# CS 161: Design and Analysis of Algorithms

# Announcements

- Homework 3, problem 3 removed

# Greedy Algorithms 4: Huffman Encoding/Set Cover

- Huffman Encoding
- Set Cover

# Alphabets and Strings

- **Alphabet** = finite set of symbols
  - English alphabet = {a,b,c,…,z}
  - Hex values = {0,1,…,9,A,B,C,D,E,F}
- **String** = sequence of symbols from some alphabet
  - "This is a string"

# How to Encode

- Computers store things as 0s and 1s
- How do we encode strings as sequence of bits?
  - Must be invertible (one-to-one)
  - What to use as few bits as possible
  - One approach: choose encoding for characters, induce encoding of strings by concatenating codes for each character

# How to Encode

- Obvious solution: If alphabet size is ≤ $2^k$ for some k, encode each character using k bits
  - Each character takes k bits
  - n characters
  - kn bits total

| Letter | Encoding |
|---|---|
| A | 00 |
| B | 01 |
| C | 10 |
| D | 11 |

$$\text{ABACBDAAADBAC} \downarrow 00010010011100000011010010$$

# How to Encode

- Issues:
  - Wasteful: If not exactly $2^k$ characters, some sequences never used

| Letter | Encoding |
|---|---|
| A | 00 |
| B | 01 |
| C | 10 |

Never use 11

# How to Encode

- Issues:
  - What if one character occurs very often?

AAAAAAABAAACAABAADAAAAAAAACAAAB

If almost all letters are A's, then an encoding that uses fewer bits to represent A and more to represent everything else would save on space

# Variable Length Encoding

- **Variable Length Encoding** = encoding of characters as bits where different letters may use a different number of bits
  - Still need encoding on strings to be one-to-one. What does this say about the encoding for characters?

# Variable Length Encoding

| Letter | Encoding |
|---|---|
| A | 0 |
| B | 01 |
| C | 10 |
| D | 11 |

AC $\longrightarrow$ 010

BA $\longrightarrow$ 010

Not one-to-one!

# Prefix-Free Encoding

- A **prefix** of a bit sequence is the first i bits, for some i

  <u>01001011010001101001</u>
  0
  01
  010
  0100
  01001
  …

# Prefix-Free Encoding

- A **prefix-free** encoding is an encoding of an alphabet such that no encoding of any character is a prefix of the encoding of any other character

| Letter | Encoding |
|---|---|
| A | 0 |
| B | 01 |
| C | 10 |
| D | 11 |

The encoding of A is a prefix of the encoding of C

# Prefix-Free Encoding

- A **prefix-free** encoding is an encoding of an alphabet such that no encoding of any character is a prefix of the encoding of any other character

| Letter | Encoding |
|---|---|
| A | 0 |
| B | 10 |
| C | 110 |
| D | 111 |

# Prefix-Free Encoding

- Theorem: Any prefix-free encoding of an alphabet induces a one-to-one encoding of strings over that alphabet

# Prefix-Free Encoding

- Proof: Suppose toward contradiction that S and T are two different strings that map to the same sequence of bits
  - Assume w.l.o.g. that S and T differ on the first character. Let c be the first character of S, d the first character of T.
  - Let $E(c)$ and $E(d)$ be the encodings of c and d
  - Assume w.l.o.g. $|E(c)| \geq |E(d)|$

