CS340 - Analysis of Algorithms

Greedy Algorithms
Huffman Codes

Upcoming important dates:

Lab3 due tomorrow (9/25)

Project checkpoint 1 due sunday (10/5)

HW4 due next monday (10/6)

Midterm (10/8) - if you have testing accommodations, let me know immediately

Agenda

1. Warmup - Color Matching Problem

2. Huffman codes

Warmup: Color Matching Problem

A digital art gallery has a large collection of N digital images. Each image is composed of a fixed number of colors.

The gallery wants to organize the images into “color palettes”

A color palette is a set of images that can be arranged in a specific sequence :

1. The first image in the sequence can be any color
2. For any two consecutive images in the sequence, I_a and I_b, they must have at least one color in common
3. A color palette is considered “complete” if it includes at least one images of each of the C total possible colors

The gallery wants to find the smallest possible complete color palette. Your task is to find the minimum number of images required to form a valid complete color palette, or determine that it's impossible.

How would this problem be represented on a graph?

Steps to finding an algorithm

1. Construct a good example (not too small).
2. Solve the example and check your solution (by hand, without worrying about the algorithm)
3. Think about how you solved and/or checked the problem and how you can use that to solve a general instance.
4. Formalize an algorithm (might have to formalize the problem too)
5. Construct a new and somehow different example and run your algorithm on it and check the solution.
6. Repeat until you are confident it works.

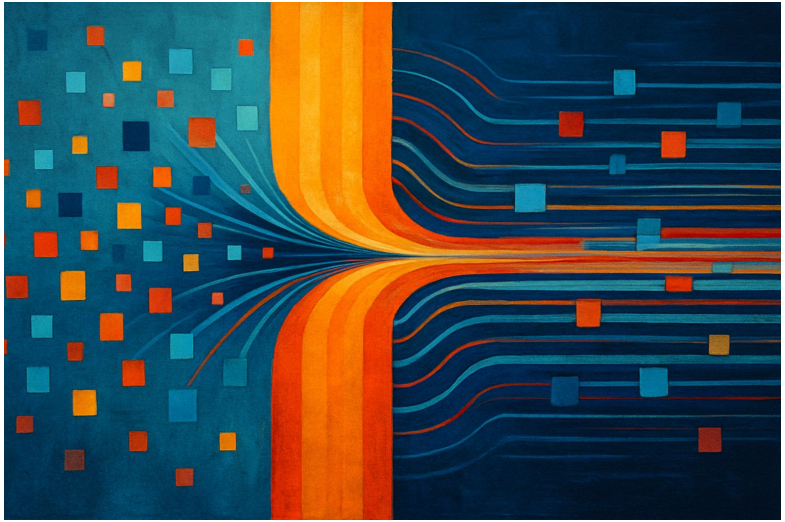
Warmup: Color Matching Problem

Vertices: Images

Edges: unweighted links exist between images with a shared color

How would you solve this?

Find any path through all nodes that does not contain a cycle



Huffman Codes

Encoding Characters Using Bits

- Computers operate on sequences of bits (consisting of only 0 and 1)

- How do we represent text as 0s and 1s?
    - Encoding schemes

- What do we want from an encoding scheme?
    - Deterministic round trip encoding / decoding
    - Data compression - reducing the average number of bits per letter

- Naive approach: fixed number of bits for each symbol in the alphabet and then just concat the bit strings for each symbol to make the text

Encoding Characters Using Bits (Naive approach)

Simple example: Suppose we want to encode the language {a, b, c, d} with only 0s and 1s.

How long will each characters encoding be?

Max: 2 a = 0, b = 1, c = 01, d = 11

Encoding Characters Using Bits (Naive approach)

Now suppose we want to encode English alphabet + 5 punctuation characters (comma, period, question mark, exclamation point, and apostrophe)

32 symbols that need to be encoded

How long will each characters encoding be?

Max: 5 ($2^5 = 32$)   a = 00000, b = 00001, ..., apostrophe = 11111

Is this optimal? Can we construct something smaller?

