Analysis of Algorithms
CSOR W4231

Eleni Drinea
Computer Science Department

Columbia University

Data compression and huffman coding

Outline

Today

Motivation

Data compression: find compact representations of data

Data compression standards

- `jpeg` for image transmission
- `mp3` for audio content, `mpeg2` for video transmission
- utilities: `gzip`, `bzip2`

All of the above use the Huffman algorithm as a basic building block.

Data representation

- An organism's genome consists of *chromosomes* (giant linear DNA molecules)
- *Chromosome maps*: sequences of hundreds of millions of bases (symbols from $\{A, C, G, T\}$).
- **Goal:** store a chromosome map with 200 million bases.

How do we represent a chromosome map?

Data representation

- An organism's genome consists of *chromosomes* (giant linear DNA molecules)
- *Chromosome maps*: sequences of hundreds of millions of bases (symbols from $\{A, C, G, T\}$).
- **Goal:** store a chromosome map with 200 million bases.

How do we represent a chromosome map?

- Encode every symbol that appears in the sequence separately by a fixed length binary string.
- Codeword $c(x)$ for symbol x: a binary string encoding x of length $\ell(x)$

↑ the length is the # of bits in the codeword

Example code

- Alphabet $\mathcal{A} = \{A, C, G, T\}$ with 4 symbols
- Encode each symbol with 2 bits

alphabet symbol x	codeword $c(x)$
A	00
C	01
G	10
T	11

* We call this collection of codewords a code.

Example code

- Alphabet $\mathcal{A} = \{A, C, G, T\}$ with 4 symbols
- Encode each symbol with 2 bits

alphabet symbol x	codeword $c(x)$
A	00
C	01
G	10
T	11

Output of encoding: the concatenation of the codewords for every symbol in the input sequence

Example code

- Alphabet $\mathcal{A} = \{A, C, G, T\}$ with 4 symbols
- Encode each symbol with 2 bits

alphabet symbol x	codeword $c(x)$
A	00
C	01
G	10
T	11

Output of encoding: the concatenation of the codewords for every symbol in the input sequence

Example

- Input sequence: $ACGTAA$

Codeword : 000110110000

length : 12

Example code

- Alphabet $\mathcal{A} = \{A, C, G, T\}$ with 4 symbols
- Encode each symbol with 2 bits

alphabet symbol x	codeword $c(x)$
A	00
C	01
G	10
T	11

Output of encoding: the concatenation of the codewords for every symbol in the input sequence

Example

- Input sequence: $ACGTAA$
- Output: $c(A)c(C)c(G)c(T)c(A)c(A) = 000110110000$
- Total length of encoding $= 6 \cdot 2 = 12$ bits.

Today

Symbol codes

Symbol code: a set of codewords where every input symbol is encoded **separately**.

Symbol codes

Symbol code: a set of codewords where every input symbol is encoded **separately**.

Examples of symbol codes

- $C_0 = \{00, 01, 10, 11\}$ is a symbol code for $\{A, C, G, T\}$.
- ASCII encoding system: every character and special symbol on the computer keyboard is encoded by a different 7-bit binary string.

Symbol codes

Symbol code: a set of codewords where every input symbol is encoded **separately**.

Examples of symbol codes

- $C_0 = \{00, 01, 10, 11\}$ is a symbol code for $\{A, C, G, T\}$.
- ASCII encoding system: every character and special symbol on the computer keyboard is encoded by a different 7-bit binary string. $\Rightarrow 2^7$ possible symbols can be encoded by ASCII

Remark 1.

C_0 and ASCII are fixed-length symbol codes: each codeword has the same length.

Unique decodability

Decoding C_0?

alphabet symbol x	codeword $c(x)$
A	00
C	01
G	10
T	11

→ 1011110001010 1 ?

G T T A C C C

Decoding algorithm for C_0 :

→ Read bit string from left to right, 2 bits at a time and decode 2 bits at a time.

Decode ASCII ?

→ Read 7 bits at a time.

Unique decodability

Decoding C_0

- read 2 bits of the output;
- print the symbol corresponding to this codeword;
- continue with the next 2 bits.

Unique decodability

Decoding C_0

- read 2 bits of the output;
- print the symbol corresponding to this codeword;
- continue with the next 2 bits.

Remark 2.

- *This decoding algorithm works for ASCII (replace 2 by 7)*
- *C_0, ASCII: distinct input sequences have distinct encodings*

$\bar{x} = x_1 x_2 \cdots x_i \cdots x_m$

$\bar{y} = y_1 y_2 \cdots y_i \cdots y_m$

$x_i \neq y_i$

$c(\bar{x}) = c(x_1)\, c(x_2) \cdots c(x_i) \cdots c(x_m)$

$c(\bar{y}) = c(y_1)\, c(y_2) \cdots c(y_i) \cdots c(y_m)$

Unique decodability

Decoding C_0

- read 2 bits of the output;
- print the symbol corresponding to this codeword;
- continue with the next 2 bits.

Remark 2.

- *This decoding algorithm works for ASCII (replace 2 by 7)*
- *C_0, ASCII: distinct input sequences have distinct encodings*

Definition 1.

A symbol code is uniquely decodable if, for any two distinct input sequences, their encodings are distinct.

Lossless compression

- Lossless compression: compress and decompress without errors.
- Uniquely decodable codes allow for lossless compression.
- A symbol code achieves optimal lossless compression when it produces an encoding of minimum length for its input (among all uniquely decodable symbol codes).
- Huffman algorithm: provides a symbol code that achieves optimal lossless compression.

Fixed-length vs variable-length codes

Chromosome map consists of 200 million bases as follows:

alphabet symbol x	frequency $freq(x)$
A	110 million
C	5 million
G	25 million
T	60 million

Will C_0 achieve optimal lossless compression in the sense that it will produce an output of minimum length?

No; it assigns the same # of bits to each symbol, where each symbol appears @ a diff. freq.

Fixed-length vs variable-length codes

Chromosome map consists of 200 million bases as follows:

alphabet symbol x	frequency $freq(x)$
A	110 million
C	5 million
G	25 million
T	60 million

- A appears much more often than the other symbols.

$\Rightarrow$ It might be best to encode A with fewer bits.

- Unlikely that the fixed-length encoding C_0 is optimal

Today

Prefix codes

Variable-length encodings

Code C_1

alphabet symbol x	codeword $c(x)$
A	0
C	00
G	10
T	1

The issue with this encoding is that it is NOT uniquely decodable. Consider 0 0 0 . . . , this could be A A A or C A or A C . . .

