ‘N’ | 42380 | 1100 |
| ‘O’ | 46499 | 1101 |
| ‘P’ | 9957 | 101001 |
| ‘Q’ | 667 | 1111011001 |

| char | frequency | code |
|---|---|---|
| ‘R’ | 37187 | 0101 |
| ‘S’ | 37575 | 1000 |
| ‘T’ | 54024 | 000 |
| ‘U’ | 16726 | 01001 |
| ‘V’ | 5199 | 1111010 |
| ‘W’ | 14113 | 00101 |
| ‘X’ | 724 | 1111011011 |
| ‘Y’ | 12177 | 111100 |
| ‘Z’ | 215 | 1111011000 |

  - File size is now 439,688 bytes
- But we can do better!

| | Raw | Huffman | gzip | bzip2 |
|---|---|---|---|---|
| Size | 799,940 | 439,688 | 301,295 | 220,156 |