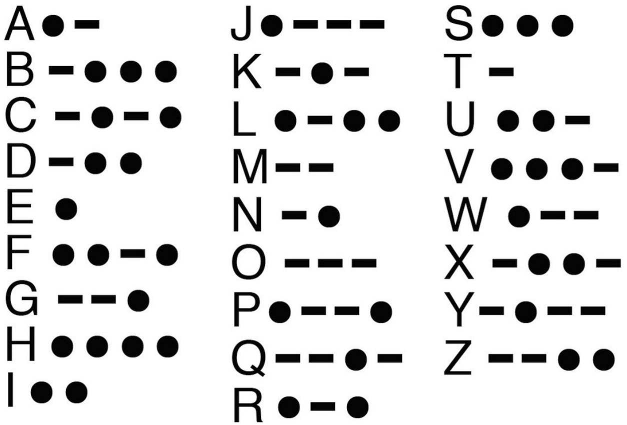
Some characters (z, q, x) are much less frequently occuring than others (e, t, i)

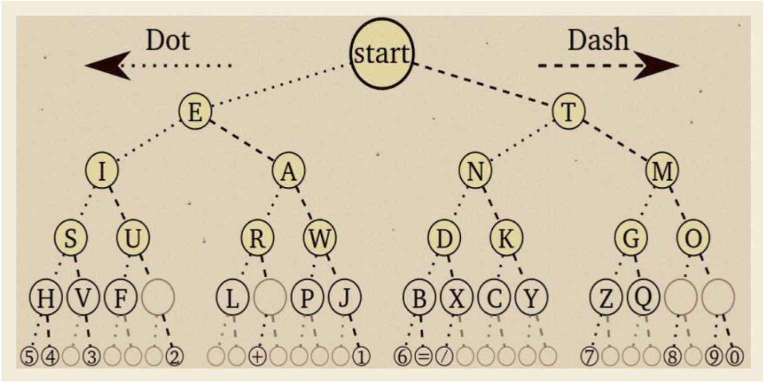
Variable Length Encoding Schemes

Morse Code: Each letter is represented as a sequence of *dots* (short pulses) and *dashes* (long pulses)

Morse consulted local printing presses to get frequency estimates for letters in English

More frequent English letters are shorter encodings





Variable Length Encoding Schemes

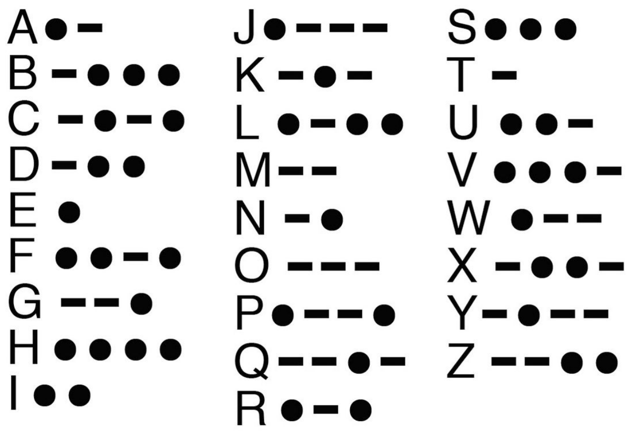
Morse Code: Let's consider dots as 0s and dashes as 1s

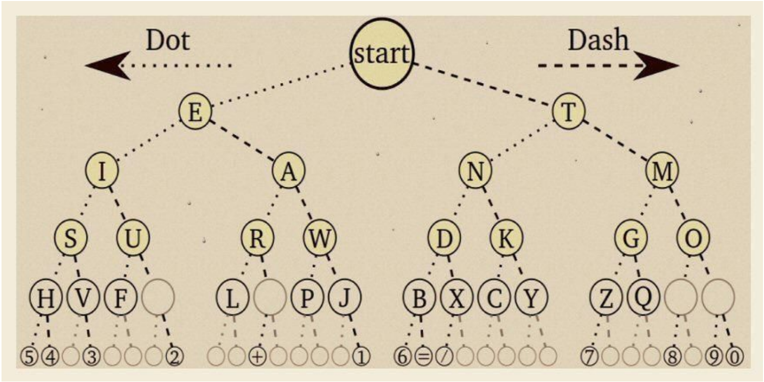
What does 0101 correspond to?

