Module 3
Greedy Strategy

Dr. Natarajan Meghanathan
Associate Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Introduction to Greedy Technique

- Main Idea: In each step, choose the best alternative available in the hope that a sequence of locally optimal choices will yield a (globally) optimal solution to the entire problem.
- Example 1: Decimal to binary representation (objective: minimal number of 1s in the binary representation): Technique – Choose the largest exponent of 2 that is less than or equal to the unaccounted portion of the decimal integer.

	64	**32**	**16**	**8**	**4**	**2**	**1**
To represent 75:	1	0	0	1	0	1	1

- Example 2: Coin Denomination in US – Quarter (25 cents), Dime (10 cents), Nickel (5 cents) and Penny (1 cent).
- Objective: Find the minimum number of coins for a change
- Strategy: Choose the coin with the largest denomination that is less than or equal to the unaccounted portion of the change.
- For example, to find a change for 48, we would choose 1 quarter, 2 dimes and 3 pennies. The optimal solution is thus 6 coins and there cannot be anything less than 6 coins for US coin denominations.

Greedy Technique: Be careful!!!

- Greedy technique (though may appear to be computationally simple) cannot always guarantee to yield the optimal solution. It may end up only as an approximate solution to an optimization problem.
- For example, consider a more generic coin denomination scenario where the coins are valued 25, 10 and 1. To make a change for 30, we would end up using 6 coins (1 coin of value 25 and 5 coins of value 1 each) following the greedy technique. On the other hand, if we had used a dynamic programming algorithm for this generic version, we would have end up with 3 coins, each of value 10.

Fractional Knapsack Problem (Greedy Algorithm): Example 1

Knapsack weight is 6lb.

Item	1	2	3	4	5
Value, $	25	20	15	40	50
Weight, lb	3	2	1	4	5
Value/Weight	8.3	10	15	10	10

- **Greedy Strategy:** Pick the items in the decreasing order of the Value/Weight.
- Break the tie among the items the same Value/Weight by picking the item with the lowest Item index
- An optimal solution would be:
- Item 3 (1 lb), Item 2 (2 lb), and 3 lbs of Item 4.
- The maximum total Value of the items would be: $65
- Item 3 ($15), Item 2 ($20) and Item 4((3/4)*40 = $30)

- **Dynamic Programming:** If the items cannot be divided, and we have to pick only either the full item or just leave it, then the problem is referred to as an Integer (a.k.a. 0-1) Knapsack problem, and we will look at it in the module on Dynamic Programming.

Fractional Knapsack Problem (Greedy Algorithm): Example 2

Knapsack weight = 5 lb.

Item	1	2	3	4
Value, $	12	10	20	15
Weight, lb	2	1	3	2

Solution: Compute the Value/Weight for each item

Item	1	2	3	4
Value/Weight	6	10	6.67	7.5

Re-ordering the items according to the decreasing order of Value/Weight (break the tie by picking the item with the lowest Index)

Item	2	4	3	1
Value/Weight	10	7.5	6.67	6
Value, $	10	15	20	12
Weight, lb	1	2	3	2
Weight collected	1	2	2	

Items collected: Item 2 (1 lb, $10); Item 4 (2 lb, $15); Item 3 (2 lb, (2/3)*20 = $13.3);
Total Value = $38.3

Variable Length Prefix Encoding

- **Encoding Problem:** We want to encode a text that comprises of symbols from some n-symbol alphabet by assigning each symbol a sequence of bits called the codeword.
- If we assign bit sequences of the same length to each symbol, it is referred to as *fixed-length encoding*, we would need $\log_2 n$ bits per symbol of the alphabet and this is also the average # bits per symbol.
  - The 8-bit ASCII code assigns each of the 256 symbols a unique 8-bit binary code (whose integer values range from 0 to 255).
  - However, note that not all of these 256 symbols appear with the same frequency.
- **Motivation for Variable Code Assignment:** If we can come up with a code assignment such that symbols are assigned a bit sequence that is inversely related to the frequency of their occurrence (i.e., symbols that occur more frequently are given a shorter bit sequence and symbols that occur less frequently are given a longer bit sequence), then we could reduce the average number of bits per symbol.

- **Motivation for Prefix-free Code:** However, care should be taken such that if a given sequence of bits encoding a text is scanned (say from left to right), we should be able to clearly decode each symbol. In other words, we should be able to tell how many bits of an encoded text represent the i^{th} symbol in the text?