Prefix codes

Variable-length encodings

Code C_1

alphabet symbol x	codeword $c(x)$
A	0
C	00
G	10
T	1

- C_1 is not unique decodable! E.g., 101110: *how to decode it?*

Not a prefix code since 0 is a prefix of 00 and 1 is a prefix of 10

Prefix codes

Variable-length encodings

Code C_2

alphabet symbol x	codeword $c(x)$
A	0
C	110
G	111
T	10

Prefix codes

Variable-length encodings

Code C_2

alphabet symbol x	codeword $c(x)$
A	0
C	110
G	111
T	10

- C_2 is uniquely decodable.
- C_2 is such that no codeword is a prefix of another.

$S = S' \circ t$, S' is a prefix of S if you can write S as the concatenation of S' followed by t, where t is possibly empty.

Prefix codes

Variable-length encodings

Code C_2

alphabet symbol x	codeword $c(x)$
A	0
C	110
G	111
T	10

- C_2 is uniquely decodable.
- C_2 is such that no codeword is a prefix of another.

Definition 2 (prefix codes).

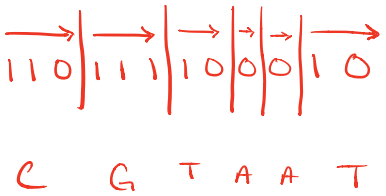
A symbol code is a prefix code if no codeword is a prefix of another.

C_2

Code C_2

alphabet symbol x	codeword $c(x)$
A	0
C	110
G	111
T	10

(prefix code) ↑



Decoding Alg: Start by evaluating the first bit, see it maps to a symbol. If it doesn't, take 2 bits and check if it maps, and so on...

Suffix code

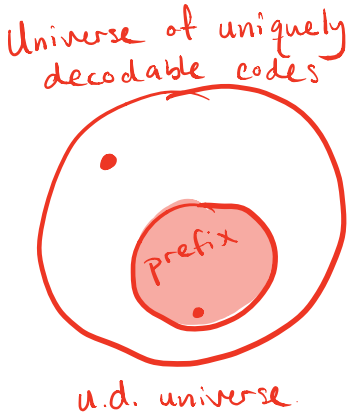
A	0
C	011
G	111
T	01

← suffix code

and it is uniquely decodable

- No code is the suffix of another.

u.d.



The universe of uniquely decodable codes is much larger than just the prefix codes.

Suffix codes lie in this universe.

Consider the code: A 1
B 101

Not a prefix code or a suffix code, and yet it is uniquely decodable. Uniquely decodable codes do not have to be prefix or suffix codes.

However, prefix codes can be decoded fast.

To find an optimal lossless compression code, we should be looking in the universe of u.d. codes. More, we should be looking for prefix codes since they can be decoded fast.

Consider a code $C = \{c(1), c(2), \ldots, c(n)\}$

$a_1 \quad a_2 \quad \ldots \quad a_n$

$\downarrow \quad \downarrow \quad \quad \downarrow$

$c(1), c(2), \ldots, c(n)$

$\downarrow \quad \downarrow \quad \quad \downarrow$

$l_1 \quad l_2 \quad \ldots \quad l_n$

- If C is uniquely decodable (u.d.), then

$$\sum_{i=1}^{n} 2^{-l_i} \leq 1$$ (McMillan's Inequality)

- If $\sum_{i=1}^{n} 2^{-l_i} \leq 1$, then $\exists$ a prefix code with the same codeword length (Kraft's Inequality Thm.)

Why is this important? Prefix code is best for our purposes, and we want to preserve the length of the optimal output encoding.

Goal / Objective:

- Minimize the total length of the output encoding

$$\min S = \sum_{i=1}^{n} f_i \cdot l_i$$

Want to minimize S

for some alphabet $A = \{a_1, \ldots, a_n\}$ with frequencies $F = \{f_1, \ldots, f_n\}$

and l_i is the length of the codeword for each symbol $a_i \in A$.

- Equivalently, minimize the average codeword length. Say there are N symbols in the input sequence. Then $p_i = \frac{f_i}{N}$.

$$\min S = \sum_{i=1}^{n} p_i \cdot N \cdot l_i = N \cdot \sum_{i=1}^{n} p_i \cdot l_i$$

Sum over all possible codeword lengths of the prob. that, that codeword appears times the codeword length.

average codeword length

- Codes with the same codeword length have the same average codeword length.

Decoding prefix codes

1. Scan the binary string from left to right until you've seen enough bits to match a codeword;
2. Output the symbol corresponding to this codeword.
   - Since no other codeword is a prefix of this codeword or contains it as a prefix, this sequence of bits cannot be used to encode any other symbol.
3. Continue starting from the next bit of the bit string.

Decoding prefix codes

1. Scan the binary string from left to right until you've seen enough bits to match a codeword;
2. Output the symbol corresponding to this codeword.
   - Since no other codeword is a prefix of this codeword or contains it as a prefix, this sequence of bits cannot be used to encode any other symbol.
3. Continue starting from the next bit of the bit string.

Thus prefix codes allow for

- unique decoding;
- fast decoding (the end of a codeword is instantly recognizable).

Decoding fixed-length codes is faster, but the goal is to obtain a minimum length with lossless compression.

Decoding prefix codes

1. Scan the binary string from left to right until you've seen enough bits to match a codeword;
2. Output the symbol corresponding to this codeword.
   - Since no other codeword is a prefix of this codeword or contains it as a prefix, this sequence of bits cannot be used to encode any other symbol.
3. Continue starting from the next bit of the bit string.

Thus prefix codes allow for

- unique decoding;
- fast decoding (the end of a codeword is instantly recognizable).

Examples of prefix codes: C_0, C_2

Prefix codes and optimal lossless compression

- Decoding a prefix code is very fast.

$\Rightarrow$ Would like to focus on prefix codes (rather than **all** uniquely decodable symbol codes) for achieving optimal lossless compression.

- Information theory guarantees this: for every uniquely decodable code, exists a prefix code with the **same codeword lengths**
- So we can solely focus on prefix codes for optimal compression.