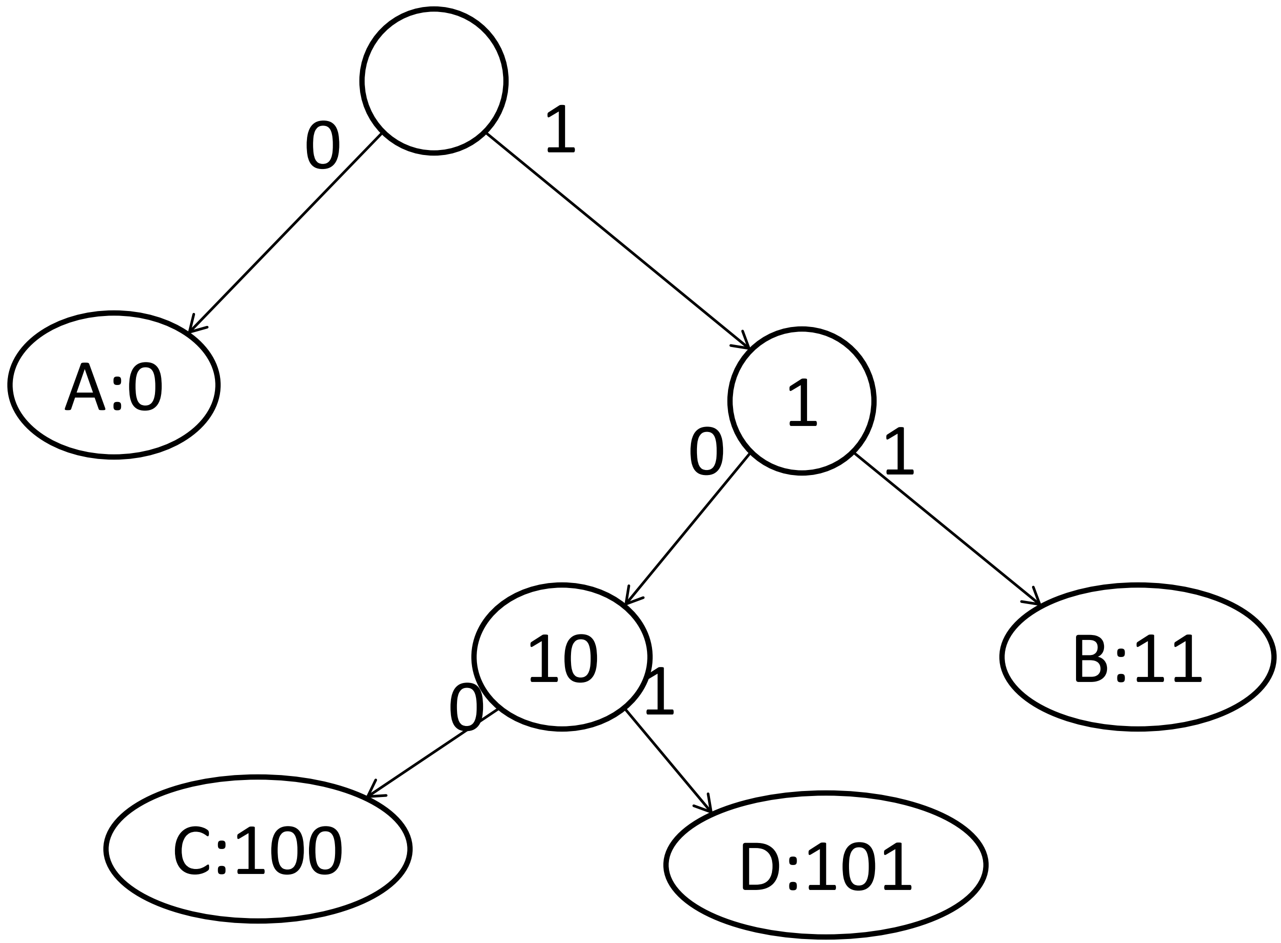
# Prefix-Free Encoding

- Since all bits in encodings of S and T are the same, the first |E(d)| bits are
- Therefore, the first |E(d)| bits of |E(c)| are equal to E(d)
- E(d) is a prefix of E(c)
- Since c was assumed different from d, our encoding is not prefix-free.

# Tree View of Prefix-Free Encoding

- Every node represents a partial codeword
- Every node has two children, one for appending 0 to the partial codeword, one for appending 1.
- Leaves correspond to actual codewords
- Root is empty

# Tree View of Prefix-Free Encoding



# Tree View of Prefix-Free Encoding

- To encode: Find path from root to character, concatenate edge labels
- To decode $b_1b_2$… : Starting from the root, follow edge labeled $b_1$, then edge labeled $b_2$, … until we find a leaf.  Output that character, and start over from the root

# Optimal Encoding

- What is the best possible prefix-free encoding we can find?
- Let n be the length of the string
- Let C be the cost of the encoding, defined as (length of encoding)/n
  - C = average length of encoding of characters, weighted by frequency

# Optimal Encoding

- Let $l_i$ be the length of the encoding of character i
- Let $f_i$ be the frequency i occurs in the string
  - $f_i$ (number of instances of i)/n

$$C = \sum_i f_i l_i$$

# Optimal Encoding

- $l_i$ is also the depth of character i in the encoding tree.
- Optimal encoding is always a full binary tree
  - If there is a node with only 1 child, replace node with child.
  - Depth of leafs only decreases.

# Optimal Encoding

- Entropy:

$$H = -\sum_i f_i \log f_i$$

- Theorem (Shannon Coding Theorem):

$$C \geq H$$

# Proof Of Coding Theorem

- Let $g(x) = x \log x$
- Lemma: $g( (x+y)/2 ) \leq ( g(x)+g(y) )/2$

# Proof Of Coding Theorem

- True when only 2 characters
  - Only possible encoding is for each character to get 1 bit. C = 1

$$H = -f_1 \log f_1 - f_2 \log f_2 = -2\left(\frac{g(f_1) - g(f_2)}{2}\right) \leq -2\left(g\left(\frac{f_1 + f_2}{2}\right)\right) = -2g(1/2) = 1$$

# Proof of Coding Theorem

- Inductively assume true for m-1 characters
- Let T be the tree corresponding to an optimal encoding over some alphabet of m characters
- At least two leafs at bottom level.  Assume w.l.o.g. these correspond to characters 1 and 2
- Replace all instances of characters 1 and 2 with a new character
  - Has frequency $f_1 + f_2$

# Proof of Coding Theorem

- Now we have an alphabet of size m-1
- Encoding for alphabet:
  - start with T
  - delete the nodes corresponding to characters 1 and 2
  - Assign the new character to the parent of these nodes (which is now a leaf)
  - New character has code length 1 less than deleted characters

