TTIC 31010, CMSC 37000-1

Lecture 3: Huffman Coding and Dynamic Programming

Yury Makarychev

Huffman Coding

Uniquely Decodable Encoding

- The simplest example of a uniquely decodable encoding is a fixed-length code. All codewords have the same length. E.g., in extended ASCII all codewords have length 8:

$$0100100100101010010101010101010101010101000101000$$

- Another example of uniquely decodable code is a prefix code. In a prefix code no codeword is a prefix of another.

prefix

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

$f: a \mapsto 0$
$f: b \mapsto 10$
$f: c \mapsto 11$

codewords: 0, 10,11

not prefix

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

$f: a \mapsto 0$
$f: b \mapsto 1$
$f: c \mapsto 10$

codewords: 0, 1,10
1 is a prefix of 10

Decoding prefix codes

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

$f: a \mapsto 0$
$f: b \mapsto 10$
$f: c \mapsto 11$

codewords: $0, 10, 11$

$$01001111001010100101010101010101010101000101$$

- Scan the string from left to right.
- Once we read a codeword output the correspondent character.

Decoding prefix codes

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

$f: a \mapsto 0$
$f: b \mapsto 10$
$f: c \mapsto 11$

codewords: $0, 10, 11$

$01001111001010100101010101010101010101000101$
a

- Scan the string from left to right.
- Once we read a codeword output the correspondent character.

Decoding prefix codes

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

$f: a \mapsto 0$

$f: b \mapsto 10$

$f: c \mapsto 11$

codewords: $0, 10, 11$

$0100111100101010010101010101010101010101000101$

a

- Scan the string from left to right.
- Once we read a codeword output the correspondent character.

Decoding prefix codes

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

$f: a \mapsto 0$
$f: b \mapsto 10$
$f: c \mapsto 11$

codewords: 0, 10,11

$01001111001010100101010101010101010101000101$
ab

- Scan the string from left to right.
- Once we read a codeword output the correspondent character.

Decoding prefix codes

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

$f: a \mapsto 0$

$f: b \mapsto 10$

$f: c \mapsto 11$

codewords: 0, 10,11

$0100111100101010010101010101010101010101000101$

$abacc$ ...

- Scan the string from left to right.
- Once we read a codeword output the correspondent character.

It's easy to prove by induction that the algorithm outputs the only possible decoding of the binary string.

Prefix codes

- Q: are fixed-length codes prefix codes?
- A: yes

Prefix codes

- Q: are there uniquely decodable codes that are not prefix codes?
- A: yes, suffix codes.

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

$f: a \mapsto 0$

$f: b \mapsto 01$

$f: c \mapsto 11$

codewords: $0, 01, 11$

Optimal Codes

Optimal encoding problem

- Assume that we are given probabilities/frequencies $p_1, \dots, p_n$ with which characters $\sigma_1, \dots, \sigma_n$ appear in texts.
- How many bits do we need to encode a text with the given character frequencies?

$$\mathrm{cost}(f) = p_1|f(\sigma_1)| + \cdots + p_n|f(\sigma_n)|$$

per character of text (here, $|f(\sigma_i)|$ is the length of codeword $f(\sigma_i)$).

The encoding problem: find uniquely decodable f that minimizes $\mathrm{cost}(f)$.

Claim: there is an optimal uniquely decodable code that is a prefix code.

Tree representation

Let f be a prefix code.

- Consider all codewords $C = \{f(\sigma_1), \dots, f(\sigma_n)\}$.
- Further consider all prefixes P of these codewords (other than codewords themselves).
- Create a binary tree on $P \cup C$
  - The tree is rooted at Λ (that is, empty string)
  - String u has children $u0$ and $u1$, if they are present in $P \cup C$
  - Codewords are leaves of the tree
  - Each leaf $f(\sigma_i)$ is labelled with σ_i