Compression gains from variable-length prefix codes

Chromosome map: *do we gain anything by using C_2 instead of C_0 when compressing the map of 200 million bases?*

Input

symbol x	$freq(x)$
A	110 million
C	5 million
G	25 million
T	60 million

Code C_0

x	$c(x)$
A	00
C	01
G	10
T	11

Code C_2

x	$c(x)$
A	0
C	110
G	111
T	10

$1 \cdot 110$ mil
$+ 3 \cdot 5$ mil
$+ 3 \cdot 25$ mil
$+ 2 \cdot 60$ mil

- C_0 : Total length $= 2 \cdot 200$ mil
- C_2 : Total length $= 320$ mil

Avg. codeword length

2 bits

1.6 bits $= \frac{320}{200}$

Compression gains from variable-length prefix codes

Chromosome map: *do we gain anything by using C_2 instead of C_0 when compressing the map of* 200 *million bases?*

Input

symbol x	$freq(x)$
A	110 million
C	5 million
G	25 million
T	60 million

Code C_0

x	$c(x)$
A	00
C	01
G	10
T	11

Code C_2

x	$c(x)$
A	0
C	110
G	111
T	10

- C_0: 2 bits $\times$200 million symbols $= 400$ million bits
- C_2: $1 \cdot 110 + 3 \cdot 5 + 3 \cdot 25 + 2 \cdot 60 = 320$ million bits
- Improvement of 20% in this example

The optimal prefix code problem

Input:

- Alphabet $\mathcal{A} = \{a_1, \ldots, a_n\}$
- Set $P = \{p_1, \ldots, p_n\}$ of empirical probabilities over $\mathcal{A}$ such that $p_i = \Pr[a_i]$

Output: a binary prefix code $C^* = \{c(a_1), c(a_2), \ldots, c(a_n)\}$ for $(\mathcal{A}, P)$, where codeword $c(a_i)$ has length ℓ_i and is such that its expected length

$$L(C^*) = \sum_{a_i \in \mathcal{A}} p_i \cdot \ell_i$$

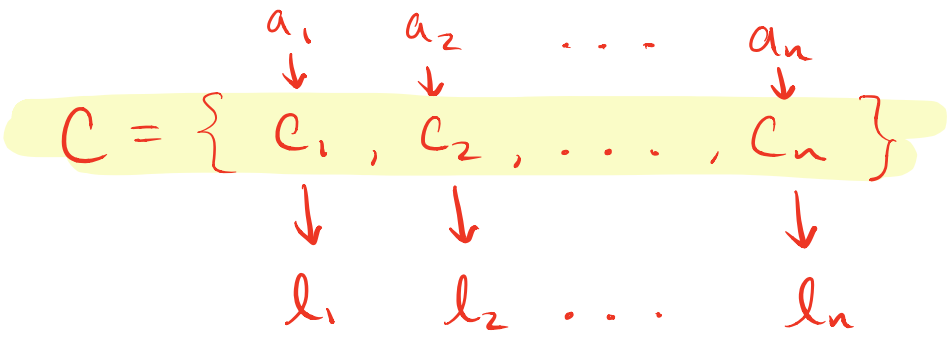
is **minimum** among all binary prefix codes.

$$E[x] = \sum_x x \cdot p_x$$

Input: $\{a_1, a_2, \ldots, a_n\}$

$\{p_1, p_2, \ldots, p_n\}$

Output: A binary prefix code C



that MINIMIZES

$$\sum_{i=1}^{n} p_i \cdot l_i$$

avg. codeword length

Example

Chromosome example

Input

symbol x	$\Pr(x)$
A	110/200
C	5/200
G	25/200
T	60/200

Code C_0

x	$c(x)$
A	00
C	01
G	10
T	11

Code C_2

x	$c(x)$
A	0
C	110
G	111
T	10

- $L(C_0) = 2$
- $L(C_2) = 1.6$
- *Coming up:* C_2 is the output of the Huffman algorithm, hence an optimal encoding for $(\mathcal{A}, P)$.

Today

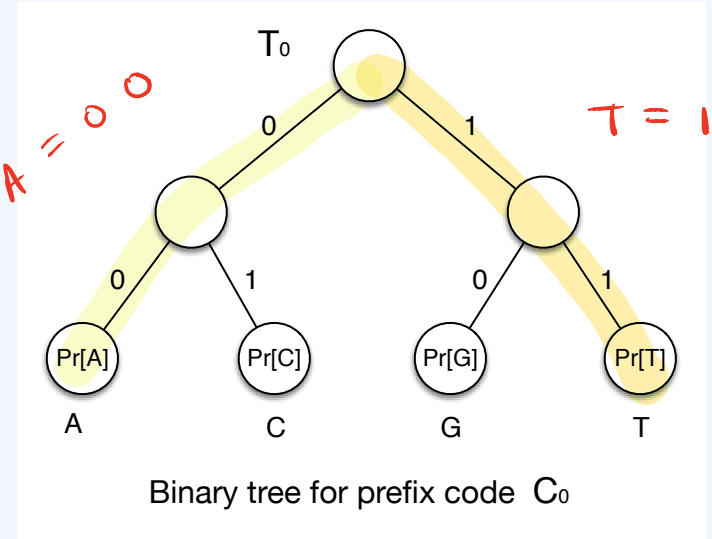
To solve this problem, we observe there is a natural correspondence btwn binary prefix codes and binary trees, so you can represent a binary prefix code as a binary tree. Therefore, you can express the codeword length in terms of tree quantities.

Prefix codes and trees

- A binary tree T is a rooted tree such that each node that is not a leaf has at most two children.
- Binary tree for a prefix code: a branch to the left represents a 0 in the encoding and a branch to the right a 1.

Code C_0

x	$c(x)$
A	00
C	01
G	10
T	11



Binary tree for prefix code C_0

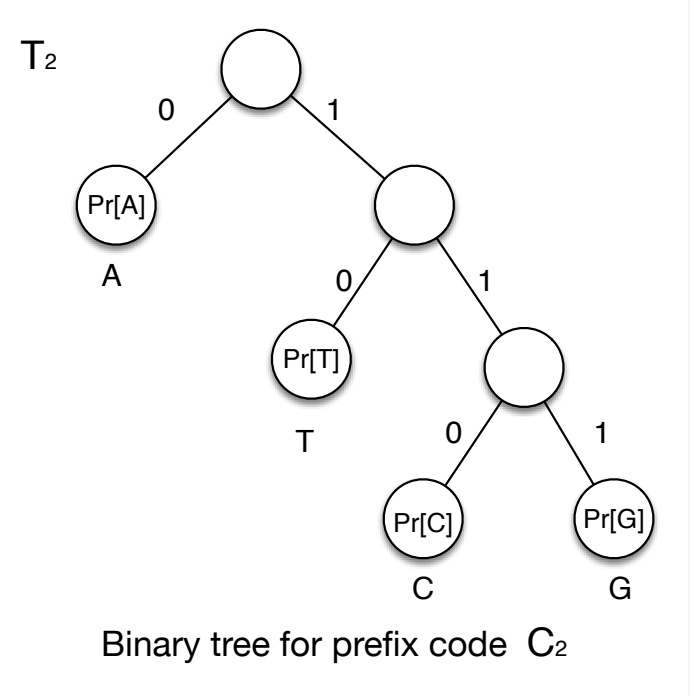
Prefix codes and trees

- A binary tree T is a rooted tree such that each node that is not a leaf has at most two children.
- Binary tree for a prefix code: a branch to the left represents a 0 in the encoding and a branch to the right a 1.