# Proof of Coding Theorem

- How does C change?
  - Removed character 1 with length l, frequency $f_1$
  - Removed character 2 with length l, frequency $f_2$
  - Added new character, length l-1, frequency $f_1 + f_2$

$$C = \sum_i f_i l_i$$

$$C' = C - (f_1 + f_2)l + (f_1 + f_2)(l - 1) = C - (f_1 + f_2)$$

# Proof of Coding Theorem

- By inductive assumption,

$$C' \geq H' = -\sum f_i' \log f_i' = -\sum_{i \geq 3} f_i \log f_i - (f_1 + f_2) \log(f_1 + f_2)$$

$$= -\sum_i f_i \log f_i + f_1 \log f_1 + f_2 \log f_2 - (f_1 + f_2) \log(f_1 + f_2)$$

$$= H + f_1 \log f_1 + f_2 \log f_2 - (f_1 + f_2) \log(f_1 + f_2)$$

- Recall

$$C = C' + f_1 + f_2$$

# Proof of Coding Theorem

$$C \geq H + f_1 \log f_1 + f_2 \log f_2 - (f_1 + f_2)\left(\log(f_1 + f_2) - 1\right)$$

$$= H + f_1 \log f_1 + f_2 \log f_2 - (f_1 + f_2)\log\left(\frac{f_1 + f_2}{2}\right)$$

$$= H + 2\left(\frac{1}{2} g(f_1) + \frac{1}{2} g(f_2) - g\left(\frac{f_1 + f_2}{2}\right)\right)$$

$$\geq H$$

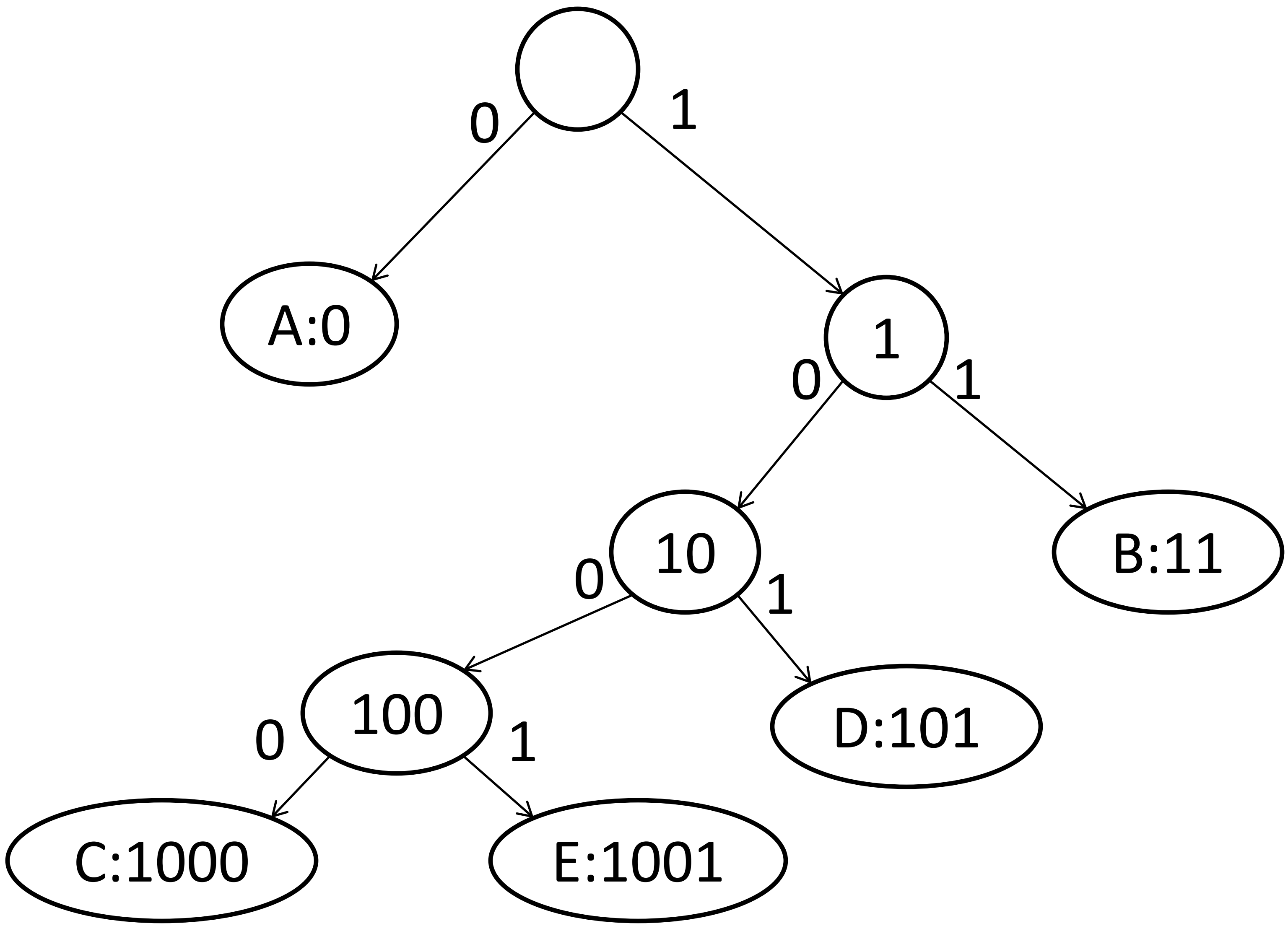
# How to Find Optimal Encoding

- Claim 1: There is an optimal solution where the two least frequent characters have the longest codewords (i.e. lowest level of tree), and are identical except for last bit
  - If not, swap these two characters with two of the characters with the longest codewords
  - Can swap with two that are siblings