Huffman Codes: Prefix-free Coding

- **Prefix-free Code:** In a prefix-free code, no codeword is a prefix of a code of another symbol. With a prefix-free code based encoding, we can simply scan a bit string until we get the first group of bits that is a codeword for some symbol, replace these bits by this symbol, and repeat this operation until the bit string's end is reached.
- **Huffman Coding:**
  - Associate the alphabet's symbols with leaves of a binary tree in which all the left edges are labeled by 0 and all the right edges are labeled by 1.
  - The codeword of a symbol can be obtained by recording the labels on the simple path (a path without any cycle) from the root to the symbol's leaf.
- **Proof of correctness:** The binary codes are assigned based on a simple path traversed from the root to a leaf node representing the symbol. Since there cannot be a simple path from the root to a leaf node that leads to another leaf node (then we have to trace back some intermediate node – meaning a cycle). Hence, Huffman codes are prefix codes.

Huffman Algorithm

- **Assumptions:** The frequencies of symbol occurrence are independent and are known in advance.
- **Optimality:** Given the above assumption, Huffman's encoding yields a minimum-length encoding (i.e., the average number of bits per symbol is the minimum). This property of Huffman's encoding has lead to its use one of the most important file-compression methods.
  - Symbols that occur at a high-frequency have a smaller number of bits in the binary code, compared to symbols that occur at a low-frequency.

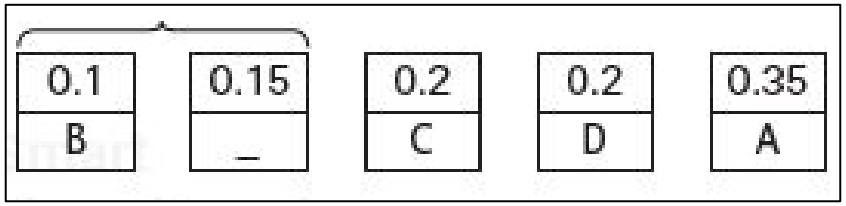
- **Step 1:** Initialize *n* one-node trees (one node for each symbol) and label them with the symbols of the given alphabet. Record the frequency of each symbol in its tree's root to indicate the tree's weight.
- **Step 2:** Repeat the following operation until a single tree is obtained:
  - Find two trees with the smallest weight (ties can be broken arbitrarily).
  - Make them the left and right sub trees of a new tree and record the sum of their weights in the root of the new tree as its weight.

Huffman Algorithm and Coding: Example

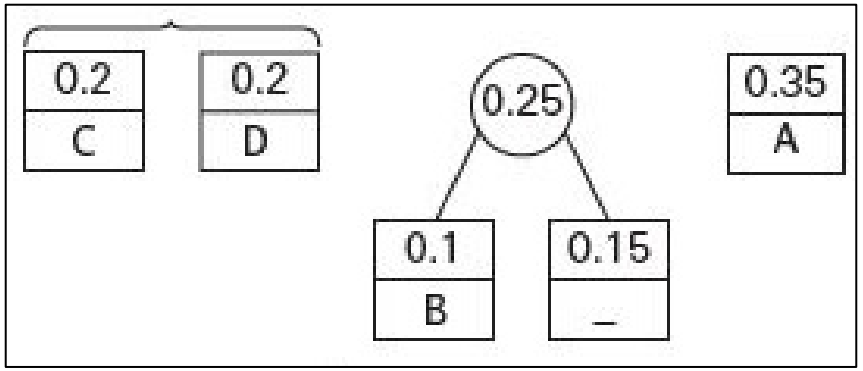
- Consider the five-symbol alphabet {A, B, C, D, -} with the following occurrence frequencies in a text made up of these symbols.
  - Construct a Huffman tree for this alphabet.
  - Determine the average number of bits per symbol.
  - Determine the compression ratio achieved compared to fixed-length encoding.