- eta
- aa
- etet
- aet ...

How do we know when a letter is done transmitting?

Morse solution: introduce a pause between letters





Prefix Codes

Ambiguity arises in Morse code because there exist pairs of letters where the bit string that encodes one letter is the *prefix* of the bit string that encodes another.

How can we eliminate this problem without introducing a delimiter? (pause)

Map letters to bit strings in such a way that no encoding is a prefix of any other

Prefix Codes

A prefix code for a set of letters is a function y that maps each letter $x \in S$ to some sequence of zeros and ones in such a way that for any distinct $x, z \in S$, the sequence $y(x)$ is not a prefix of the sequence $y(z)$

Example: Is the encoding y_1 for $S = \{a, b, c, d, e\}$ a prefix code?

y_1 (a) = 11 y_1 (b) = 01 y_1 (c) = 001 y_1 (d) = 10 y_1 (e) = 000

Yes. No encoding is a prefix of the other

Decode 0010000011101

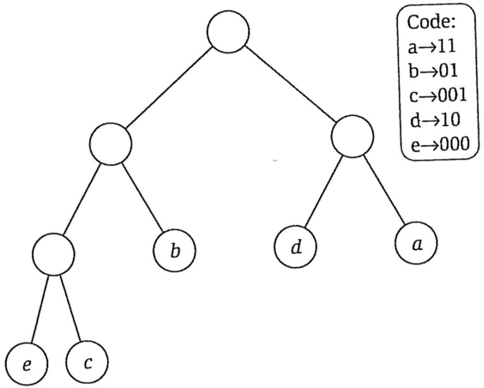
c e c a b

Prefix Codes as Binary Trees

We can represent binary (0/1) prefix coding using a binary tree

Consider a binary tree T with |S| leaves

- Labels can only be on leaf nodes
  - Otherwise, the encoding of one character would be a prefix of the encoding of another
- Decoding = traversal
  - For each letter x, we follow the path from the root to leaf x
  - Left -> 0
  - Right -> 1
- **Decode** 0010000011101
- Height of the tree is the max code length for a letter



Optimal Prefix Codes

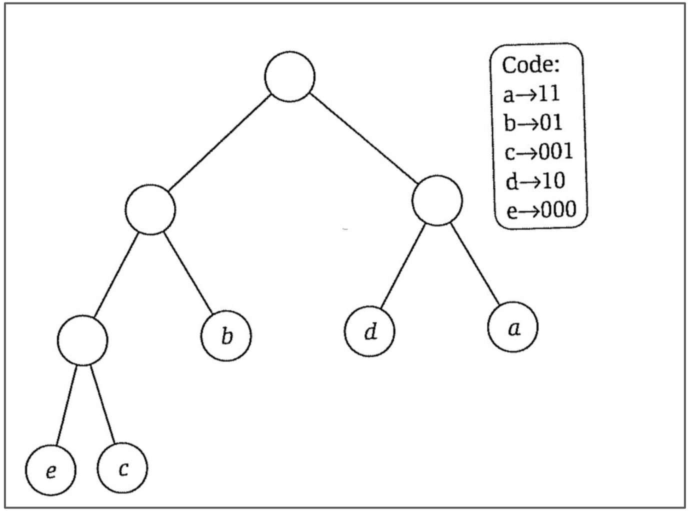
- Given an alphabet S and the probabilities $p(x)$ of occurrence for each character $x \epsilon S$, compute a prefix code T that minimizes the expected length of the encoded bit-string, $B(T)$

$$B(T) = \sum_{x \in S} p(x) d_T(x)$$

- Optimal code is not unique

B(T) for our example encoding y_1?

$$B(T) = \sum_{x \in S} p(x) d_T(x)$$



p(a) = .32

p(b) = .25

p(c) = .20

p(d) = .18

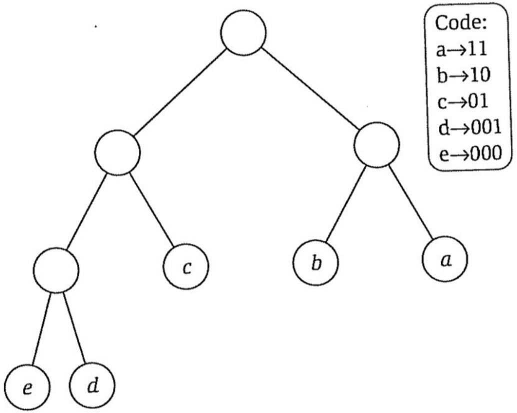
p(e) = .05

B(T) = .32(3) + .25(2) + .20(3) + .18(2) + .05(3)
= 2.25

Is this the best we can do?