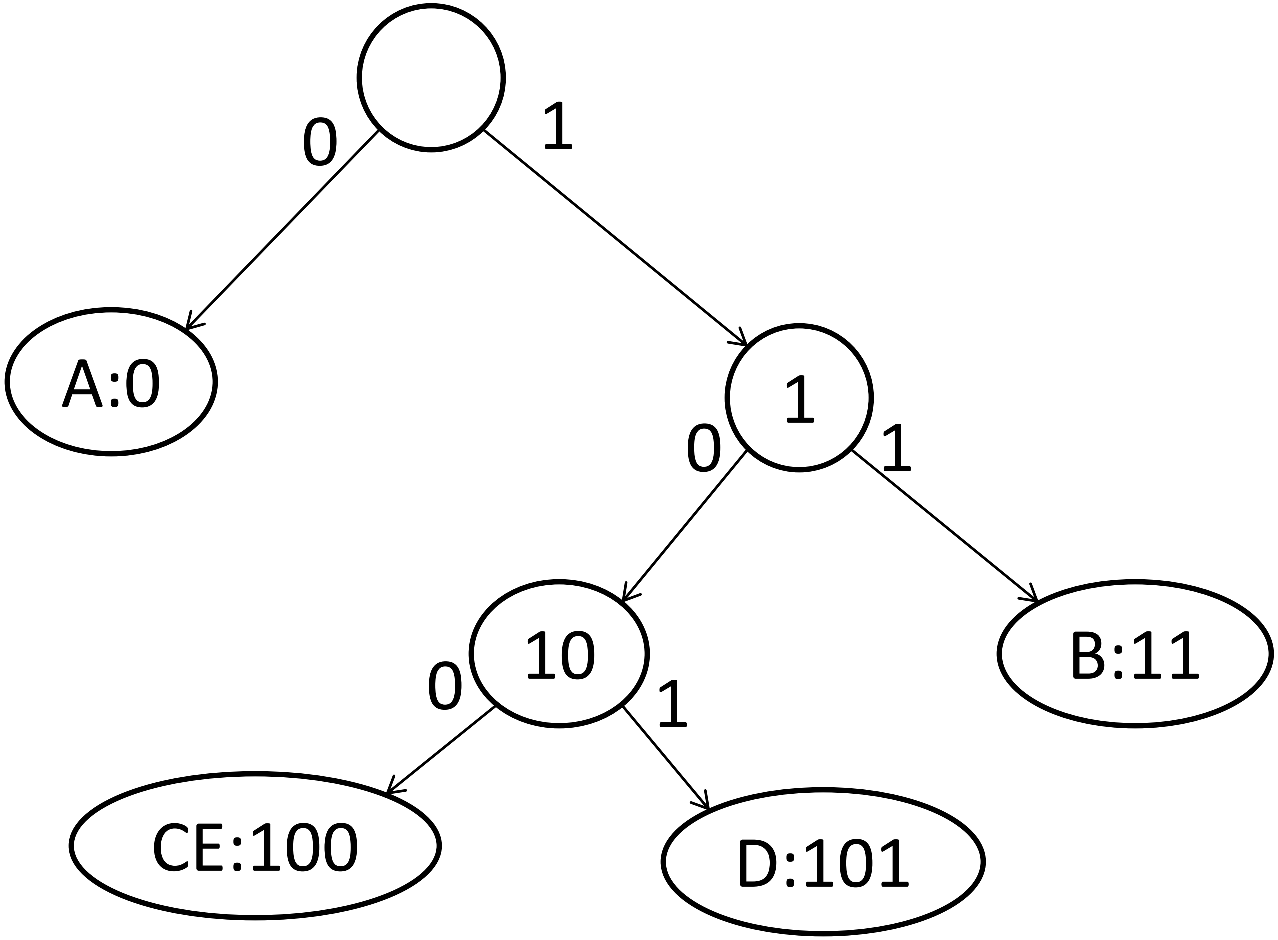
# How to Find Optimal Encoding

- Assume the two lowest-frequency characters are 1 and 2.
- What if we merge the two characters into a new character with frequency $f_1 + f_2$?
  - New character gets codeword obtained by dropping last bit of the codewords for 1 or 2

# Merging Two Characters



# Merging Two Characters



# How to Find Optimal Encoding

- Claim 2: For any optimal encoding, the encoding obtained by merging characters 1 and 2 must be an optimal encoding for the reduced alphabet, where characters 1 and 2 are replaced with a new character of frequency $f_1 + f_2$

# How to Find Optimal Encoding

| Character | Frequency | Codeword |
|---|---|---|
| A | $f_1$ | 0 |
| B | $f_2$ | 11 |
| C | $f_3$ | 1000 |
| D | $f_4$ | 101 |
| E | $f_5$ | 1001 |

| Character | Frequency | Codeword |
|---|---|---|
| A | $f_1$ | 0 |
| B | $f_2$ | 11 |
| CE | $f_3 + f_5$ | 100 |
| D | $f_4$ | 101 |

# How to Find Optimal Encoding

- Idea:
  - Take two characters with lowest frequency
  - Merge them
  - Recursively solve reduced problem
  - Split characters apart again

# How to Find Optimal Encoding

| Character | Frequency | Codeword |
|---|---|---|
| A | 0.45 | |
| B | 0.25 | |
| C | 0.10 | |
| D | 0.15 | |
| E | 0.05 | |

# How to Find Optimal Encoding

| Character | Frequency | Codeword |
|---|---|---|
| A | 0.45 | |
| B | 0.25 | |
| C | 0.10 | |
| D | 0.15 | |
| E | 0.05 | |

# How to Find Optimal Encoding

| Character | Frequency | Codeword |
|---|---|---|
| A | 0.45 | |
| B | 0.25 | |
| [CE] | 0.15 | |
| D | 0.15 | |