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

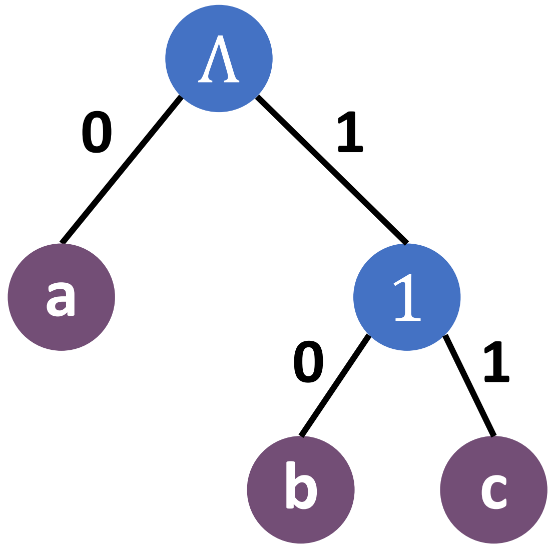
$f: a \mapsto 0$

$f: b \mapsto 10$

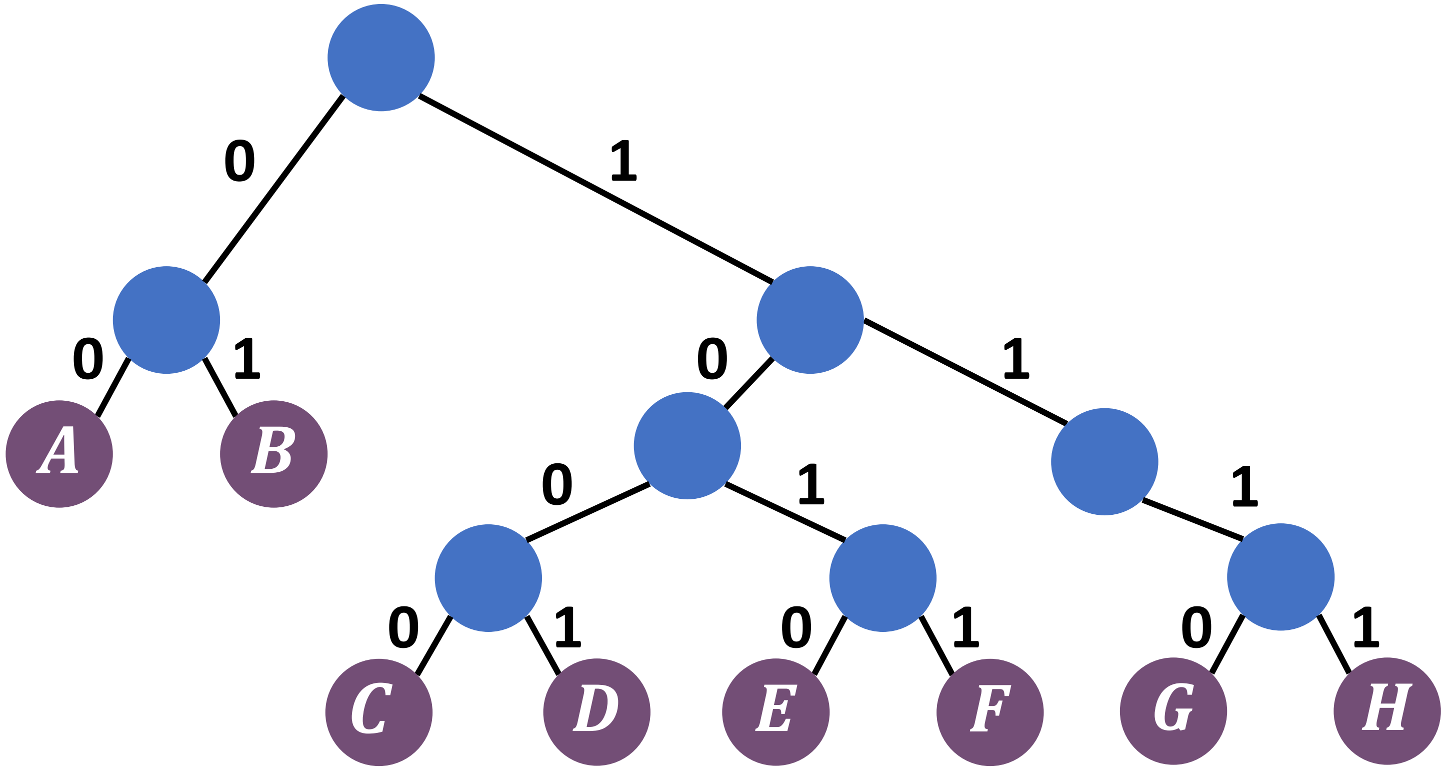
$f: c \mapsto 11$

codewords: $0, 10, 11$

prefixes: $\Lambda, 1$

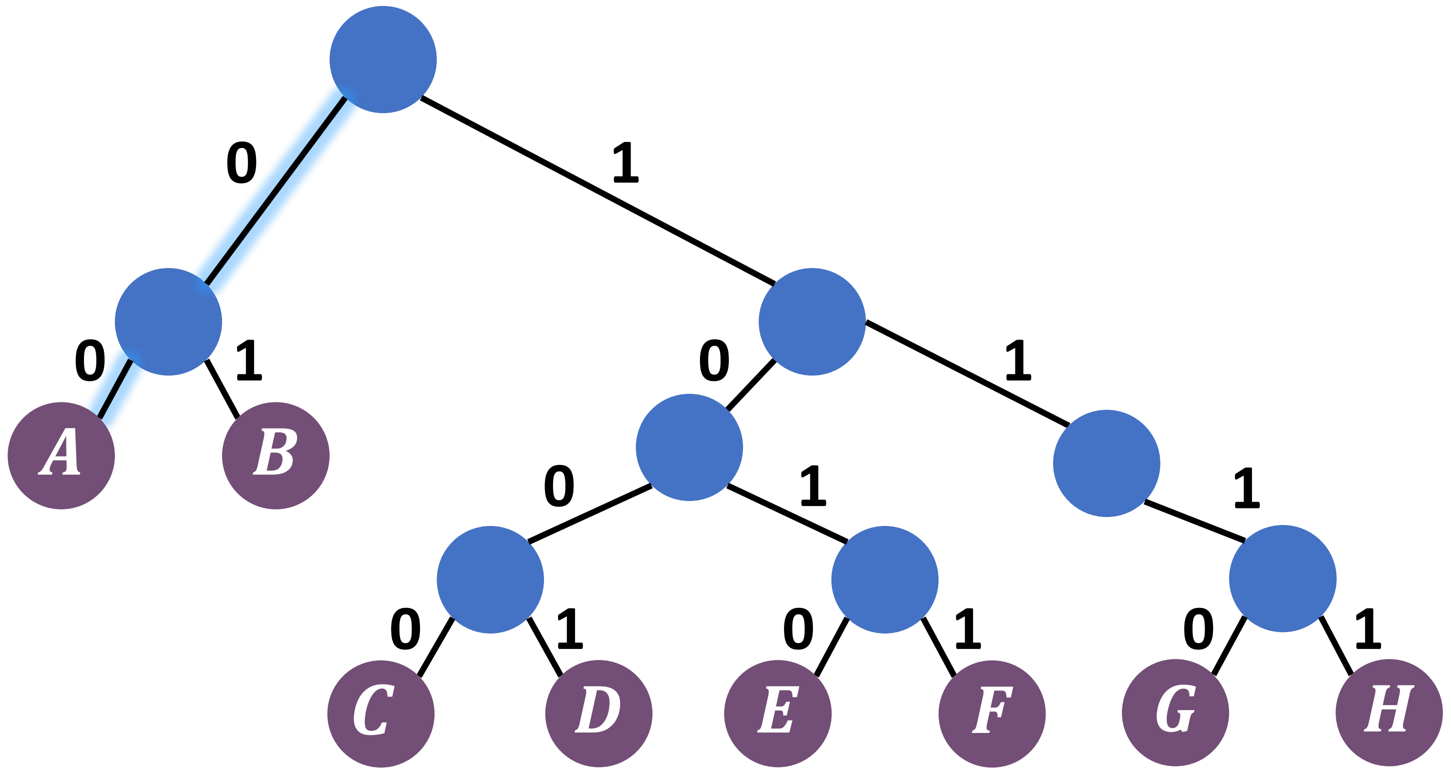


Tree Representation



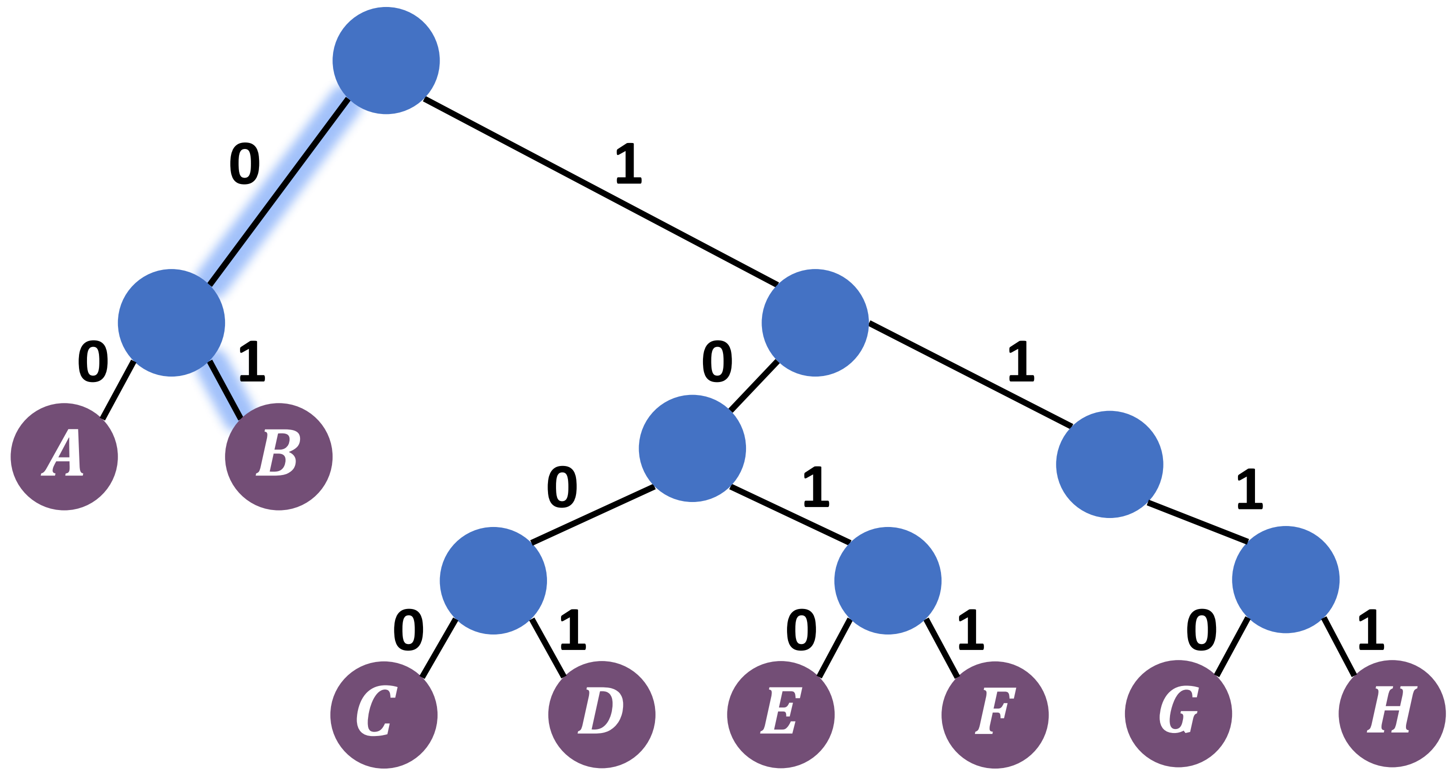
Codewords
$A \mapsto 00$
$B \mapsto 01$
$C \mapsto 1000$ $D \mapsto 1001$ $E \mapsto 1010$ $F \mapsto 1011$ $G \mapsto 1110$ $H \mapsto 1111$

Tree Representation



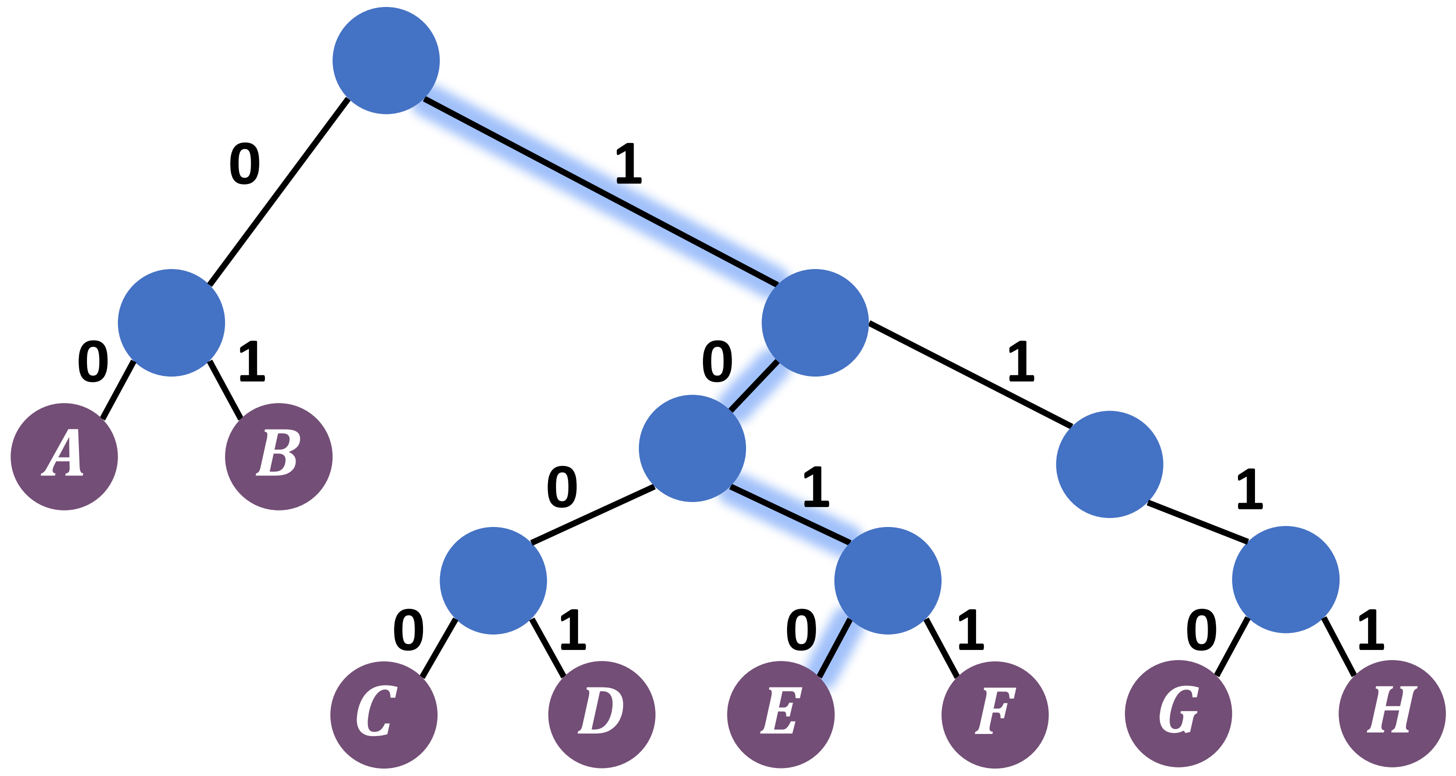
Codewords
$A \mapsto 00$
$B \mapsto 01$
$C \mapsto 1000$ $D \mapsto 1001$ $E \mapsto 1010$ $F \mapsto 1011$ $G \mapsto 1110$ $H \mapsto 1111$

Tree Representation



Codewords
$A \mapsto 00$
$B \mapsto 01$
$C \mapsto 1000$ $D \mapsto 1001$ $E \mapsto 1010$ $F \mapsto 1011$ $G \mapsto 1110$ $H \mapsto 1111$

Tree Representation



Codewords
$A \mapsto 00$
$B \mapsto 01$
$C \mapsto 1000$
$D \mapsto 1001$
$E \mapsto 1010$
$F \mapsto 1011$
$G \mapsto 1110$
$H \mapsto 1111$

Tree Representation

- Each prefix code defines a prefix tree.