The depth of the tree gives you the length of the codeword.

Code C_2

x	$c(x)$
A	0
C	110
G	111
T	10



Binary tree for prefix code C_2

Now, we can express the avg. codeword length in terms of tree quantities.:

- Alphabet symbols appear at leaves of the tree
- Codewords correspond to root to leaf paths.
- The length of the codeword corresponds to the depth of the symbol in the tree.

$$L(C) = \sum_{i=1}^{n} p_i \cdot l_i$$

$$= \sum_{i=1}^{n} p_i \cdot d_T(\alpha_i)$$

$$\triangleq L(T)$$

↖ avg. codeword length.

Where C denotes the code, and T the tree.

Properties of binary trees representing prefix codes

1. *Where do alphabet symbols appear in the tree?*
2. *What do codewords correspond to in the tree?*
3. *Consider the tree corresponding to the optimal prefix code. Can it have internal nodes with one child?*

Properties of binary trees representing prefix codes

1. Symbols must appear at the leaves of the tree T (*why?*)
   $\Rightarrow$ T has n leaves.

2. Codewords $c(a_i)$ are given by root-to-leaf paths.

   Recall that ℓ_i is the length of the codeword $c(a_i)$ for input symbol a_i. Therefore, on the tree T, ℓ_i corresponds to the depth of a_i (we assume that the root is at depth 0).

   $\Rightarrow$ Can rewrite the **expected length** of the prefix code as:

$$L(C) = \sum_{a_i \in \mathcal{A}} p_i \cdot \ell_i = \sum_{1 \leq i \leq n} p_i \cdot \text{depth}_T(a_i) = L(T).$$

3. Optimal tree must be full: all internal nodes must have exactly two children (*why?*).

Otherwise, we could then have a tree w/ smaller depth by removing all the nodes w/ one 1 child.

More on optimal tree

Claim 1.

There is an optimal prefix code, with corresponding tree T^*, in which the two lowest frequency characters are assigned to leaves that are siblings in T^* at maximum depth.

- The higher the frequency, the lower the depth of the leaf.
- The more infrequent the symbol, the larger the codeword length we can assign to them. Since they're very infrequent it's okay to give them longer codeword lengths.. Large codeword lengths correspond to greater depths in the binary tree.

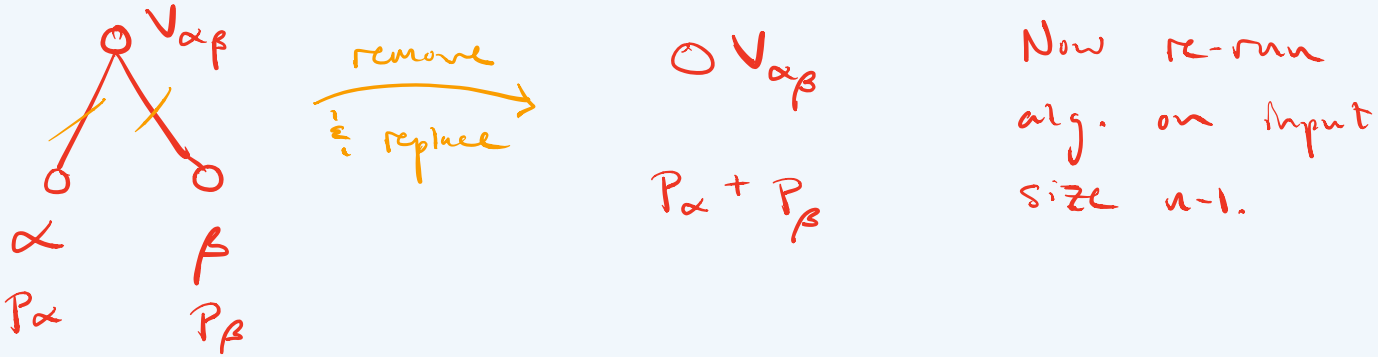
More on optimal tree

Claim 1.

There is an optimal prefix code, with corresponding tree T^*, in which the two lowest frequency characters are assigned to leaves that are siblings in T^* at maximum depth.

Proof.

By an exchange argument: start with a tree for an optimal prefix code and transform it into T^*. □



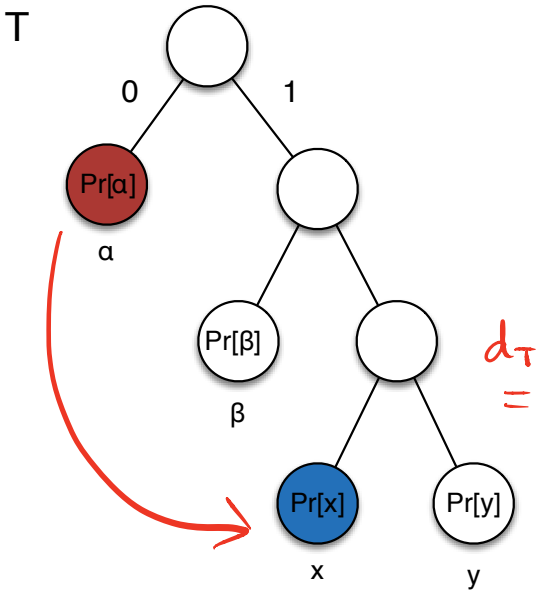
This is a greedy approach because it makes myopic choice that optimizes the local criteria.

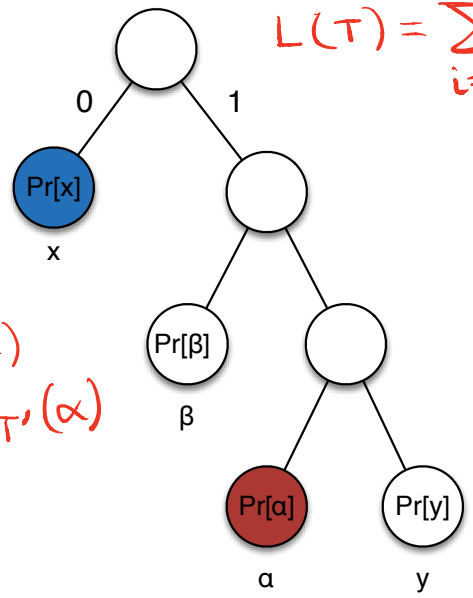
Proof of correctness of greedy algorithms is generally by an exchange argument.