# How to Find Optimal Encoding

| Character | Frequency | Codeword |
|---|---|---|
| A | 0.45 | |
| B | 0.25 | |
| [CE] | 0.15 | |
| D | 0.15 | |

# How to Find Optimal Encoding

| Character | Frequency | Codeword |
|---|---|---|
| A | 0.45 | |
| B | 0.25 | |
| [[CE]D] | 0.30 | |

# How to Find Optimal Encoding

| Character | Frequency | Codeword |
|---|---|---|
| A | 0.45 | |
| B | 0.25 | |
| [[CE]D] | 0.30 | |

# How to Find Optimal Encoding

| Character | Frequency | Codeword |
| --- | --- | --- |
| A | 0.45 | |
| [[[CE]D]B] | 0.55 | |

# How to Find Optimal Encoding

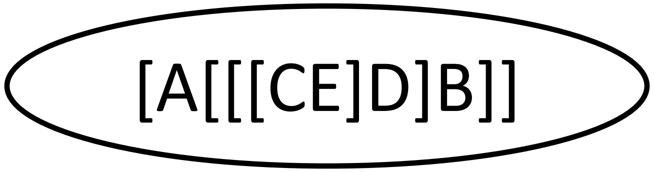
| Character | Frequency | Codeword |
| --- | --- | --- |
| A | 0.45 | |
| [[[CE]D]B] | 0.55 | |

# How to Find Optimal Encoding

| Character | Frequency | Codeword |
| --- | --- | --- |
| [A[[[CE]D]B]] | 1.00 | |

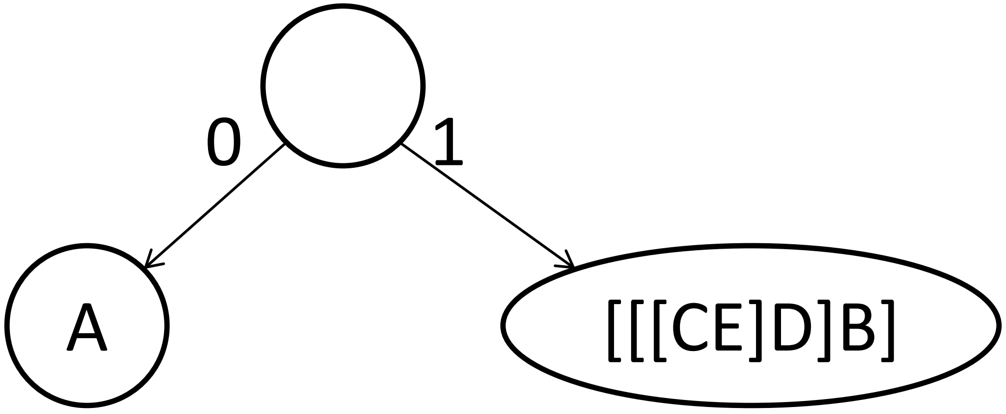
# How to Find Optimal Encoding

| Character | Frequency | Codeword |
|---|---|---|
| [A[[[CE]D]B]] | 1.00 | |



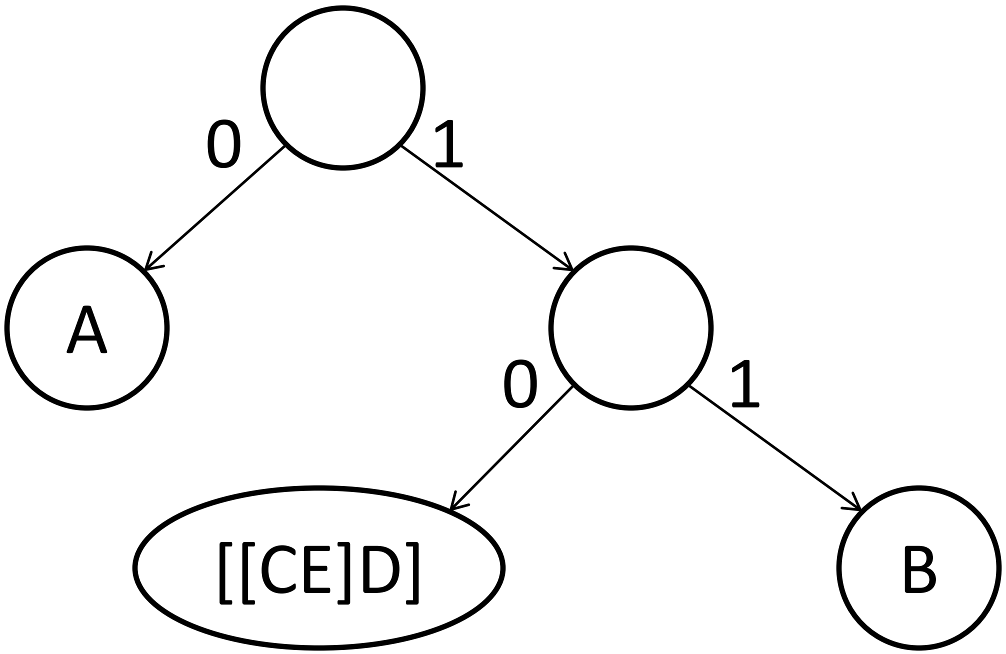
# How to Find Optimal Encoding

| Character | Frequency | Codeword |
| --- | --- | --- |
| A | 0.45 | 0 |
| [[[CE]D]B] | 0.55 | 1 |