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

Initial

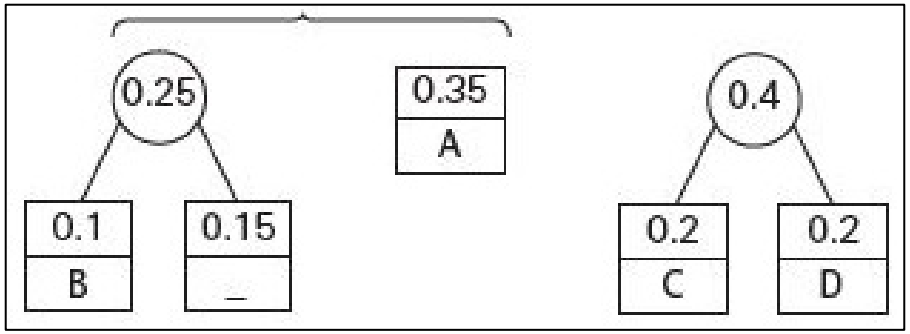


Iteration - 1

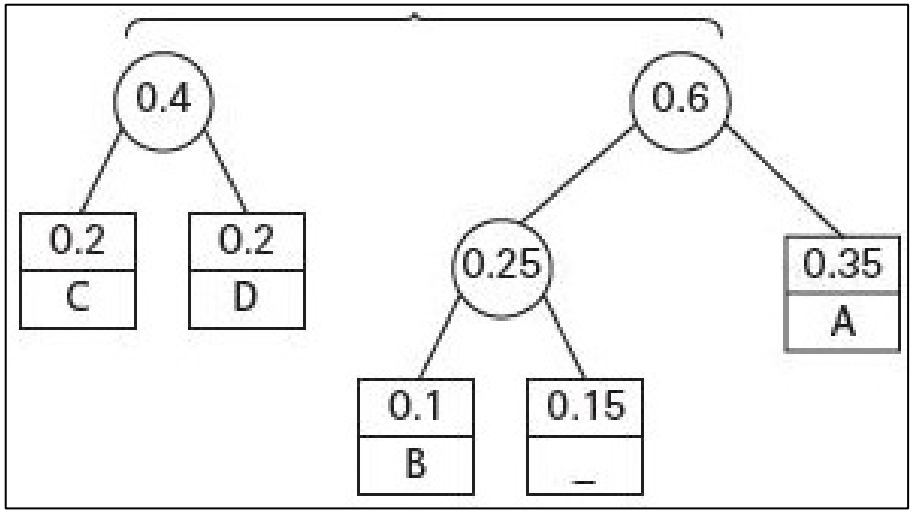


Huffman Algorithm and Coding: Example

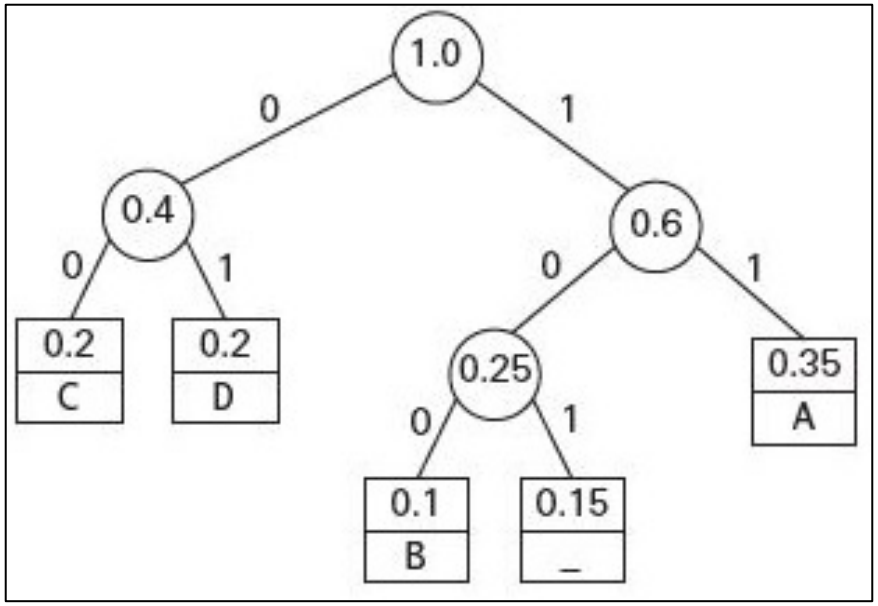
Iteration - 2



Iteration - 3



Iteration – 4 (Final)



symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Avg. # bits per symbol

= 2*0.35 + 3*0.1 + 2*0.2 + 2*0.2 + 3*0.15

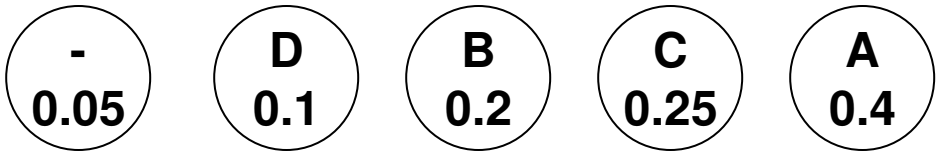
= 2.25 bits per symbol.