Proof of Claim 1

- Let T be the tree for the optimal prefix code.
- Let α, β be the two symbols with the smallest probabilities, that is, $\Pr[\alpha] \leq \Pr[\beta] \leq \Pr[s]$ for all $s \in \mathcal{A} - \{\alpha, \beta\}$.
- Let x and y be the two siblings at maximum depth in T.

Step 1: Move α to max depth position (swapping w/ x)





$L(T) = \sum_{i=1}^{n} p_i d_T(i)$

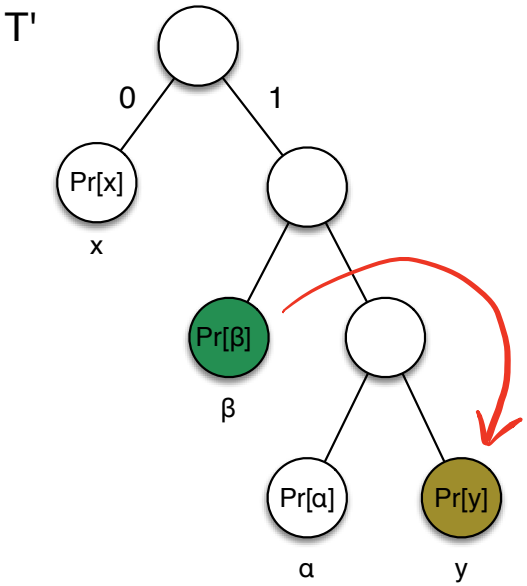
$d_T(x) = d_{T'}(\alpha)$

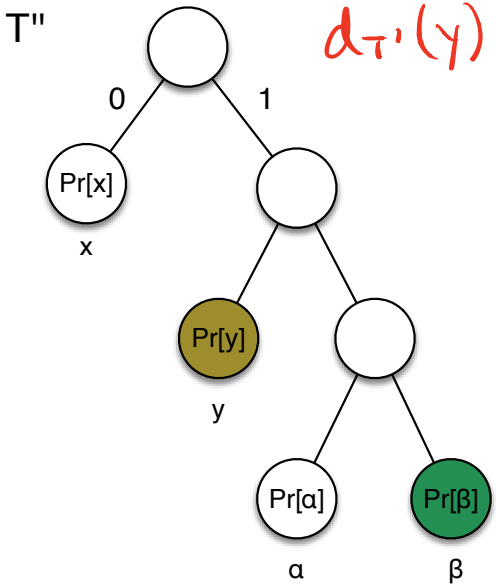
We want L(T) ≥ L(T') $\iff L(T) - L(T') \geq 0$

Proof of Claim 1

- Let T be the tree for the optimal prefix code.
- Let α, β be the two symbols with the smallest probabilities, that is, $\Pr[\alpha] \leq \Pr[\beta] \leq \Pr[s]$ for all $s \in \mathcal{A} - \{\alpha, \beta\}$.
- Let x and y be the two siblings at maximum depth in T.

Step 2: Move β to max depth position (swapping w/ y)





$d_{T'}(y) = d_{T''}(\beta)$

We want $L(T') \geq L(T'')$ $\iff$ $L(T') - L(T'') \geq 0$

How do the expected lengths of the two trees compare?

$$
\begin{aligned}
L(T) - L(T') &= \sum_{a_i \in \mathcal{A}} \Pr[a_i] \cdot \mathrm{depth}_T(a_i) - \sum_{a_i \in \mathcal{A}} \Pr[a_i] \cdot \mathrm{depth}_{T'}(a_i) \\
&= \Pr[\alpha] \cdot \mathrm{depth}_T(\alpha) + \Pr[x] \cdot \mathrm{depth}_T(x) \\
&\quad - \Pr[\alpha] \cdot \mathrm{depth}_{T'}(\alpha) - \Pr[x] \cdot \mathrm{depth}_{T'}(x) \\
&= \Pr[\alpha] \cdot \mathrm{depth}_T(\alpha) + \Pr[x] \cdot \mathrm{depth}_T(x) \\
&\quad - \Pr[\alpha] \cdot \mathrm{depth}_T(x) - \Pr[x] \cdot \mathrm{depth}_T(\alpha) \\
&= \underbrace{(\Pr[\alpha] - \Pr[x])}_{\leq 0} \cdot \underbrace{(\mathrm{depth}_T(\alpha) - \mathrm{depth}_T(x))}_{\leq 0} \geq 0
\end{aligned}
$$

$\Pr[\alpha] \leq \Pr[x]$ and $d_T(\alpha) \leq d_T(x)$ (in T)

- The third equality follows from the exchange.
- Similarly, exchanging β and y in T' yields $L(T') - L(T'') \geq 0$.
- Hence $L(T) - L(T'') \geq 0$.
- Since T is optimal, it must be $L(T) = L(T'')$.
- So T'' is also optimal.

The claim follows by setting T^* to be T''.

Building the optimal tree

Claim 1 tells us how to build the optimal tree greedily!

1. Find the two symbols with the lowest probabilities.
2. Remove them from the alphabet and replace them with a new meta-character with probability equal to the sum of their probabilities.
   - **Idea:** this meta-character will be the parent of the two deleted symbols in the tree.
3. Recursively construct the optimal tree using this process.

Greedy *algorithms: make a local (myopic) decision at every step that optimizes some criterion and eventually show that this is the optimal way for building the entire solution.*

Today

Huffman algorithm

$|\mathcal{A}| = n$

Optimal for symbol codes

$O(n^2)$ { • n rec. calls • $O(n)$

`Huffman`$(\mathcal{A}, P)$

if $|\mathcal{A}| = 2$ **then** — $O(1)$

  Encode one symbol using 0 and the other using 1

end if

(Can get $O(\log n)$ here using a binary min-heap) $O(\log n)$

Let α and β be the two symbols with the lowest probabilities — $O(n)$

Let ν be a new meta-character with probability $\Pr[\alpha] + \Pr[\beta]$ — $O(1)$

Let $\mathcal{A}_1 = \mathcal{A} - \{\alpha, \beta\} + \{\nu\}$ ← $O(\log n)$

Let P_1 be the new set of probabilities over $\mathcal{A}_1$ ← $O(1)$

$T_1 = $ `Huffman`$(\mathcal{A}_1, P_1)$

`return` T as follows: replace leaf node ν in T_1 by an internal node, and add two children labelled α and β below ν. — $O(1)$

Remark 3.