Consider two vertices (binary strings) in a prefix tree: B_1 and B_2.

B_1 is a prefix of B_2 if and only if B_1 is an ancestor of B_2

- Given a prefix tree, consider the leaves and their labels. They define a code.

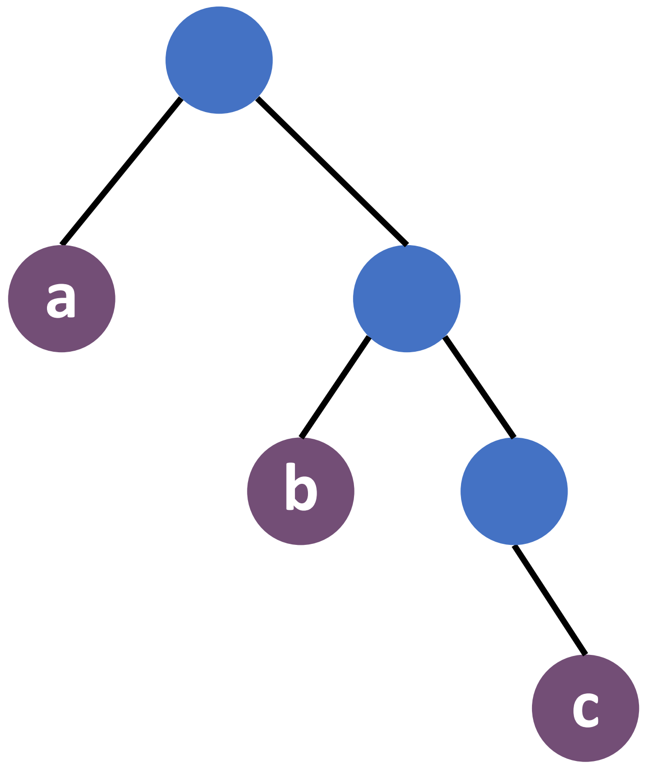
Since no leaf is an ancestor of another, each prefix tree defines a prefix code.

Cost

$$\operatorname{cost}(f) = \sum_i p_i |f(\sigma_i)| = \sum_i p_i \operatorname{depth}(f(\sigma_i))$$

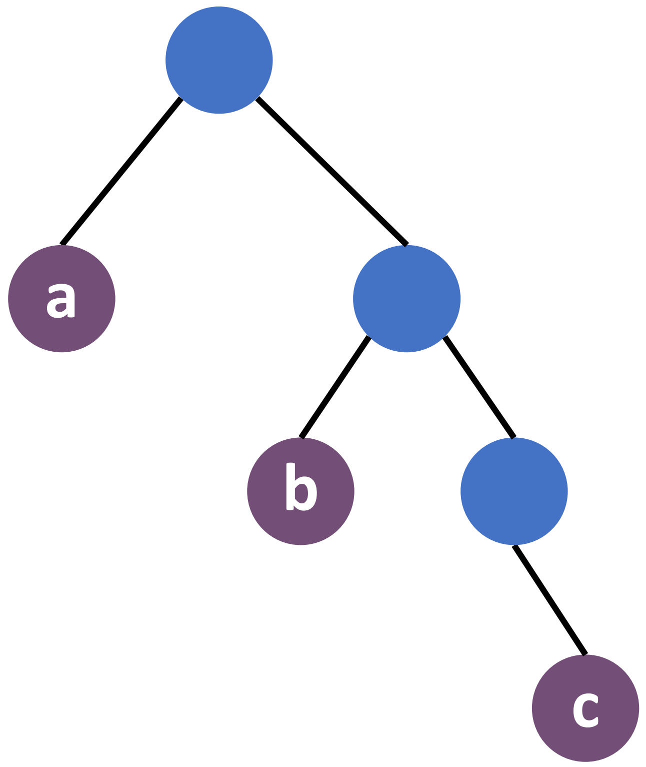
Optimal Binary Tree

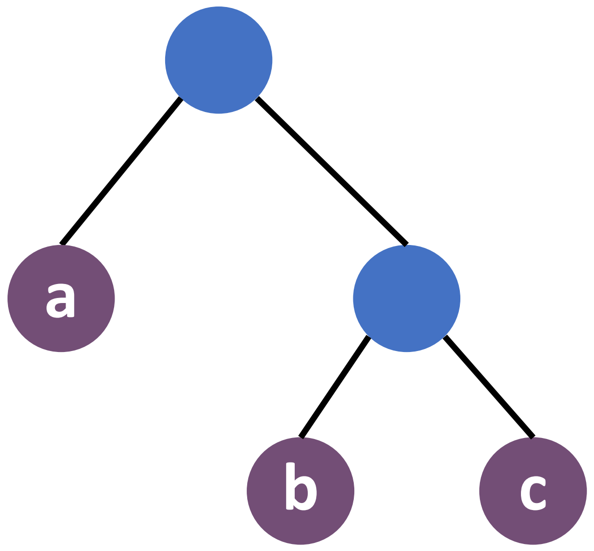
Can this tree be an optimal tree for some code and some set of p_i?



Optimal Binary Tree

Can this tree be an optimal tree for some code and some set of p_i? No!





Optimal Binary Tree

Claim:

- An optimal tree is a full binary tree:
  - all internal nodes (not leaves) have exactly two children.
  - in other words, each node has 0 or 2 children.

Optimal Labeling?

- **Q:** Suppose we are given frequencies $p_1, \ldots, p_n$ and an optimal prefix tree. But the labels of the leaves (codewords) are hidden.

How can we find the optimal labeling of leaves with characters σ_i?

Optimal Labeling?

- **Q:** Suppose we are given frequencies $p_1, \dots, p_n$ and an optimal prefix tree. But the labels of the leaves (codewords) are hidden.

How can we find the optimal labeling of leaves with characters σ_i?

A: Sort all codewords by their length / depth in the tree.
Assign shorter codewords to more frequent letters.

Proof: Assume that leaves at depths d_1 and d_2 are labeled with characters with frequencies $p_1 > p_2$, and $d_1 > d_2$.

Swap the labels of the leaves. We get a better encoding:

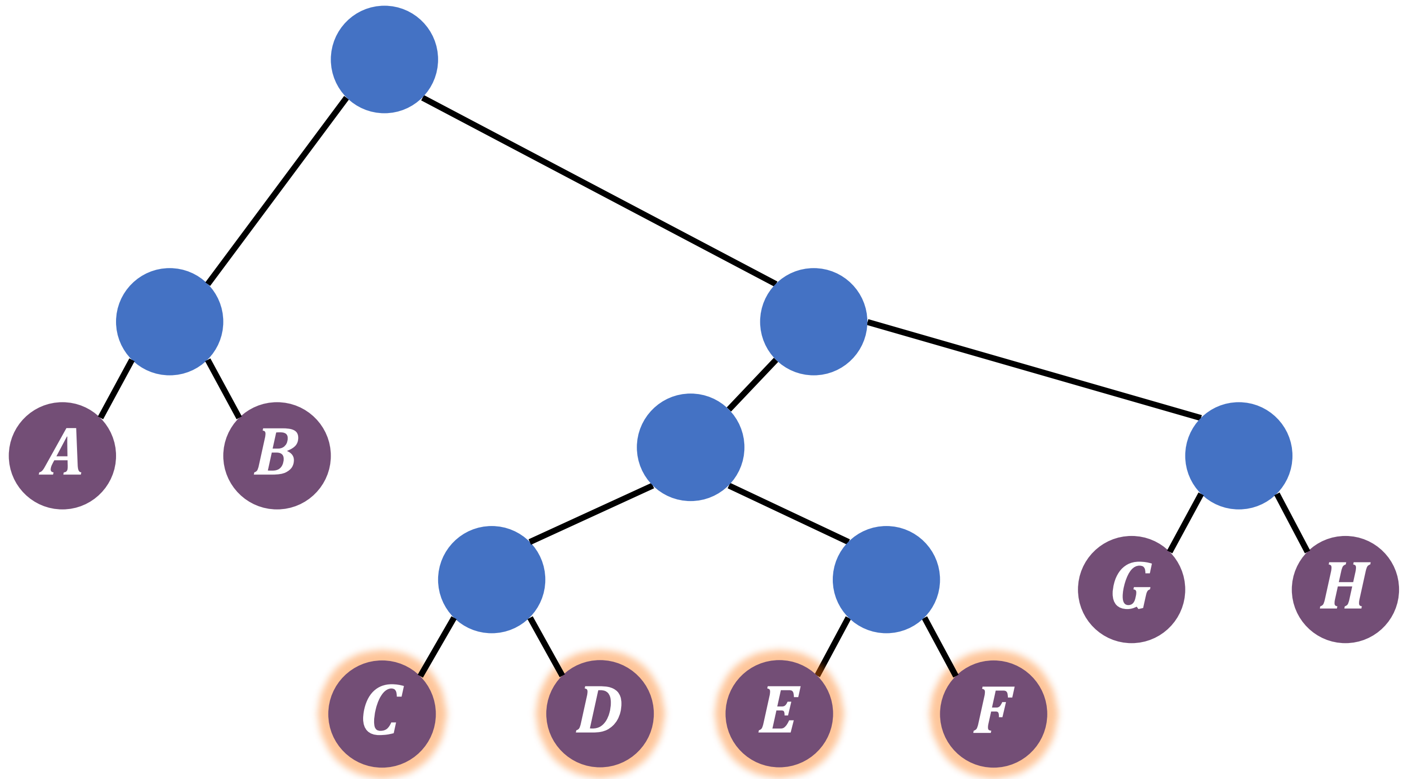
$$p_1 d_1 + p_2 d_2 > p_1 d_2 + p_2 d_1 \text{ since } (p_1 - p_2)(d_1 - d_2) > 0.$$

Optimal Labeling?

Consider leaves at maximum depth d in an optimal tree.

They are labeled with least frequent characters.