A fixed-length encoding of 5 symbols would require $\lceil \log_2 5 \rceil$ = 3 symbols. Hence, the **compression ratio** is 1 – (2.25/3) = 25%.

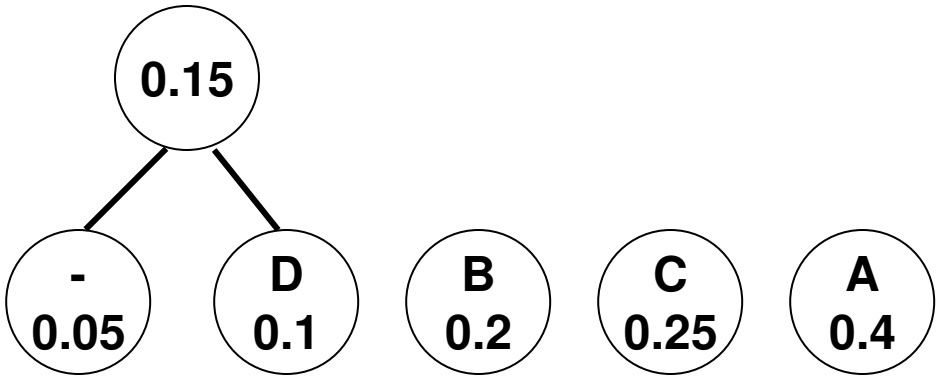
Huffman Coding: Example 2

A	0.4
B	0.2
C	0.25
D	0.1
-	0.05

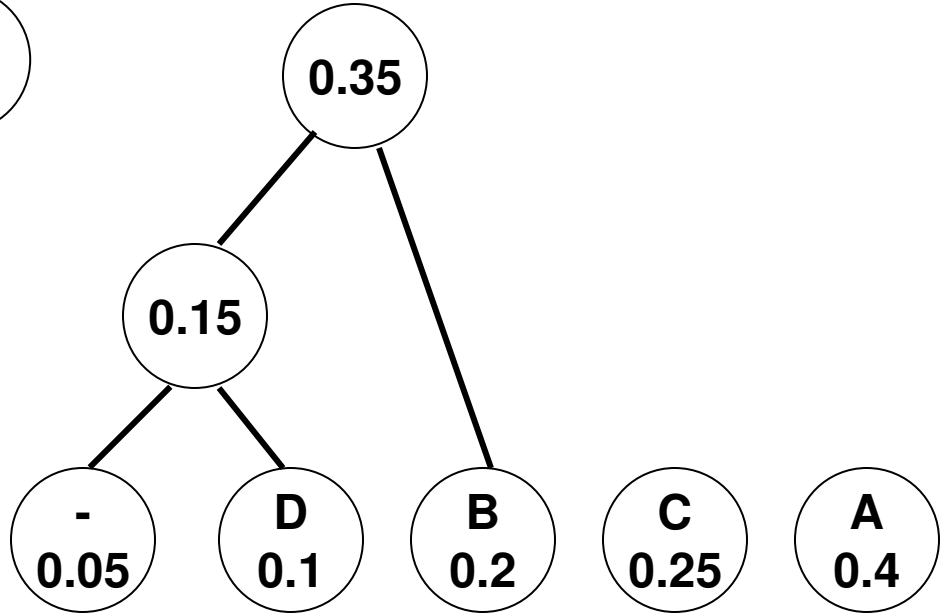
Initial



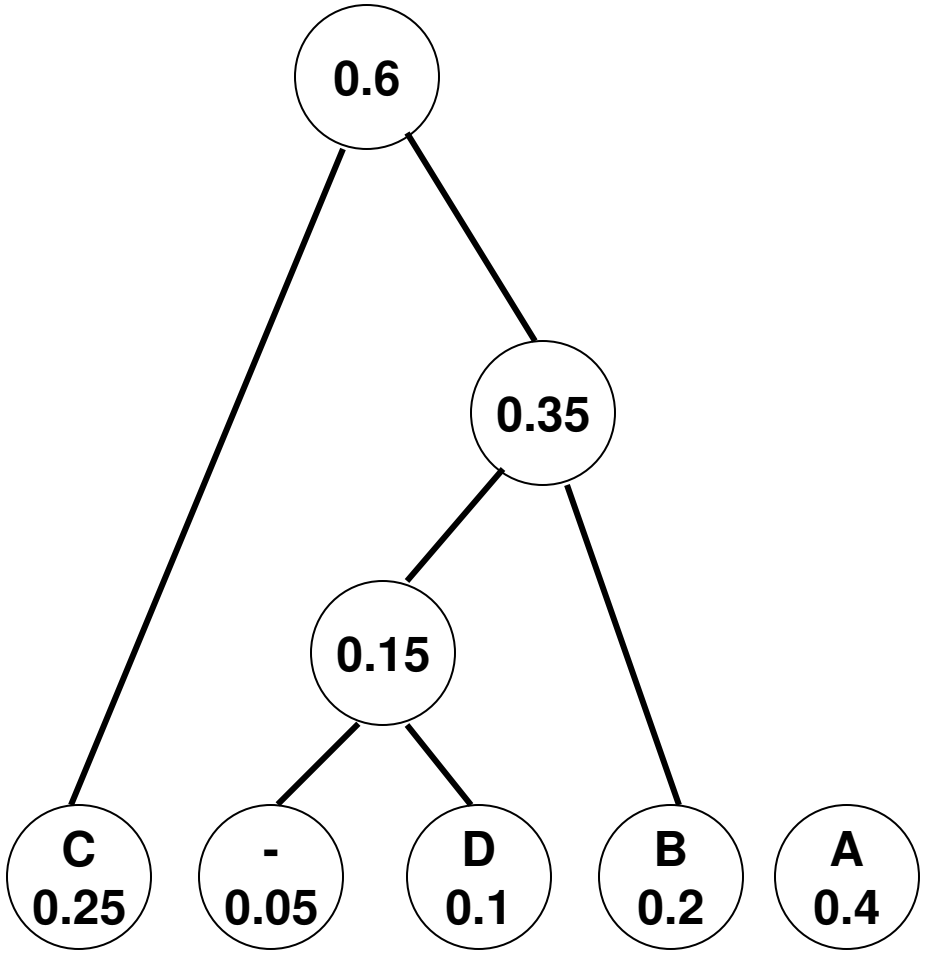
Iteration 1



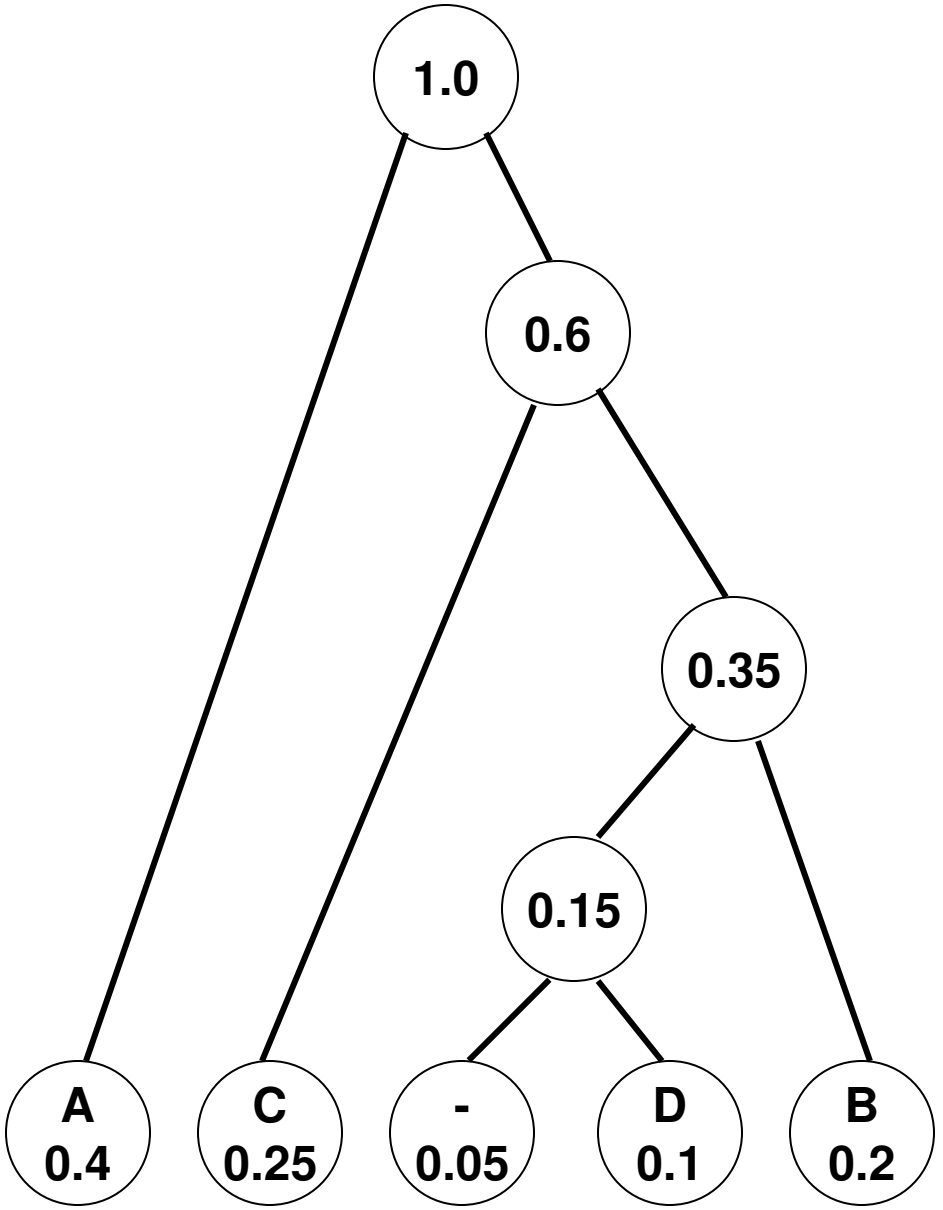
Iteration 2



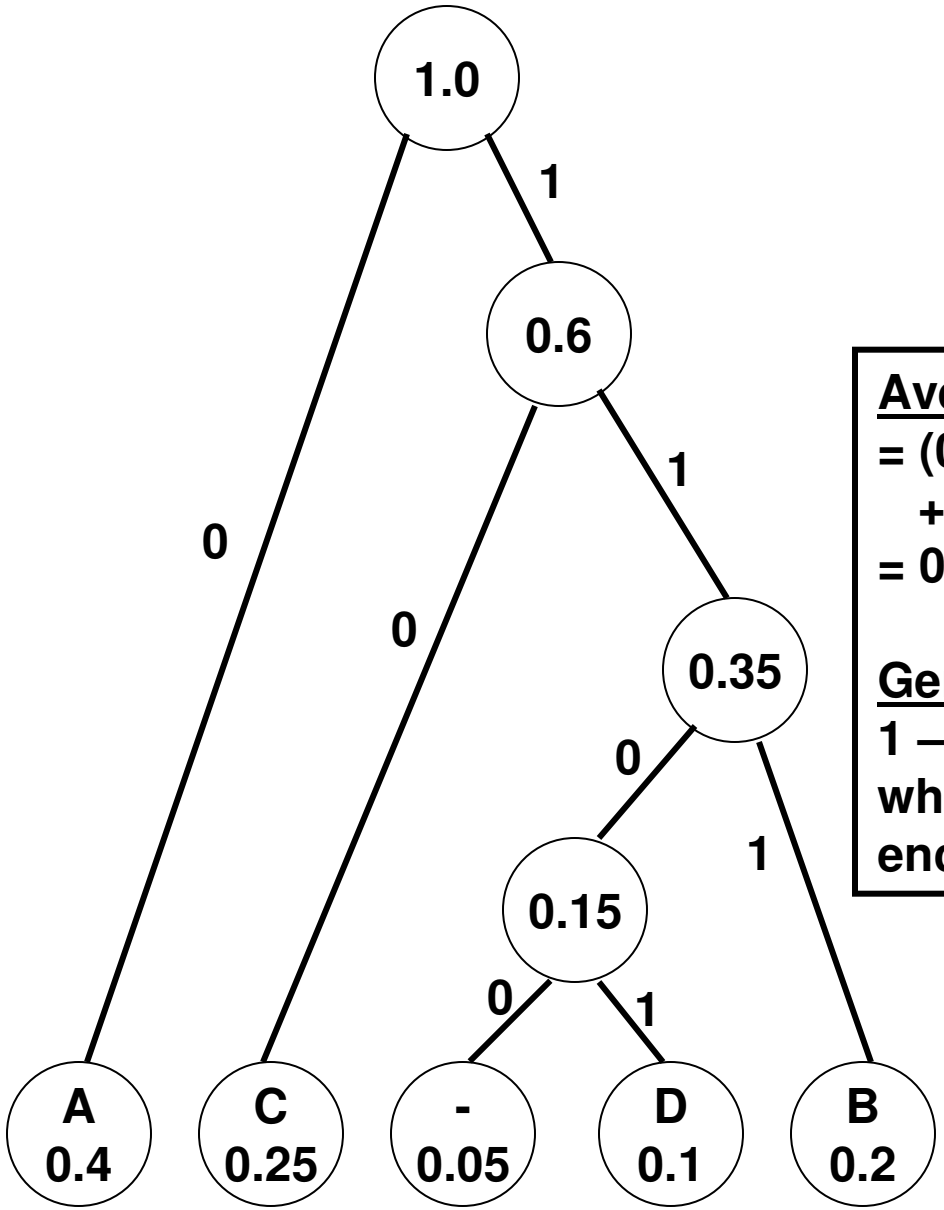
Iteration 3



Iteration 4



Huffman Tree



Huffman Codes

A	0.4	0
B	0.2	111
C	0.25	10
D	0.1	1101
-	0.05	1100

Average # bits per symbol (generic)

= (0.4)*(1) + (0.2)*(3) + (0.25)*(2) + (0.1)*(4)
+ (0.05)*(4)

= 0.4 + 0.6 + 0.5 + 0.4 + 0.2 = 2.1 bits/symbol

Generic Compression Ratio

1 – (2.1/3) = 0.3 = 30%

where 3 is the # bits/symbol under fixed encoding scheme.

Huffman Codes

A	**0.4**	**0**