Output of `Huffman` *procedure is a binary tree T; the code for $(\mathcal{A}, P)$ is its corresponding prefix code.*

Huffman algorithm

Running Time w/ Binary min-heap

Use binary min-heap to achieve $O(n \log n)$

`Huffman`$(\mathcal{A}, P)$

if $|\mathcal{A}| = 2$ **then** — $O(1)$

  Encode one symbol using 0 and the other using 1

end if

2 Extract Min-operations — $O(\log n)$

Let α and β be the two symbols with the lowest probabilities

Let ν be a new meta-character with probability $\Pr[\alpha] + \Pr[\beta]$ — $O(1)$

Let $\mathcal{A}_1 = \mathcal{A} - \{\alpha, \beta\} + \{\nu\}$ ← 1 insert — $O(\log n)$

$O(\log n)$

Let P_1 be the new set of probabilities over $\mathcal{A}_1$

$T_1 = \texttt{Huffman}(\mathcal{A}_1, P_1)$ ← Run n times.

return T as follows: replace leaf node ν in T_1 by an internal node, and add two children labelled α and β below ν.

Remark 3.

Output of `Huffman` *procedure is a binary tree T; the code for $(\mathcal{A}, P)$ is its corresponding prefix code.*

STUDY BINARY MIN-HEAP!!

Example: $A = \{A, C, G, T\}$

$P = \left\{\frac{110}{200}, \frac{5}{200}, \frac{25}{200}, \frac{60}{200}\right\}$

Run Huffman(A, P):

```
Huffman(𝒜, P)
  if |𝒜| = 2 then
      Encode one symbol using 0 and the other using 1
  end if
  Let α and β be the two symbols with the lowest probabilities
  Let ν be a new meta-character with probability Pr[α] + Pr[β]
  Let 𝒜₁ = 𝒜 − {α, β} + {ν}
  Let P₁ be the new set of probabilities over 𝒜₁
  T₁ = Huffman(𝒜₁, P₁)
  return T as follows: replace leaf node ν in T₁ by an internal
  node, and add two children labelled α and β below ν.
```

- Find the 2 symbols w/ min probability: C and G
- Introduce a new meta-character

Update alphabet:

$A_1 = \{A, V_{CG}, T\}$

$P_1 = \left\{\frac{110}{200}, \frac{30}{200}, \frac{60}{200}\right\}$

Call Huffman(A_1, P_1):

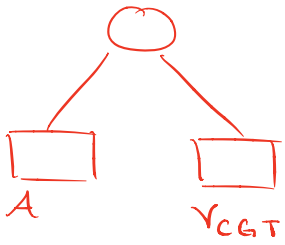
Find next 2 smallest probabilities.

$A_1 = \{A, V_{CGT}\}$

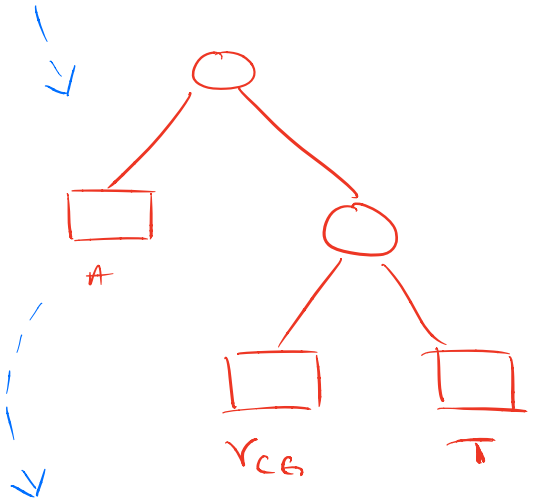
$P_1 = \left\{\frac{110}{200}, \frac{90}{200}\right\}$

Call Huffman(A_1, P_1):

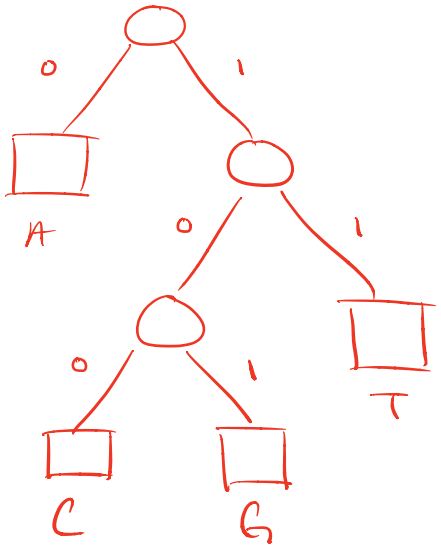
Now start building the tree.



Start to expand V_{CGT} into its own child nodes!



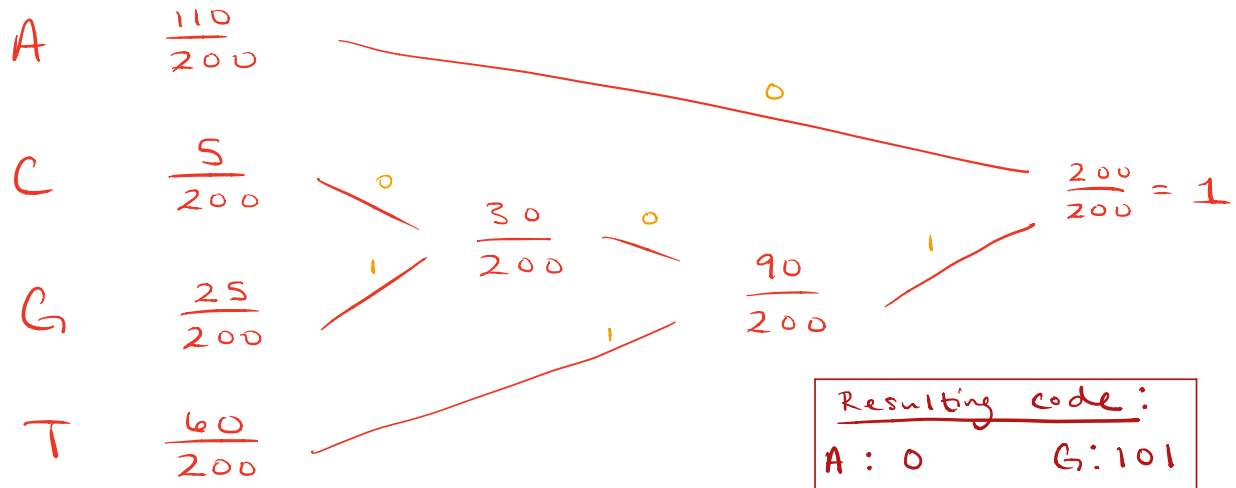
Final Tree



This tree gives you codewords:

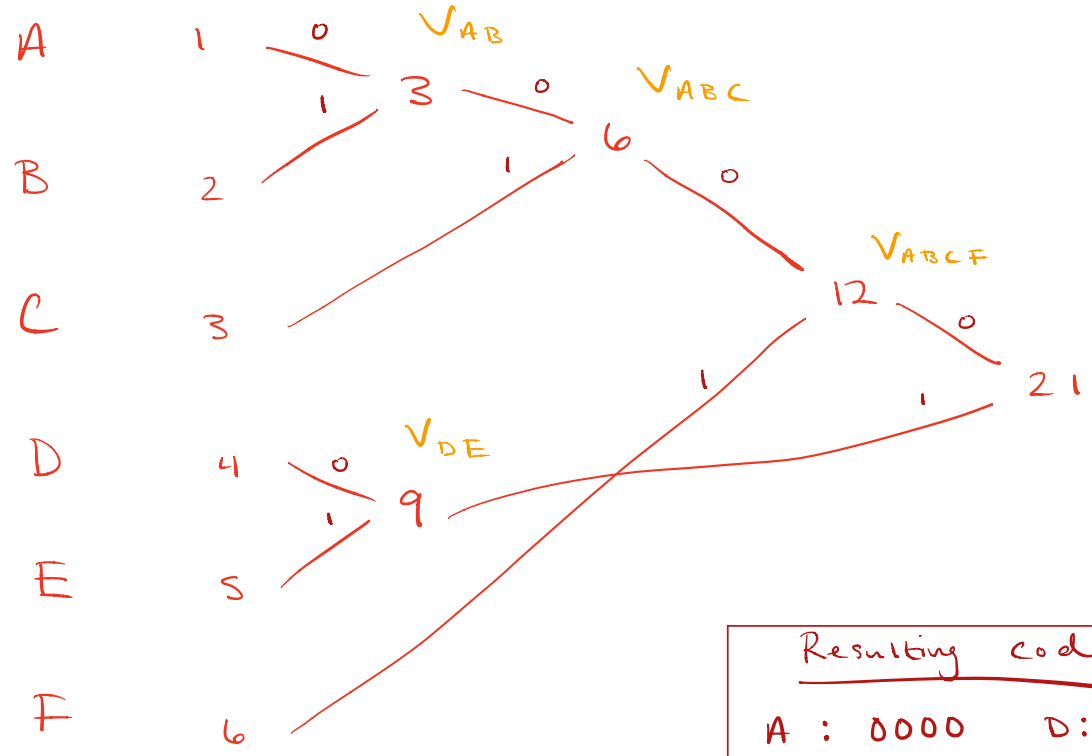
$C = \{0, 100, 101, 11\}$

How Huffman is run in practice:



Resulting code:	
A : 0	G : 101
C : 100	T : 11

Ex 2:



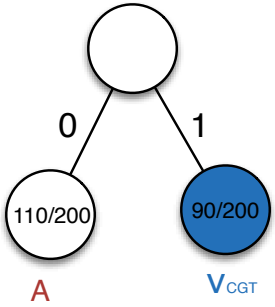
Resulting code:	
A : 0000	D : 10
B : 0001	E : 11
C : 001	F : 01

Example: recursive Huffman for chromosome map

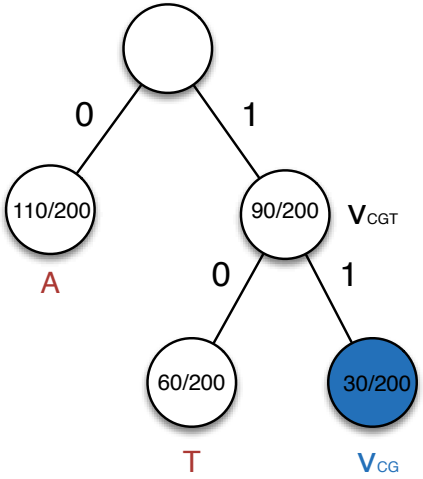
Recursive call 1: $\texttt{Huffman}(\{A, C, G, T\}, \{\frac{110}{200}, \frac{5}{200}, \frac{25}{200}, \frac{60}{200}\})$

Recursive call 2: $\texttt{Huffman}(\{A, \nu_{CG}, T\}, \{\frac{110}{200}, \frac{30}{200}, \frac{60}{200}\})$

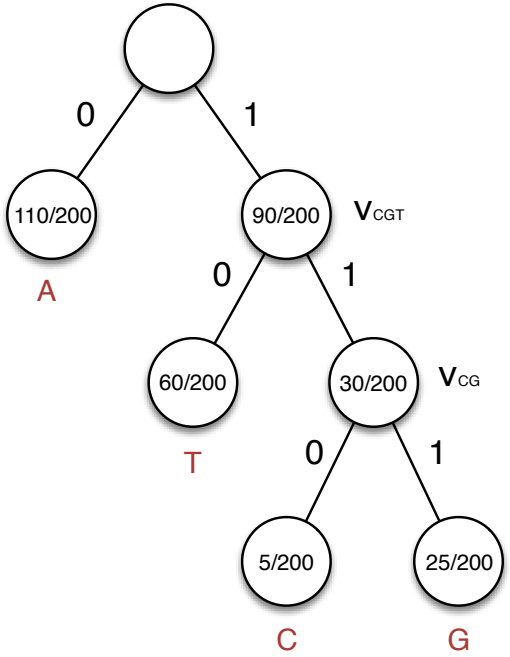
Recursive call 3: $\texttt{Huffman}(\{A, \nu_{CGT}\}, \{\frac{110}{200}, \frac{90}{200}\})$



End of rec. call 3



End of rec. call 2



End of rec. call 1

Correctness

Proof: by induction on the size of the alphabet $n \geq 2$.

- **Base case.** For $n = 2$, Huffman is optimal.
- **Hypothesis.** Assume that Huffman returns the optimal prefix code for alphabets of n symbols.
- **Induction Step.** Let $\mathcal{A}$ be an alphabet of size $n + 1$, P the corresponding set of probabilities.

  Let T_1 be the optimal (by the hypothesis) tree returned by our algorithm for $(\mathcal{A}_1, P_1)$, where $\mathcal{A}_1, P_1, T_1$ as in the pseudocode. Let T be the final tree returned for $(\mathcal{A}, P)$ by our algorithm. We claim that T is optimal.