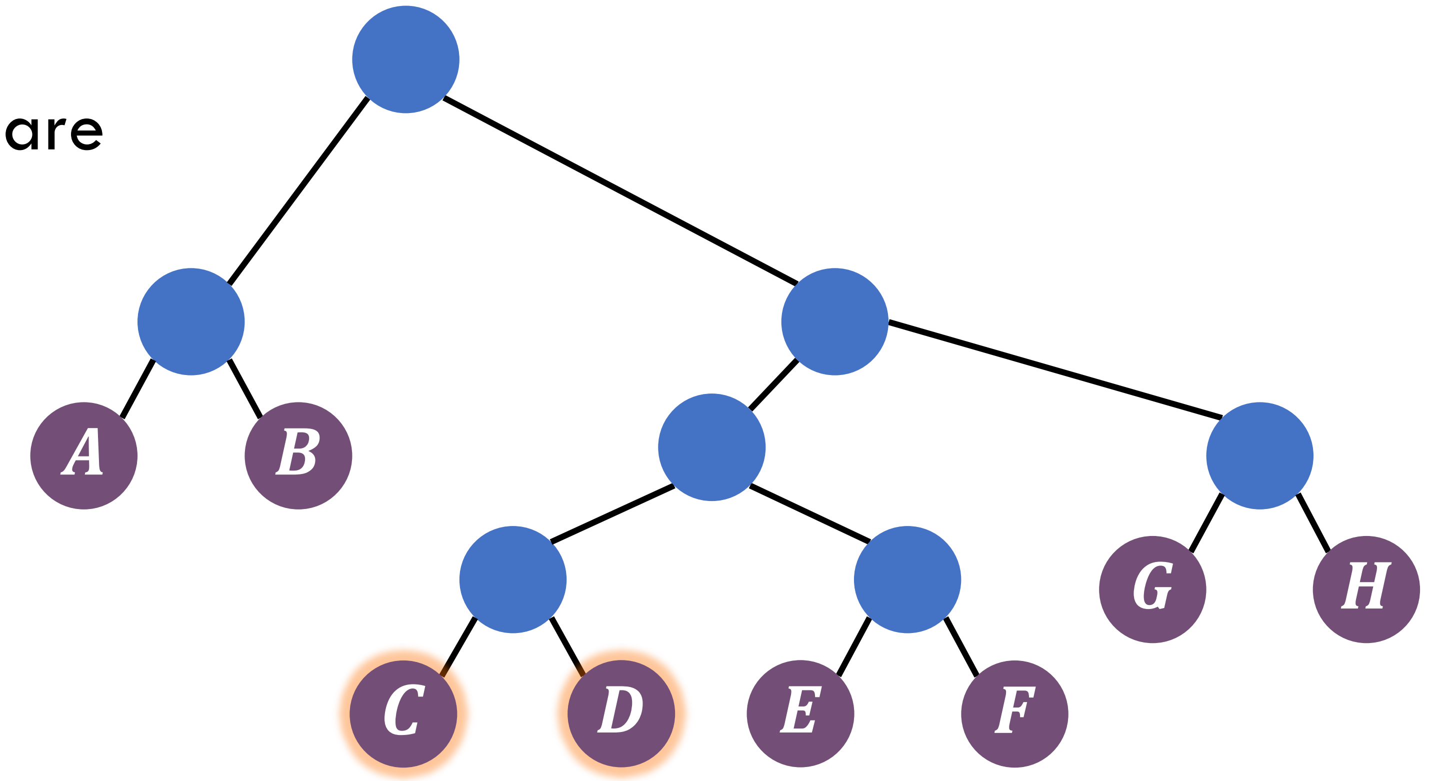
If we permute the labels of vertices of depth d, the cost will not change.



Optimal Labeling?

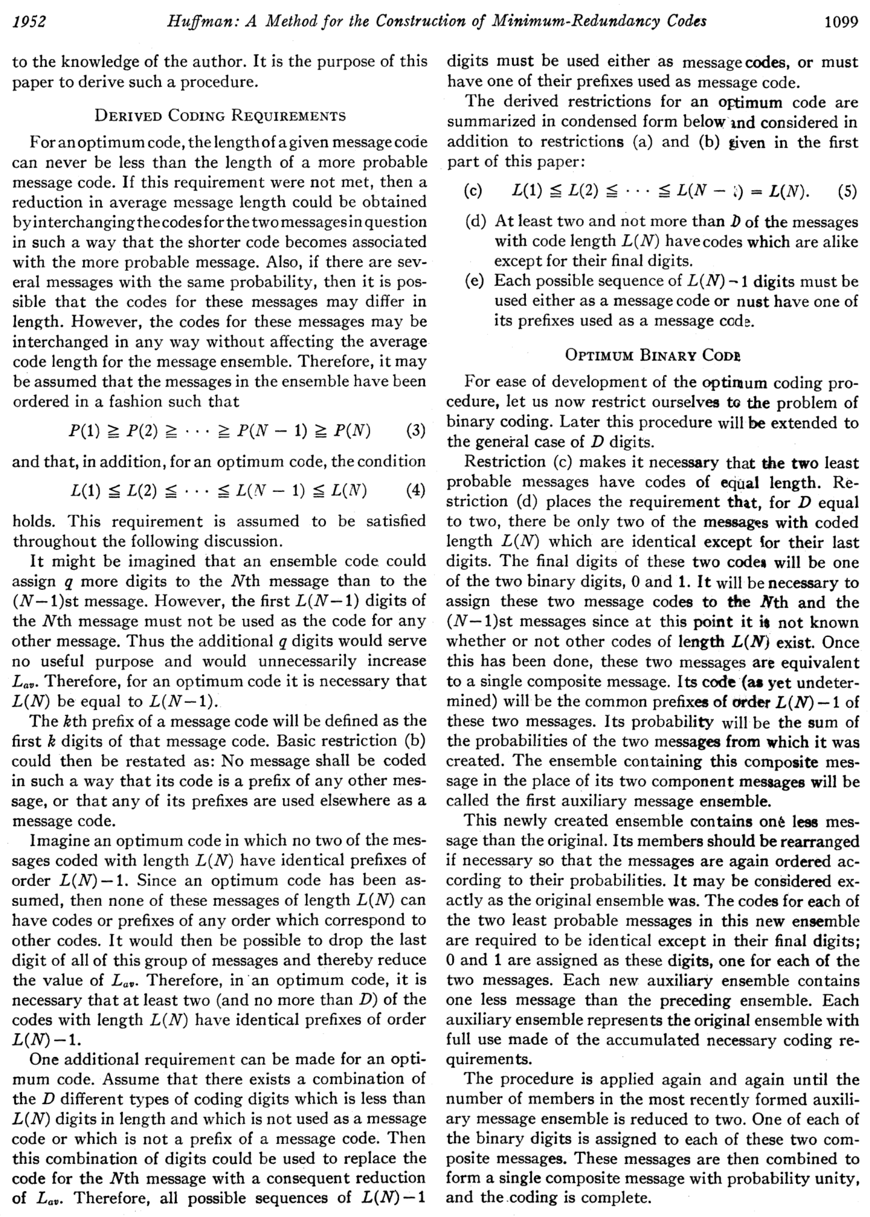
All leaves of maximum depth are split into pairs of siblings.

Two least frequent characters α and β are siblings in some optimal solution T^*.



Huffman Coding

- Huffman proposed a greedy algorithms for constructing optimal prefix codes in 1952.



to the knowledge of the author. It is the purpose of this paper to derive such a procedure.

Derived Coding Requirements

For an optimum code, the length of a given message code can never be less than the length of a more probable message code. If this requirement were not met, then a reduction in average message length could be obtained by interchanging the codes for the two messages in question in such a way that the shorter code becomes associated with the more probable message. Also, if there are several messages with the same probability, then it is possible that the codes for these messages may differ in length. However, the codes for these messages may be interchanged in any way without affecting the average code length for the message ensemble. Therefore, it may be assumed that the messages in the ensemble have been ordered in a fashion such that

$$P(1) \geqq P(2) \geqq \cdots \geqq P(N-1) \geqq P(N) \qquad (3)$$

and that, in addition, for an optimum code, the condition

$$L(1) \leqq L(2) \leqq \cdots \leqq L(N-1) \leqq L(N) \qquad (4)$$

holds. This requirement is assumed to be satisfied throughout the following discussion.

It might be imagined that an ensemble code could assign q more digits to the Nth message than to the $(N-1)$st message. However, the first $L(N-1)$ digits of the Nth message must not be used as the code for any other message. Thus the additional q digits would serve no useful purpose and would unnecessarily increase L_{av}. Therefore, for an optimum code it is necessary that $L(N)$ be equal to $L(N-1)$.

The kth prefix of a message code will be defined as the first k digits of that message code. Basic restriction (b) could then be restated as: No message shall be coded in such a way that its code is a prefix of any other message, or that any of its prefixes are used elsewhere as a message code.

Imagine an optimum code in which no two of the messages coded with length $L(N)$ have identical prefixes of order $L(N)-1$. Since an optimum code has been assumed, then none of these messages of length $L(N)$ can have codes or prefixes of any order which correspond to other codes. It would then be possible to drop the last digit of all of this group of messages and thereby reduce the value of L_{av}. Therefore, in an optimum code, it is necessary that at least two (and no more than D) of the codes with length $L(N)$ have identical prefixes of order $L(N)-1$.

One additional requirement can be made for an optimum code. Assume that there exists a combination of the D different types of coding digits which is less than $L(N)$ digits in length and which is not used as a message code or which is not a prefix of a message code. Then this combination of digits could be used to replace the code for the Nth message with a consequent reduction of L_{av}. Therefore, all possible sequences of $L(N)-1$ digits must be used either as message codes, or must have one of their prefixes used as message code.

The derived restrictions for an optimum code are summarized in condensed form below and considered in addition to restrictions (a) and (b) given in the first part of this paper:

(c) $L(1) \leqq L(2) \leqq \cdots \leqq L(N-1) = L(N)$. (5)

(d) At least two and not more than D of the messages with code length $L(N)$ have codes which are alike except for their final digits.

(e) Each possible sequence of $L(N)-1$ digits must be used either as a message code or must have one of its prefixes used as a message code.