B	**0.2**	**111**
C	**0.25**	**10**
D	**0.1**	**1101**
-	**0.05**	**1100**

Specific Character/Symbol Sequence:

A	A	B	C	A	C	D	–	A	B
0	0	111	10	0	10	1101	1100	0	111

Total # bits in the above sequence = 22 bits
Average # bits / symbol in the above sequence = 22 / 10 = 2.2 bits/symbol
where 10 is the number of symbols in the above sequence

If we had used fixed-length encoding, we would have used:
3 bits/symbol * 10 symbols = 30 bits

Compression ratio = 1 – (22/30) = 26.7%

Sorting by Reversals: Reversal Distance

- Here, given a permutation π, we want to find the minimum number of reversals that transforms it to the identity permutation (1, 2, 3, … n)
- The reversal distance of π is denoted d(π).

π = 3 4 2 1 5 6 7 10 9 8

4 3 2 1 5 6 7 10 9 8

4 3 2 1 5 6 7 8 9 10

1 2 3 4 5 6 7 8 9 10

So $d(\pi) = 3$

Greedy Alg: Prefix based Sorting

- The number of elements (in a given permutation π) that are already in their final position in the identity permutation is called the prefix.
- We proceed in iterations 1 to n.
  - In iteration *i*, we do reversal in such a way (if needed) to bring element *i* to its final position in the identity permutation.
- Example: Sort by reversals: 5 6 1 3 2 4

		Prefix	# Reversals	# Integers moved around	
– Step 0:	5 6 1 3 2 4	0	0	0	
– Step 1:	1 ‖ 6 5 3 2 4	1	1	3	
– Step 2:	1 2 3 ‖ 5 6 4	3	2	4	
– Step 3:	1 2 3 4 ‖ 6 5	4	3	3	**Total**
– Step 4:	1 2 3 4 5 6	6	4	2	**15**

Prefix based Sorting

- Prefix based sorting does not guarantee the smallest number of reversals and at the worst case, could take *n*-1 reversals to arrive at the identity permutation 1, 2, …,

Step 0:	6 1 2 3 4 5	
Step 1:	1 6 2 3 4 5	2
Step 2:	1 2 6 3 4 5	2
Step 3:	1 2 3 6 4 5	2
Step 4:	1 2 3 4 6 5	2
Step 5:	1 2 3 4 5 6	2

5 Reversals
Total of 10 integers moved around

- But, the above permutation could be sorted in two reversals.

Step 0:	6 1 2 3 4 5	
Step 1:	5 4 3 2 1 6	6
Step 2:	1 2 3 4 5 6	5

2 Reversals
Total of 11 integers moved around

Activities Selection Problem

- Problem: Given a set of activities with a start time and finish time, we want to select the largest number of non-overlapping activities.

- Idea: Sort the activities in the increasing order of their finish time.
  - Select the activity a_i with the smallest finish time. Remove from the list of activities anything that overlaps with a_i .
  - Repeat the above procedure after a_i finishes.

- Time-Complexity: The pre-processing step of sorting the activities in the increasing order of finish times is the most dominating task. We can sort ‘n’ activities in $\Theta(n \log n)$ time.