An alternate encoding y_2

$$B(T) = \sum_{x \in S} p(x) d_T(x)$$



y_2

p(a) = .32
p(b) = .25
p(c) = .20
p(d) = .18
p(e) = .05

B(T) = .32(2) + .25(2) + .20(2) + .18(3) + .05(3)
= 2.23 ✓

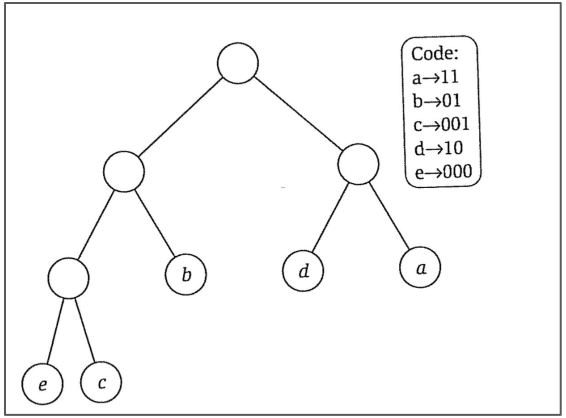
Shannon-Fano Codes

A top-down approach:

1. Split the alphabet into two sets S_1 and S_2 such that the probabilities are as balanced as possible (.5 and .5 ideally)
2. Recursively construct prefix codes for S_1 and S_2 separately
   a. Each are subtrees of the root
   b. One is prefixed with 0 the other is prefixed with 1

p(a) = .32
p(b) = .25
p(c) = .20
p(d) = .18
p(e) = .05

{b, c, e} and {a, d}



Results in our y_1 (non-optimal)

Huffman's Algorithm

- Greedy Algorithm to construct an optimal prefix code
- **Bottom up approach**

- *What if we knew the tree structure of the optimal prefix code?*
  - Suppose u and v are leaves and depth(u) < depth(v)
  - We should label u and v with x and y if p(x) ≥ p(y)