Optimum Binary Code

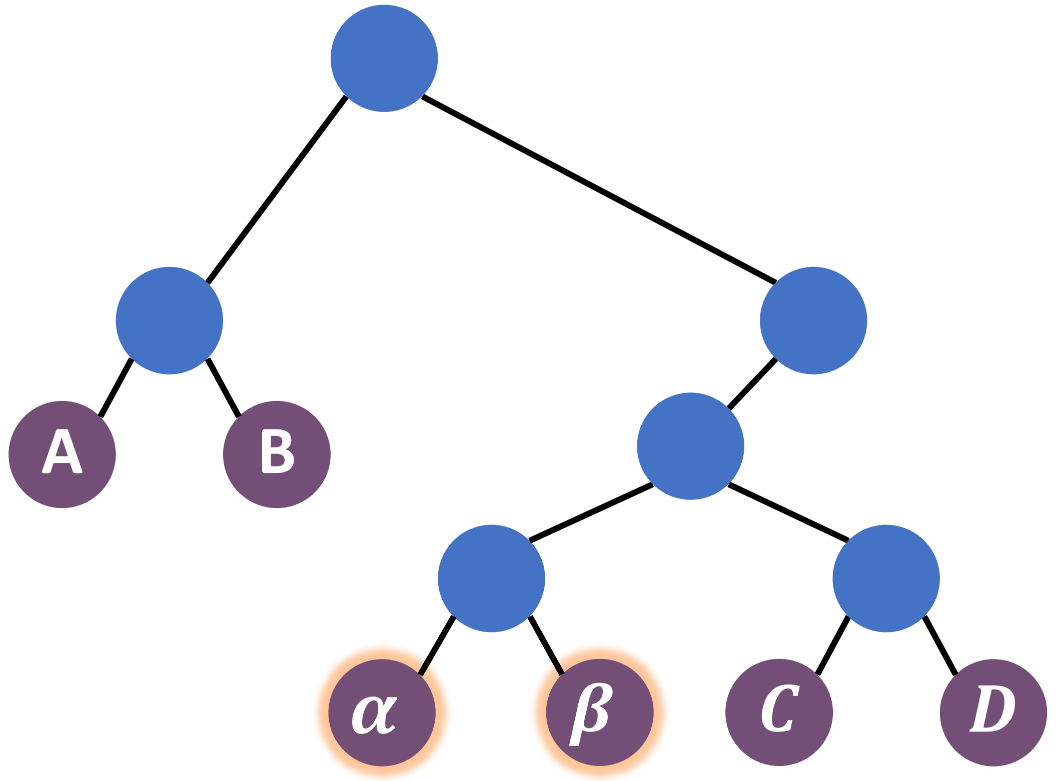
For ease of development of the optimum coding procedure, let us now restrict ourselves to the problem of binary coding. Later this procedure will be extended to the general case of D digits.

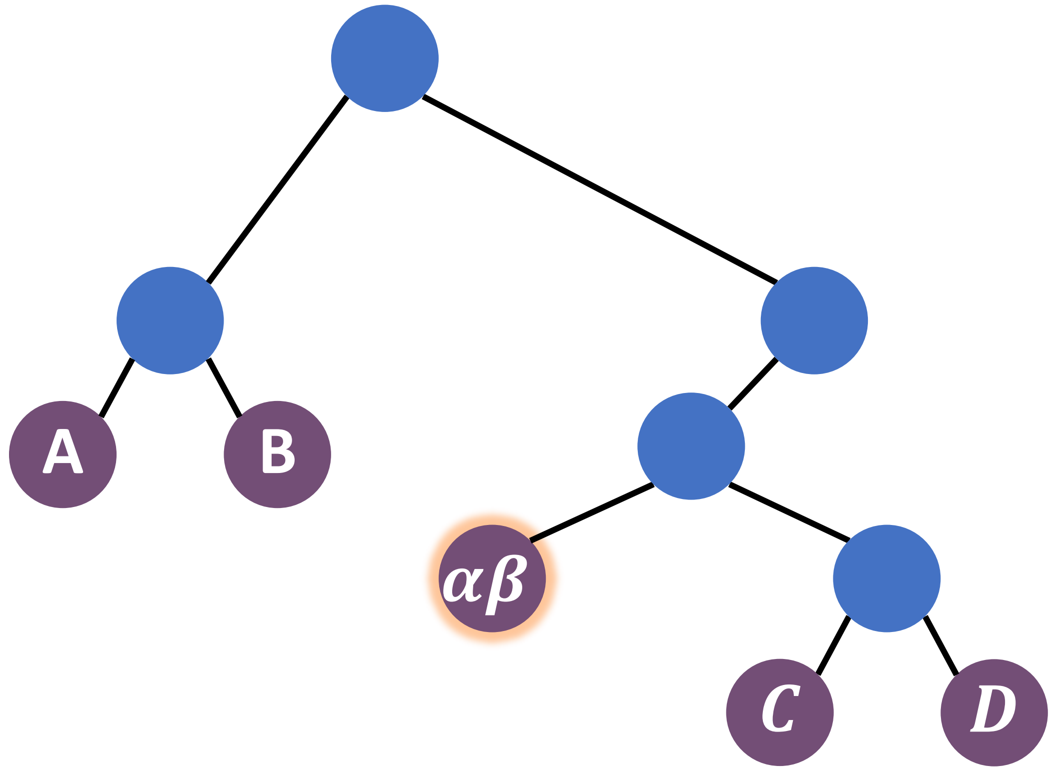
Restriction (c) makes it necessary that the two least probable messages have codes of equal length. Restriction (d) places the requirement that, for D equal to two, there be only two of the messages with coded length $L(N)$ which are identical except for their last digits. The final digits of these two codes will be one of the two binary digits, 0 and 1. It will be necessary to assign these two message codes to the Nth and the $(N-1)$st messages since at this point it is not known whether or not other codes of length $L(N)$ exist. Once this has been done, these two messages are equivalent to a single composite message. Its code (as yet undetermined) will be the common prefixes of order $L(N)-1$ of these two messages. Its probability will be the sum of the probabilities of the two messages from which it was created. The ensemble containing this composite message in the place of its two component messages will be called the first auxiliary message ensemble.

This newly created ensemble contains one less message than the original. Its members should be rearranged if necessary so that the messages are again ordered according to their probabilities. It may be considered exactly as the original ensemble was. The codes for each of the two least probable messages in this new ensemble are required to be identical except in their final digits; 0 and 1 are assigned as these digits, one for each of the two messages. Each new auxiliary ensemble contains one less message than the preceding ensemble. Each auxiliary ensemble represents the original ensemble with full use made of the accumulated necessary coding requirements.

The procedure is applied again and again until the number of members in the most recently formed auxiliary message ensemble is reduced to two. One of each of the binary digits is assigned to each of these two composite messages. These messages are then combined to form a single composite message with probability unity, and the coding is complete.

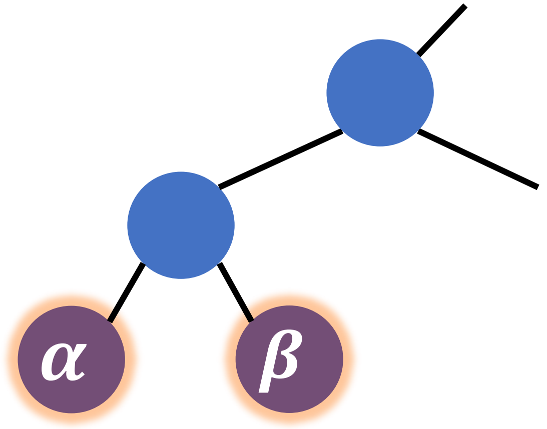
Thought Experiment

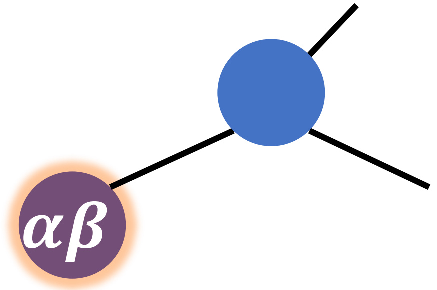




Let's replace two letters α and β with one new letter $(\alpha\beta)$: $\Sigma' = \Sigma \cup \{(\alpha\beta)\} \setminus \{\alpha, \beta\}$
with frequency $p_{\alpha\beta} = p_\alpha + p_\beta$.
One-to-one correspondence: a tree T for Σ $\Leftrightarrow$ a tree T' for Σ'.

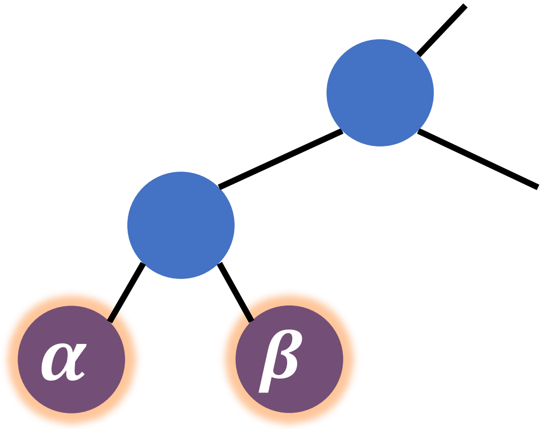
Thought Experiment

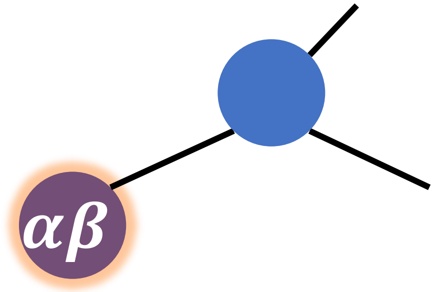




$$\begin{aligned} cost(T') &= cost(T) - p_\alpha \cdot d - p_\beta \cdot d + p_{\alpha\beta} \cdot (d-1) \\ &= cost(T) - p_\alpha \cdot d - p_\beta \cdot d + \left(p_\alpha + p_\beta\right) \cdot (d-1) \\ &= cost(T) - (p_\alpha + p_\beta) \end{aligned}$$