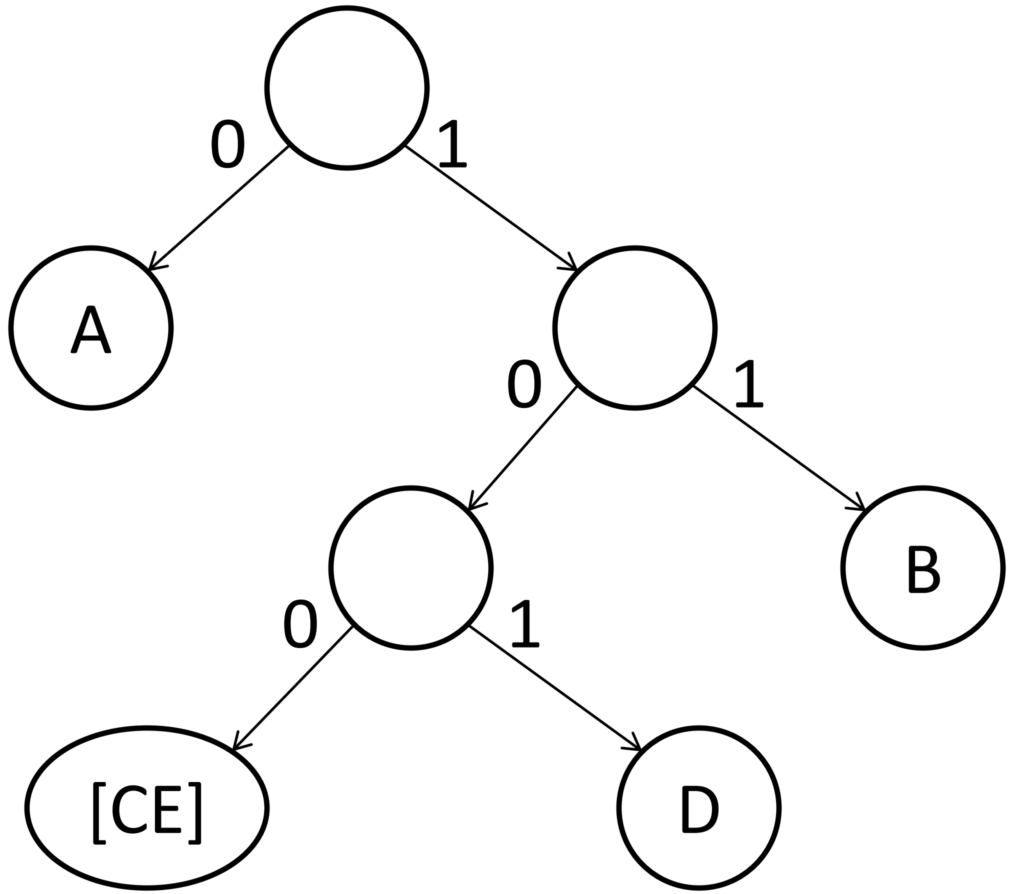
# How to Find Optimal Encoding

| Character | Frequency | Codeword |
|---|---|---|
| A | 0.45 | 0 |
| B | 0.25 | 11 |
| [[CE]D] | 0.30 | 10 |