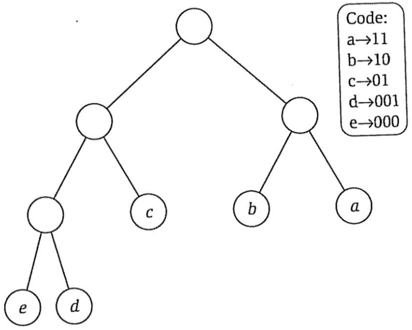
p(a) = .32

p(b) = .25

p(c) = .20

p(d) = .18

p(e) = .05



Huffman's Algorithm

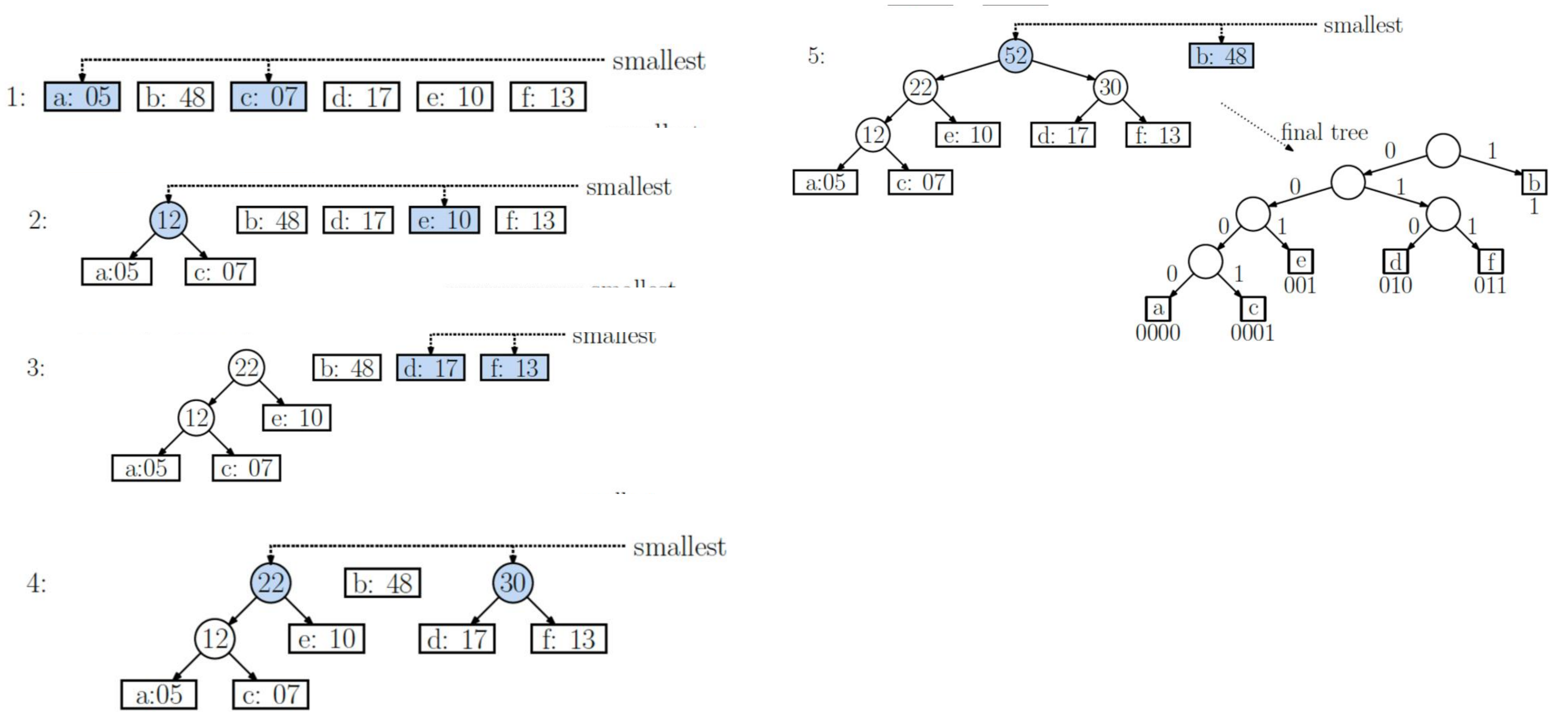
- *What if we knew the tree structure of the optimal prefix code?*
  - Suppose u and v are leaves and depth(u) < depth(v)
  - We should label u and v with x and y if $p(x) \geq p(y)$

- This leads us to an algorithm... If we had the tree structure:
  - Label the leaf nodes with the deepest depth with the least probability letters
  - Then, move the the next level and label with the next least probable letters
  - .... and so on
  - High probability tokens will be at the lowest depth (lower encoding length)
  - Among labels we assign to a block of leaves of the same depth, it doesn't matter which label we assign to each leaf

Huffman’s Algorithm

- Build the tree from the bottom up
- In order of increasing frequency (starting with least probable), merge each pair of children (x and y) into a parent node
  - The parent assumes the sum of x and y’s probability and replaces both
  - Continue to merge...
- When the tree is complete, assign code by appending 0 to the left child and 1 to the right child

Huffman's Algorithm



Huffman's Algorithm

```
// C is a list of chars with associated probabilities
huffman(Node[] C){
  for each (x in C){
    add x to Q sorted by x.prob
  }
  n = size of C
  for (i = 1 to n-1) {
    z = new Node
    z.left = x = extractMin from Q
    z.right = y = extractMin from Q
    z.prob = x.prob + y.prob
    insert z into Q
   }
   return last element in Q as root
}
```

p(a) = .32

p(b) = .25

p(c) = .20

p(d) = .18

p(e) = .05

Huffman's Algorithm - Runtime Analysis

```
// C is a list of chars with associated probabilities
huffman(Node[] C){
  for each (x in C){
    add x to Q sorted by x.prob
  }
  n = size of C
  for (i = 1 to n-1) {
    z = new Node
    z.left = x = extractMin from Q
    z.right = y = extractMin from Q
    z.prob = x.prob + y.prob
    insert z into Q
   }
   return last element in Q as root
}
```