Thought Experiment





$$cost(T') = cost(T) - (p_\alpha + p_\beta)$$

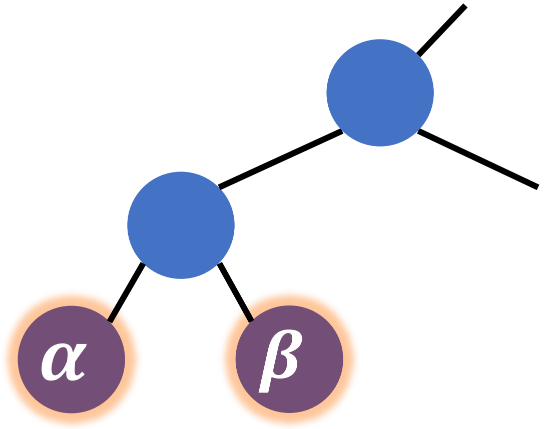
Start with an optimal tree T for Σ, construct T' for Σ':

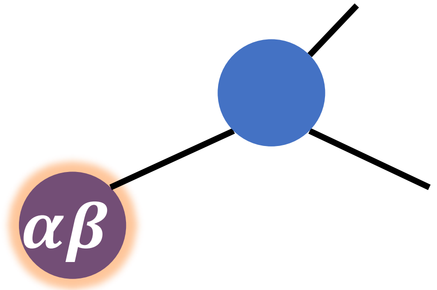
$$opt.cost(\Sigma) = cost(\mathrm{T}) = cost\ (T') + \left(p_\alpha + p_\beta\right) \geq opt.cost(\Sigma') + \left(p_\alpha + p_\beta\right)$$

Start with an optimal tree T' for Σ', construct T for Σ:

$$opt.cost(\Sigma') = cost(\mathrm{T}') = cost\ (T) - \left(p_\alpha + p_\beta\right) \geq opt.cost(\Sigma) - \left(p_\alpha + p_\beta\right)$$

Thought Experiment





$$cost(T') = cost(T) - (p_\alpha + p_\beta)$$

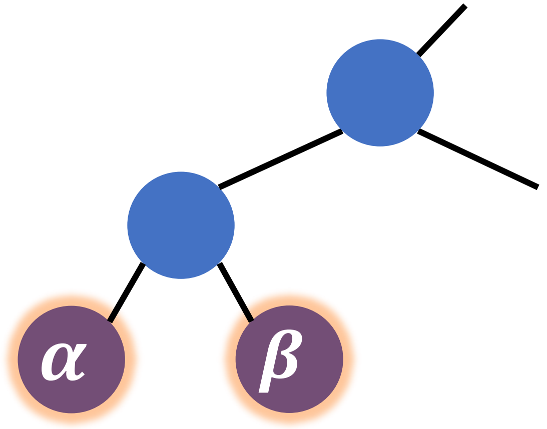
Start with an optimal tree T for Σ, construct T' for Σ':

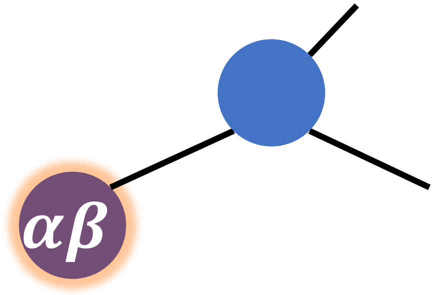
$$opt.cost(\Sigma) = cost(\mathrm{T}) = cost\ (T') + \left(p_\alpha + p_\beta\right) \geq opt.cost(\Sigma') + \left(p_\alpha + p_\beta\right)$$

Start with an optimal tree T' for Σ', construct T for Σ:

$$opt.cost(\Sigma') = cost(\mathrm{T}') = cost\ (T) - \left(p_\alpha + p_\beta\right) \geq opt.cost(\Sigma) - \left(p_\alpha + p_\beta\right)$$

Thought Experiment





$$cost(T') = cost(T) - (p_\alpha + p_\beta)$$

$$opt.cost(\Sigma) = opt.cost(\Sigma') + \left(p_\alpha + p_\beta\right)$$

T is an optimal for Σ **if and only if** T' is optimal for Σ'

Huffman's Algorithm (recursive version)

```
function ConstructTree($\Sigma$ with frequencies $\{p_i\}$)
    If $|\Sigma| = 2$
        return the only full binary tree with two leaves
    else
        find two least frequent characters $\alpha$ and $\beta$ in $\Sigma$
        let $\Sigma' = \Sigma \cup \{(\alpha\beta)\} \setminus \{\alpha, \beta\}$
        let $p_{\alpha\beta} = p_\alpha + p_\beta$
        $T' =$ ConstructTree($\Sigma'$)
        add two children $\alpha$ and $\beta$ to vertex $(\alpha\beta)$
        return the obtained tree $T$
```

Huffman's Algorithm (recursive version)

```
function ConstructTree($\Sigma$ with frequencies $\{p_i\}$)
    If $|\Sigma| = 2$
        return the only full binary tree with two leaves
    else
        find two least frequent characters $\alpha$ and $\beta$ in $\Sigma$
        let $\Sigma' = \Sigma \cup \{(\alpha\beta)\} \setminus \{\alpha, \beta\}$
        let $p_{\alpha\beta} = p_\alpha + p_\beta$
        $T' =$ ConstructTree($\Sigma'$)
        add two children $\alpha$ and $\beta$ to vertex $(\alpha\beta)$
        return the obtained tree $T$
```

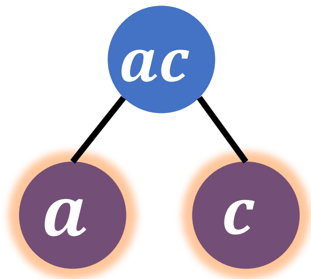
Huffman’s Algorithm

$\boldsymbol{\sigma}$	$\boldsymbol{p}$
a	0.1
b	0.2
c	0.15
d	0.3
e	0.25

Huffman's Algorithm

σ	p
a	0.1
b	0.2
c	0.15
d	0.3
e	0.25

Find two characters with least frequencies: a and c.



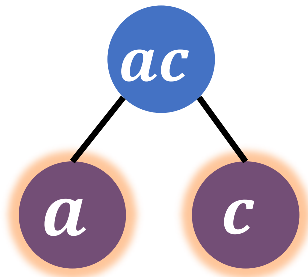
Huffman's Algorithm

Find two characters with least frequencies: a and c.

Add a new character ac.

Create a tree with root ac and leaves a and c.

Update the table of frequencies.

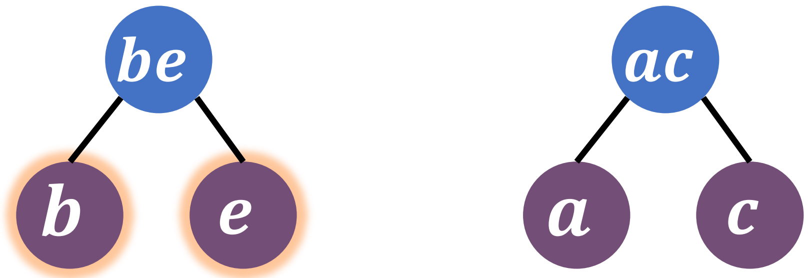


σ	p
~~a~~	~~0.1~~
b	0.2
~~c~~	~~0.15~~
d	0.3
e	0.25
(ac)	0.25

Huffman's Algorithm

σ	p
b	0.2
d	0.3
e	0.25
(ac)	0.25

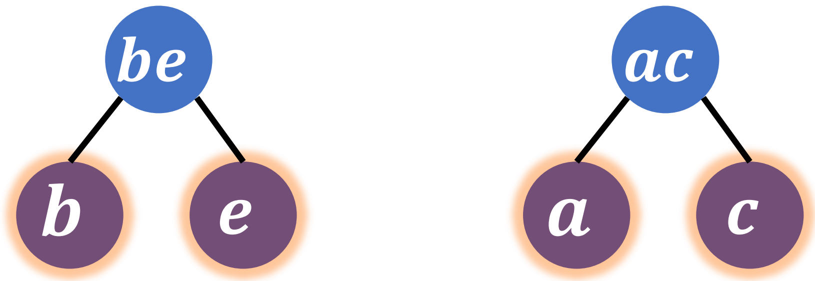
Find two least frequent characters in the new table: b and either e or (ac). Let us say we choose b and e.



Huffman's Algorithm

σ	p
~~b~~	~~0.2~~
d	0.3
~~e~~	~~0.25~~
(ac)	0.25
(be)	0.45

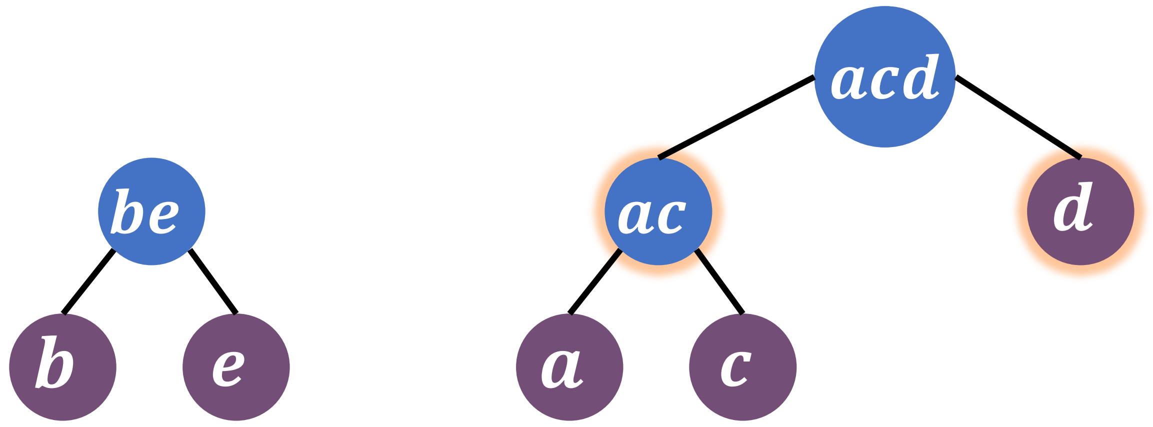
Find two least frequent characters in the new table: b and either e or (ac). Let us say we choose b and e.