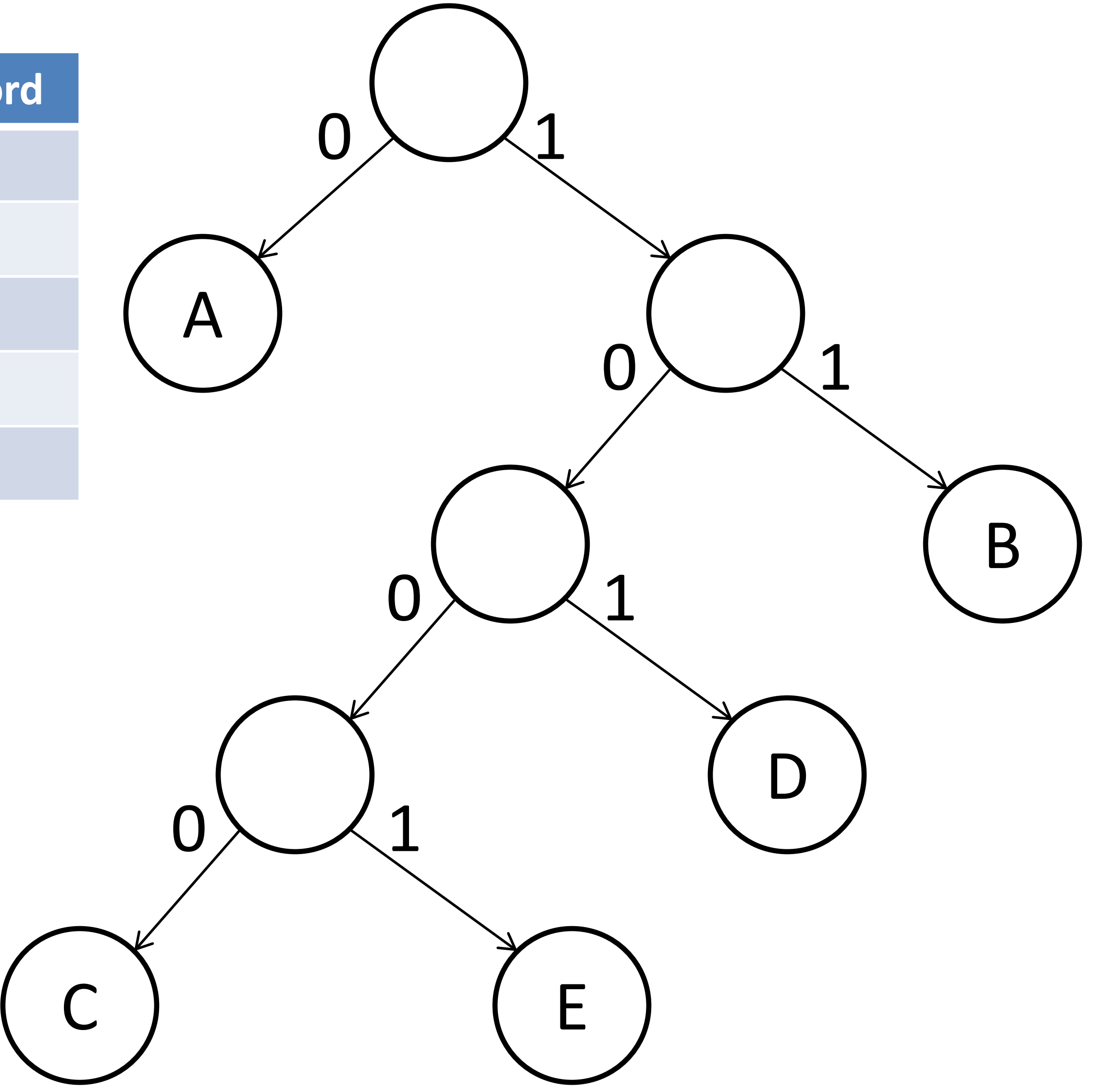
# How to Find Optimal Encoding

| Character | Frequency | Codeword |
|---|---|---|
| A | 0.45 | 0 |
| B | 0.25 | 11 |
| [CE] | 0.15 | 100 |
| D | 0.15 | 101 |



# How to Find Optimal Encoding

| Character | Frequency | Codeword |
|---|---|---|
| A | 0.45 | 0 |
| B | 0.25 | 11 |
| C | 0.10 | 1000 |
| D | 0.15 | 101 |
| E | 0.05 | 1001 |



# How to Find Optimal Encoding

- Let q be a heap of characters, ordered by frequency
- For each character c, q.insert(c)
- While q has at least two characters:
  - $c_1$ = q.deletemin(), $c_2$ = q.deletemin()
  - Create a node labeled $[c_1 c_2]$ with children $c_1$ and $c_2$
  - $f([c_1 c_2]) = f(c_1) + f(c_2)$
  - q.insert ($[c_1 c_2]$ )
- Return q.deletemin()

# Running Time

- n inserts initially: O(n log n)
- Every run of loop decreases size of heap by 1
  - n-1 runs of loop
- Each run of loop involves 3 heap operations: O(log n)
- Total running time: O(n log n)

# Set Cover

- Given a set of elements B, and a collection of subsets $S_i$, output a selection of the $S_i$ whose union is B, such that the number of subsets used is minimal.

# Example: Schools

- Suppose we have a collection of towns, and we want to figure out the best towns to put schools
  - Need at least one school within 20 miles of each town
  - Every school should be in a town

# Example: Schools

- B = set of towns
- $S_i$ = subset of towns within 20 miles of town i

# Greedy Solution

- Obvious solution: repeatedly pick the set $S_i$ with the largest number of uncovered elements.

# Example

- $B = \{1, 2, 3, 4, 5, 6\}$
- $S_1 = \{1, 2, 3\}$
- $S_2 = \{1, 4\}$
- $S_3 = \{2, 5\}$
- $S_4 = \{3, 6\}$

# Example

- $B = \{1, 2, 3, 4, 5, 6\}$
- $S_1 = \{1, 2, 3\}$
- $S_2 = \{1, 4\}$
- $S_3 = \{2, 5\}$
- $S_4 = \{3, 6\}$

<u>Greedy Algorithm</u>

Sets used:

$\{\}$

Elements left:

$\{1, 2, 3, 4, 5, 6\}$

# Example

- $B = \{1, 2, 3, 4, 5, 6\}$
- $S_1 = \{1, 2, 3\}$
- $S_2 = \{1, 4\}$
- $S_3 = \{2, 5\}$
- $S_4 = \{3, 6\}$

<u>Greedy Algorithm</u>

| Sets used: | Elements left: |
|---|---|
| $\{S_1\}$ | $\{4, 5, 6\}$ |

# Example

- B = {1, 2, 3, 4, 5, 6}
- $S_1$ = {1, 2, 3}
- $S_2$ = {1, 4}
- $S_3$ = {2, 5}
- $S_4$ = {3, 6}

<u>Greedy Algorithm</u>

Sets used:
{$S_1$, $S_2$}

Elements left:
{5, 6}

# Example

- $B = \{1, 2, 3, 4, 5, 6\}$
- $S_1 = \{1, 2, 3\}$
- $S_2 = \{1, 4\}$
- $S_3 = \{2, 5\}$
- $S_4 = \{3, 6\}$

<u>Greedy Algorithm</u>

| Sets used: | Elements left: |
|---|---|
| $\{S_1, S_2, S_3\}$ | $\{6\}$ |

# Example

- $B = \{1, 2, 3, 4, 5, 6\}$
- $S_1 = \{1, 2, 3\}$
- $S_2 = \{1, 4\}$
- $S_3 = \{2, 5\}$
- $S_4 = \{3, 6\}$

<u>Greedy Algorithm</u>

| Sets used: | Elements left: |
| --- | --- |
| $\{S_1, S_2, S_3, S_4\}$ | $\{\}$ |