  We will prove the claim by contradiction. **Assume** T^* is the optimal tree for $(\mathcal{A}, P)$ such that

$$L(T^*) < L(T). \tag{1}$$

A useful fact

Fact 3.

Let T be a binary tree representing a prefix code. If we replace sibling leaves α, β in T by a meta-character ν where $\Pr[\nu] = \Pr[\alpha] + \Pr[\beta]$, we obtain a tree T_1 such that

$$L(T) = L(T_1) + (\Pr[\alpha] + \Pr[\beta]).$$

Proof.

Notation: $d_T(a_i) = \text{depth}_T(a_i)$

- α, β are sibling leaves in T, hence $d_T(\alpha) = d_T(\beta)$.
- T differs from T_1 only in that α, β are replaced by ν. Since $d_{T_1}(\nu) = d_T(\alpha) - 1$, we obtain

$$\begin{aligned} L(T) - L(T_1) &= \Pr[\alpha]d_T(\alpha) + \Pr[\beta]d_T(\beta) - (\Pr[\alpha] + \Pr[\beta])d_{T_1}(\nu) \\ &= \Pr[\alpha] + \Pr[\beta]. \end{aligned} \tag{2}$$

□

Correctness (cont'd)

- Claim 1 guarantees there is such an optimal tree for $(\mathcal{A}, P)$ where α, β appear as siblings at maximum depth.
- W.l.o.g. assume that T^* is such an optimal tree. By Fact 3, if we replace siblings α, β in T^* by ν' where $\Pr[\nu'] = \Pr[\alpha] + \Pr[\beta]$, the resulting tree T_1^* satisfies $L(T^*) = L(T_1^*) + (\Pr[\alpha] + \Pr[\beta])$.
- Similarly, the tree T returned by the Huffman algorithm satisfies $L(T) = L(T_1) + (\Pr[\alpha] + \Pr[\beta])$.
- By the induction hypothesis, we have $L(T_1^*) \geq L(T_1)$ since T_1 is optimal for alphabets of size n. Hence

$$L(T^*) = L(T_1^*) + \Pr[\alpha] + \Pr[\beta] \geq L(T_1) + \Pr[\alpha] + \Pr[\beta] = L(T), \quad (3)$$

  where the inequality follows from the induction hypothesis.
- Equation (3) contradicts Assumption (1). Thus T must be optimal.

Implementation and running time

STUDY THIS!

Read Ch. 6 to understand binary min-heap (or max-heap)

1. Straightforward implementation: $O(n^2)$ time
2. Store the alphabet symbols in a min priority queue implemented as a binary min-heap with keys their probabilities
   - **Operations:** `Initialize` ($O(n)$), `Extract-min` ($O(\log n)$), `Insert` ($O(\log n)$)

   Total time: $O(n \log n)$ time

For an iterative implementation of Huffman, see your textbook.

Ch. 6

Running time w/ binary min-heap : $O(n \log n)$

Running time w/o binary min-heap : $O(n^2)$

Example: iterative Huffman for chromosome map

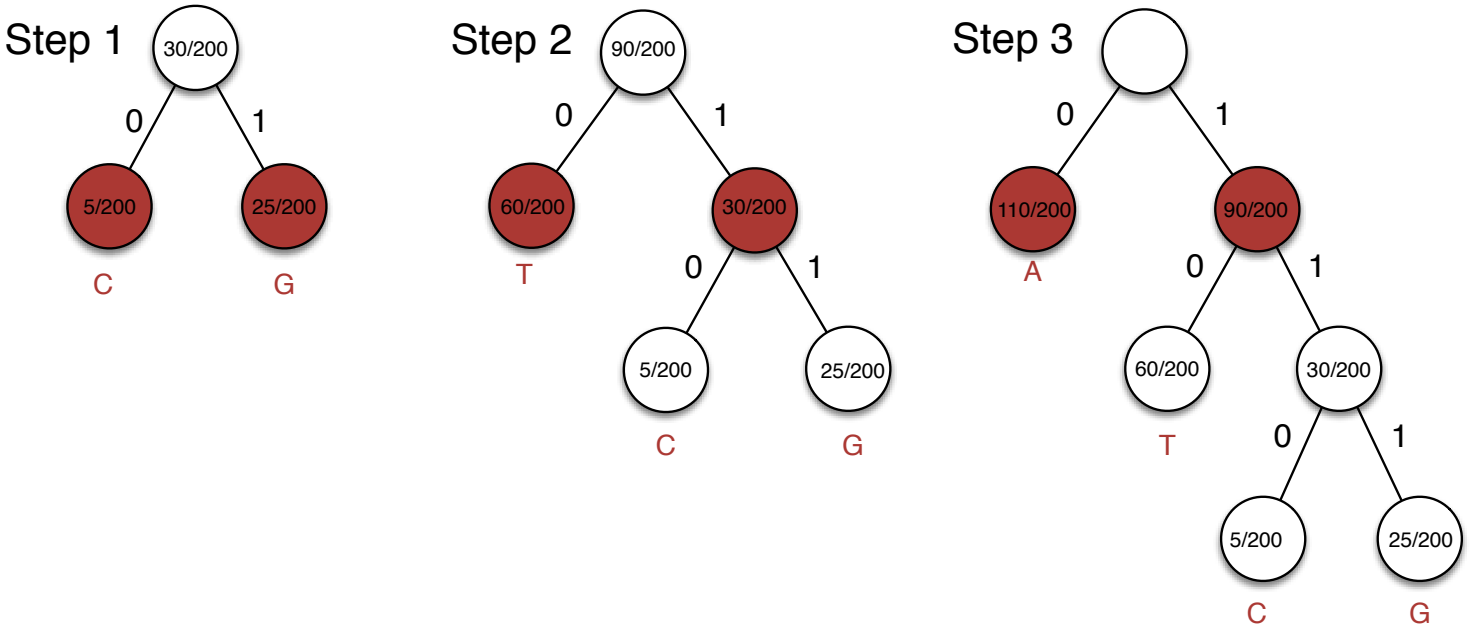
Input $(\mathcal{A}, P)$

symbol x	$\Pr(x)$
A	$110/200$
C	$5/200$
G	$25/200$
T	$60/200$

$\rightarrow$

Output code

symbol x	$c(x)$
A	0
C	110
G	111
T	10



Beyond Huffman coding

- Huffman algorithm provides an optimal symbol code.
- Codes that encode larger blocks of input symbols might achieve better compression.
- Storage on noisy media: *what if a bit of the output of the compressor is flipped?*
  - Decompression cannot carry through.
  - Need error correction on top of compression.