Huffman's Algorithm

σ	p
d	0.3
(ac)	0.25
(be)	0.45

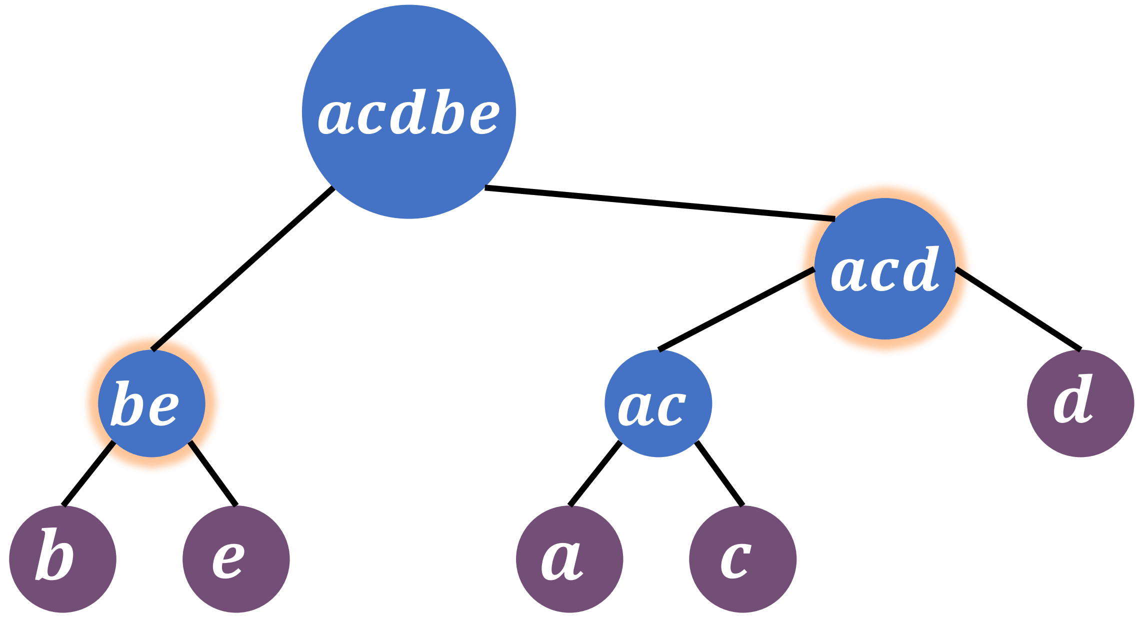
Again, find two least frequent characters: (ac) and d



Huffman's Algorithm

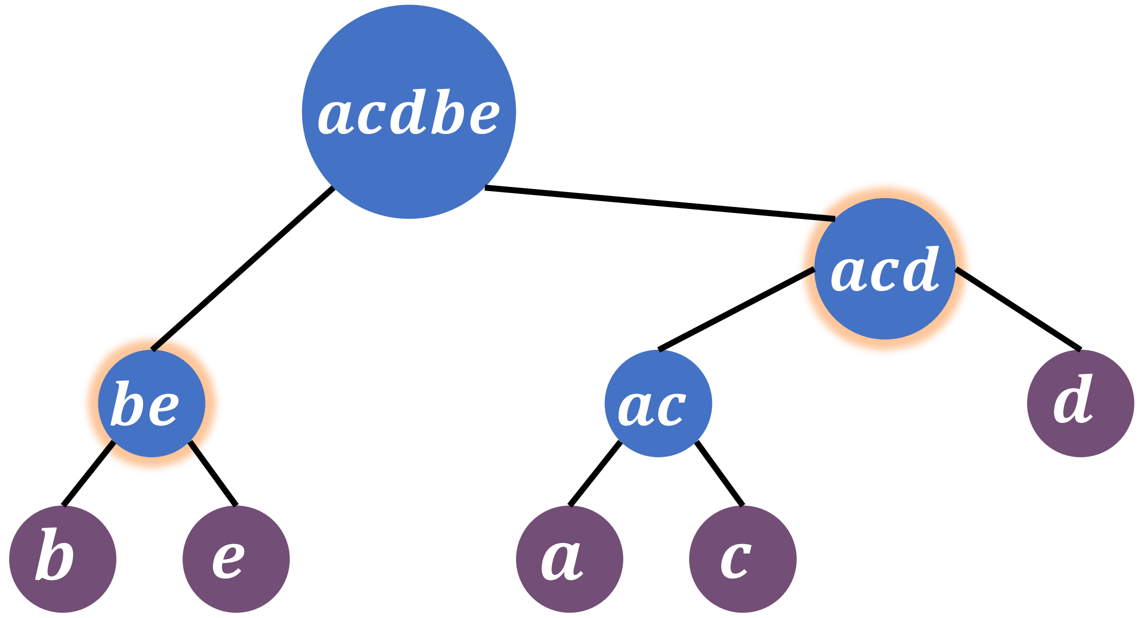
σ	p
(ac)	0.55
(be)	0.45

We are left with two characters.



Huffman's Algorithm

σ	code
a	100
b	00
c	101
d	11
e	01



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

Implementation

- Create a vertex v_σ for every $\sigma \in \Sigma$
- Put all vertices in a priority queue Q.
- The weight/priority of v_σ is the frequency of σ.
- while Q has 2 elements or more do
  - Extract two vertices u and v with smallest weights
  - Create a new vertex w with $p_w = p_u + p_v$
  - Make u and v children of w.
  - Put w in Q.
- Extract the last element r from Q.
- Return the tree rooted at r.

Shannon Encoding

- Can we do better than Huffman encoding?
  - No, if we encode messages character-by-character.
  - Yes, if we may encode multiple characters at once.

- Asymptotically optical encoding rate is

$$H(p_1, \ldots, p_n) = \sum_i p_i \log_2 1/p_i$$

bits per character. Function $H(\cdots)$ is called the Shannon entropy function.

Dynamic Programming

Puzzle

- We are given n digits: $a_1 a_2 \dots a_n$.
- Need to group the digits into one-digit and two-digit numbers so that their sum is as large as possible.

Example:

$$19118237281$$

Exponentially many feasible solutions:

$$19 + 11 + 8 + 2 + 37 + 28 + 1,\ \ 19 + 1 + 18 + 23 + 72 + 81, \dots$$

Optimal solution:

$$1\ +\ 91\ \ +\ 1\ +\ 82\ \ +\ 3\ +\ 72\ +\ 81 = 331$$

Puzzle

- Greedy doesn’t work.
- A brute force algorithm runs in exponential time.
- Need a new approach.

Thought Experiment

Consider the optimal solution.

There are two options: in this solution,

- the last digit is a separate one-digit number
- the last two digits are grouped together

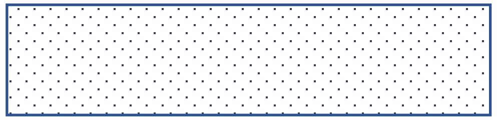
$$191182033781$$

$\boxed{\qquad\qquad\qquad} + 1$ $\boxed{\qquad\qquad\qquad} + 81$

191182033781

Case 1

Assume that we know that the first case holds.

 $+ 1$

Can we find the solution?

- Solve the problem for digits $a_1, \dots, a_{n-1}$ and add a_n.

Case 1

Assume that we know that the first case holds.

$\boxed{f(n-1)} + 1$

Can we find the solution?

- Solve the problem for digits $a_1, \dots, a_{n-1}$ and add a_n.

Let $f(i)$ be the optimal solution for digits $a_1, \dots, a_i$. Then

$$f(n) = f(n-1) + a_n$$

191182033781

$\boxed{f(n-1)} + 1$

Case 1: Claim

- We can always find a solution of value

$$f(n-1) + a_n$$

Thus,

$$f(n) \geq f(n-1) + a_n$$

- If we are in case 1,

$$f(n-1) \geq f(n) - a_n$$

Proof: …

191182033781

$\boxed{f(n-1)} + 1$

Case 1: Proof of the claim

➢ If we are in case 1,

$$f(n-1) \geq f(n) - a_n$$

Proof:

- Consider optimal solution for n of value $f(n)$.
- We are given that the last group consists of single digit a_n.
- Remove it.
- We get a feasible solution for $n-1$ of value $f(n) - a_n$. Thus

$$f(n-1) \geq f(n) - a_n$$

as required.

191182033781

$\boxed{f(n-1)} + 1$

Case 1: Claim

Conclusion:

$f(n) = f(n-1) + a_n$ if we are in case 1

$f(n) \geq f(n-1) + a_n$ always

$$191182033781$$

$$\boxed{f(n-2)} + 81$$

Case 2

Assume that we know that the second case holds.

$$\boxed{f(n-2)} + 81$$

Can we find the solution?

- Solve the problem for digits $a_1, \dots, a_{n-2}$ and add number $\overline{a_{n-1}a_n}$.

$f(n) = f(n-2) + \overline{a_{n-1}a_n}$ if we are in case 2

$f(n) \geq f(n-2) + \overline{a_{n-1}a_n}$ always

here $\overline{a_{n-1}a_n} = 10\, a_{n-1} + a_n$

Cases 1 & 2

We proved that

$$f(n) = \begin{cases} f(n-1) + a_n & \text{if we are in case 1} \\ f(n-2) + \overline{a_{n-1}a_n} & \text{if we are in case 2} \end{cases}$$

Plan:

- Use this formula for $f(n)$
- Recursively compute $f(n-1)$ and $f(n-2)$
- Any obstacles?

Cases 1 & 2

We proved that

$$f(n) = \begin{cases} f(n-1) + a_n & \text{if we are in case 1} \\ f(n-2) + \overline{a_{n-1}a_n} & \text{if we are in case 2} \end{cases}$$

Our algorithm doesn’t know whether case 1 or 2 holds!