# Example

- B = {1, 2, 3, 4, 5, 6}
- $S_1$ = {1, 2, 3}
- $S_2$ = {1, 4}
- $S_3$ = {2, 5}
- $S_4$ = {3, 6}

Greedy Algorithm

Sets used:
{$S_1$ , $S_2$, $S_3$, $S_4$}

Elements left:
{}

Optimal:
{ $S_2$, $S_3$, $S_4$}

# Set Cover

- Greedy algorithm isn't optimal!
- Obtaining optimal solution believed hard
- Settle for approximation:
  - If optimal uses k sets, want to get solution using only slightly more than k sets

# Approximation

- Claim: If B contains n elements, and the optimal solution uses k sets, then greedy uses at most k ln n sets

# Proof

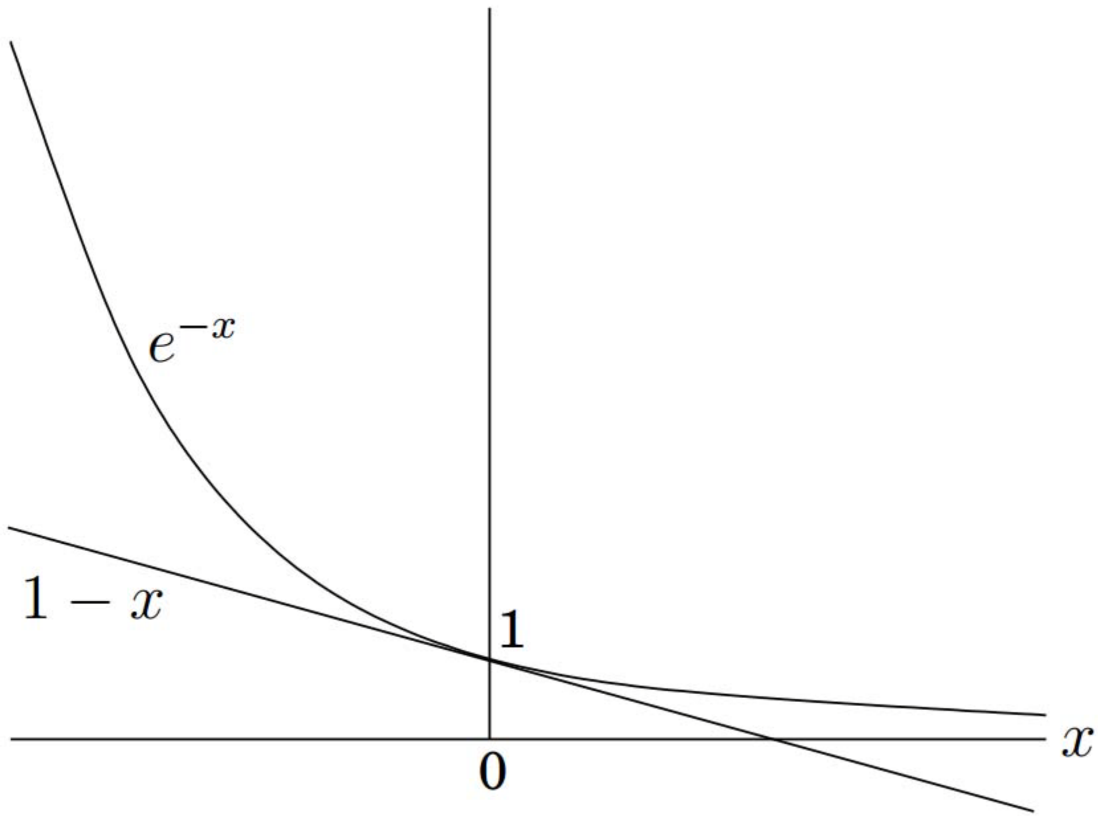
- Let $n_t$ be the number of uncovered elements after t iterations of greedy algorithm ($n_0 = n$)
- Remaining elements covered by the optimal k sets
- Must be some set with at least $n_t/k$ of the uncovered elements
- Therefore, greedy picks a set that covers at least $n_t/k$ of the remaining elements

# Proof

- Greedy picks a set that covers at least $n_t/k$ of the remaining elements
- $n_{t+1} \le n_t - n_t/k = n_t(1-1/k)$
- Therefore, $n_t \le n_0(1-1/k)^t = n(1-1/k)^t$

# Proof

- Fact: $1\text{-}x \leq e^{-x}$, with equality if and only if $x = 0$



# Proof

- $n_t \leq n(1\text{-}1/k)^t < n(e^{-1/k})^t < ne^{-t/k}$
- After t = k ln n iterations, $n_t < n\ e^{-\ln n} = 1$
- Therefore, after t = k ln n iterations, $n_t = 0$
- Therefore, greedy algorithm uses at most k ln n sets, as desired

# Can We Do Better

- Our algorithm achieves an approximation ratio of ln n
- This gives two questions:
  - Can the analysis be tightened so that greedy achieves a better approximation ratio?
  - Are there more sophisticated algorithms that achieve better approximation ratio?
- Answer to both: most likely not
  - If domr efficient algorithm can do much better, than we can solve a whole host of very difficult problems efficiently