O(n)

n iterations

O(logn)

O(logn)

O(logn)

Total?
O(nlogn)

Proof of Correctness

Important Lemmas:

1. An optimal prefix code tree will be a full binary tree
2. Huffman’s produces a full binary tree
3. Two characters with the lowest frequencies will be siblings at the max depth of any optimal tree

Proof of Correctness

Lemma: An optimal prefix code tree will be a full binary tree

Proof: by contradiction.

Suppose there is a tree T that is optimal, but not a full binary tree.

T has at least one internal node u with a single child v.

By the definition of prefix trees, characters are represented by leaf nodes.

We should simply delete u and replace it with its single child v. Nothing else is affected and v’s depth is reduced by 1, resulting in a more optimal tree. Contradiction.

Proof posted on webpage

Proof of Correctness

Lemma: Huffman's produces a full binary tree

Huffman's clearly produces a full binary tree

Each node it adds has 2 children by construction

```
// C is a list of chars with associated probabilities
huffman(Node[] C){
  for each (x in C){
    add x to Q sorted by x.prob
  }
  n = size of C
  for (i = 1 to n-1) {
    z = new Node
    z.left = x = extractMin from Q
    z.right = y = extractMin from Q
    z.prob = x.prob + y.prob
    insert z into Q
   }
   return last element in Q as root
}
```

Proof of Correctness

Lemma: Two characters with the lowest frequencies will be siblings at the max depth of any optimal tree

Proof by contradiction: Suppose x and y are two characters with the lowest probabilities and T is an optimal prefix code for S which has two siblings b and c at max depth of T, where b and c are distinct from x and y.

Suppose WLOG that $p(c) \geq p(b)$ and $p(x) \geq p(y)$.

$p(c) \geq p(b) > p(x) \geq p(y)$ because x and y have lowest probabilities

If we swap b and x to construct a T', $B(T) > B(T')$

If we swap c and y to construct a T'' where x and y are siblings at max depth, $B(T') > B(T'')$

Full proof posted on webpage

Proof of Correctness

Huffman's Proof: will show the optimality of Huffman coding by induction on the size of the alphabet S.

Base case: |S| = 1

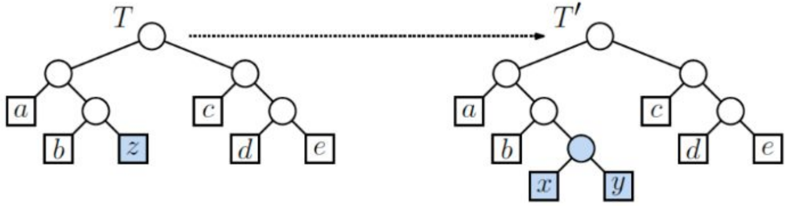
IH: assume for |S| = n - 1 Huffmans produces an optimal encoding

Proof: |S| = n

We will show this by considering an x,y in S with lowest probabilities.

Consider a tree T with x and y replaced with a z such that p(z) = p(x) + p(y)

By IH, T is optimal.



$B(T') = B(T) - p(z)d_T(z) + p(x)d_{T'}(x) + p(y)\, d_{T'}(y)$

$B(T') = B(T) + p(x) + p(y)$ and $B(T)$ is optimal by the IH.

Full proof posted on webpage

Summary

1. Huffman's coding
    a. Bottom up merging of lowest probability characters
    b. O(nlogn) runtime
    c. Proved optimal by proving properties of the optimal prefix code tree
2. Upcoming deadlines:
    a. Lab 3 due tomorrow
    b. Project checkpoint 1 (10/5)
    c. HW4 due monday (10/6)
    d. Midterm (10/8)
3. Next Class: Midterm review