Cases 1 & 2

We proved that

$$f(n) = \begin{cases} f(n-1) + a_n & \text{if we are in case 1} \\ f(n-2) + \overline{a_{n-1}a_n} & \text{if we are in case 2} \end{cases}$$

… but we know that

$$\left.\begin{aligned} f(n) &\geq f(n-1) + a_n \\ f(n) &\geq f(n-2) + \overline{a_{n-1}a_n} \end{aligned}\right\} \Rightarrow f(n) \geq \max(f(n-1) + a_n, f(n-2) + \overline{a_{n-1}a_n})$$

Cases 1 & 2

- Proved

$$f(n) = \max(f(n-1) + a_n, f(n-2) + \overline{a_{n-1}a_n})$$

Recurrence

- Proved a recurrence formula

$$f(i) = \max(f(i-1) + a_i, f(i-2) + \overline{a_{i-1}a_i})$$

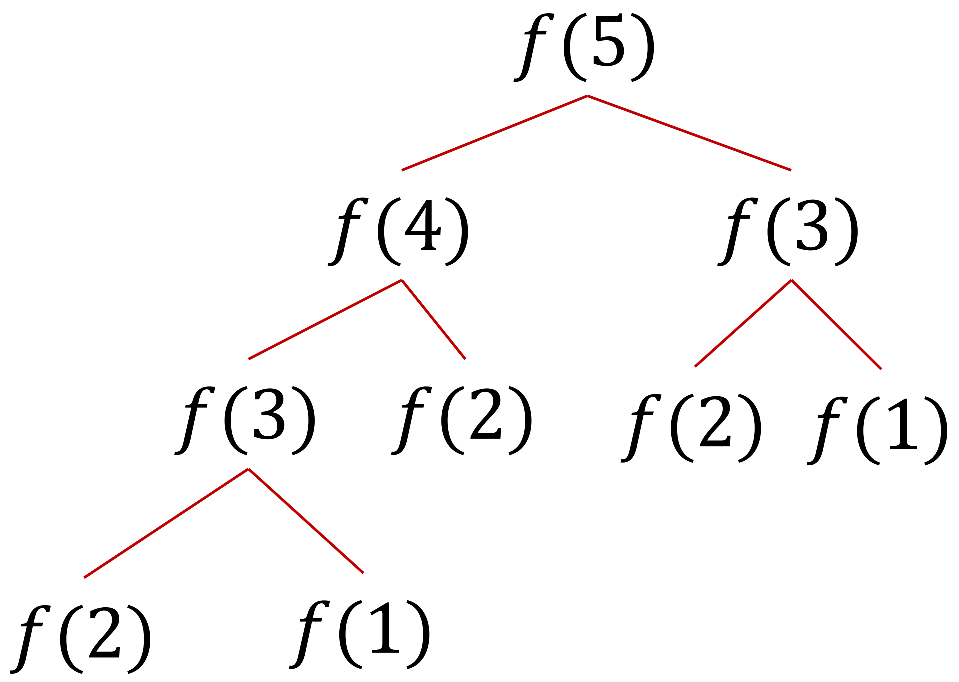
- Can we compute $f(n)$ recursively?

```
function $f(i)$
    if $i = 1$, return $a_1$
    if $i = 2$, return $\overline{a_1 a_2}$
    if $i > 2$, return $\max(f(i-1) + a_i, f(i-2) + \overline{a_{i-1}a_i})$
```

- This solution works correctly but it is very slow.

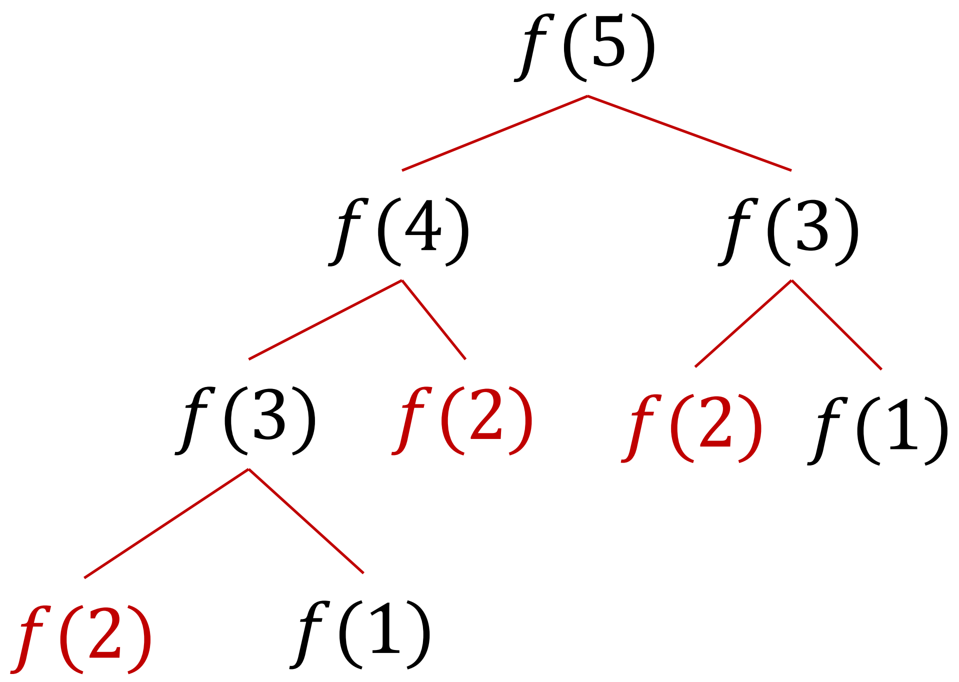
Recursion

To compute $f(5)$ we make 9 recursive calls



i	#calls
1	1
2	1
3	3
4	5
5	9
6	15
7	25
8	41
9	67
10	109
11	177
grows exponentially	

Recursion



To compute $f(5)$ we make 9 recursive calls

- compute $f(3)$ twice times
- compute $f(2)$ three times
- compute $f(1)$ twice times

… many more times when we compute $f(i)$ for large i

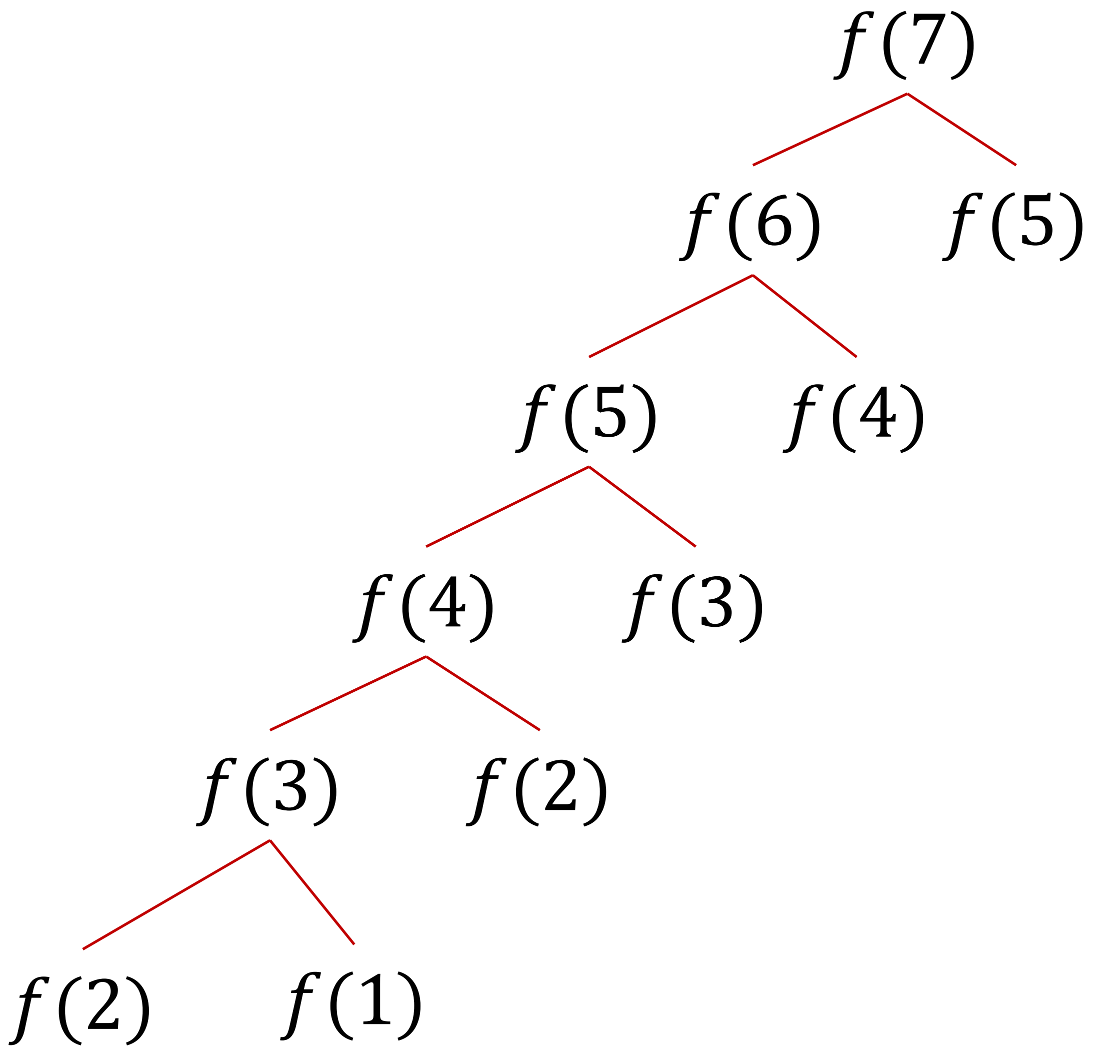
This is wasteful!

Memoization

- Store computed values of $f(i)$ in a table. Don't recompute them!

```
create a table $T[1:n]$; mark all entries as "non-initialized"
function $f(i)$
    if $T[i]$ is initialized, return $T[i]$
    if $i = 1$, result $= a_1$
    else if $i = 2$, result $= \overline{a_1 a_2}$
    else if $i > 2$, result $= \max(f(i-1) + a_i, f(i-2) + \overline{a_{i-1} a_i})$
    $T[i] = \text{result}$
    return result
```

Memoization



i	#calls now	#calls prev.
1	1	1
2	1	1
3	3	3
4	5	5
5	7	9
6	9	15
7	11	25
8	13	41
9	15	67
10	17	109
11	19	177
linear vs exponential growth		

To be continued

Unrolling the Recursion

```
create table $T[1{:}n]$
$T[1] = a_1$
$T[2] = 10 * a_1 + a_2$
for $i = 3$ to $n$ do
    $T[i] = \max(T(i-1) + a_i, T(i-2) + \overline{a_{i-1}a_i})$
return $T[n]$
```

Running time is $O(n)$.