Given List

Activity	1	2	3	4	5	6	7	8	9	10
Start	1	1	2	4	5	8	9	11	12	13
Finish	3	8	5	7	9	10	11	14	17	16

Sorted List

Activity	1	3	4	2	5	6	7	8	10	9
Start	1	2	4	1	5	8	9	11	13	12
Finish	3	5	7	8	9	10	11	14	16	17

Sorted List (Selected/ Discarded Activities)

Activity	1	3	4	2	5	6	7	8	10	9
Start	1	2	4	1	5	8	9	11	13	12
Finish	3	5	7	8	9	10	11	14	16	17

Optimal Solution = {a1, a4, a6, a8}

Proof of Optimality

- Theorem 1: At least one maximal conflict-free schedule includes the activity that finishes first.

- Proof (by contradiction): There may be several maximal conflict-free schedules.
- But, assume the activity finishing first (say *u*) is in none of them.
- Let X be one such maximal conflict-free schedule that does not include *u*. Let *v* be the activity finishing first in X.
- Since *u* finishes before *v*, *u* should not conflict with activities $X - \{v\}$.
- Hence, *v* could be removed from X and *u* could be inserted to X, leading to $X' = X \cup \{u\} - \{v\}$.
- The set X' featuring *u* would also be a maximal conflict-free schedule.

Proof of Optimality

- Theorem 2: The greedy schedule formed based on the earliest finishing activities is optimal.

- Proof:
- Let *u* be the earliest finishing activity. According to Theorem 1, *u* will be part of some maximal conflict-free schedule X.
- Since *u* is the earliest finishing activity, it should be the first activity in X.
- Among all the activities that overlap with *u* in X, only one of them could be selected for X (in this case, *u* is indeed selected for X).
- Let Y = X – {*u*} – {set of all activities overlapping with *u*}. The optimality of the conflict-free schedule for Y will hold true due to induction.

Designing a Tape for File Read: Ex 1

- Unlike a disk, a tape is read sequentially.
- If a tape has a sequence of files and a particular file is to be read, then all the preceding files have to be scanned before reaching the target file.
- If each file is equally likely to be accessed, an optimal strategy to minimize the average cost for searching a random file would be to store the files in the increasing order of size.

File Index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
File Size	**10**	**15**	**5**	**20**	**45**	**12**	**25**	**18**

Storing as it is in the increasing order of file index

File Index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
File Size	**10**	**15**	**5**	**20**	**45**	**12**	**25**	**18**
Cost to Access	**10**	**25**	**30**	**50**	**95**	**107**	**132**	**150**

Average cost to access any file = (10 + 25 + 30 + 50 + 95 + 107 + 132 + 150) / 8
= 74.88

Designing a Tape for File Read: Ex 1

File Index	1	2	3	4	5	6	7	8
File Size	10	15	5	20	45	12	25	18

Sorting based on the File Size and storing in the increasing order of file size

File Index	3	1	6	2	8	4	7	5
File Size	5	10	12	15	18	20	25	45
Cost to Access	5	15	27	42	60	80	105	150

Average cost to access any file = (5 + 15 +27 + 42 + 60 + 80 + 105 + 150) / 8
= 60.5

Designing a Tape for File Read: Ex 2

File Index	1	2	3	4	5	6	7	8
File Size	10	15	5	20	45	12	25	18
Acc. Frequency	5	10	8	7	9	6	12	13
Size/Frequency	2	1.5	0.625	2.857	5	2	2.083	1.385

Sorting based on the increasing order of File Size / Access Frequency

File Index	3	8	2	1	6	7	4	5
File Size	5	18	15	10	12	25	20	45
Acc. Freq.	8	13	10	5	6	12	7	9
Size/Frequency	0.625	1.385	1.5	2	2	2.083	2.857	5
Cost to Access	5	23	38	48	60	85	105	150
Cost*Freq	40	299	380	240	360	1020	735	1350

$$\text{Average cost to access any file} = \frac{(40 + 299 + 380 + 240 + 360 + 1020 + 735 + 1350)}{(8 + 13 + 10 + 5 + 6 + 12 + 7 + 9)}$$

$$= 63.2$$

Designing a Tape for File Read: Ex 2

File Index	1	2	3	4	5	6	7	8
File Size	10	15	5	20	45	12	25	18
Acc. Frequency	5	10	8	7	9	6	12	13
Size/Frequency	2	1.5	0.625	2.857	5	2	2.083	1.385

Sorting based on the increasing order of File Index only

File Index	1	2	3	4	5	6	7	8
File Size	10	15	5	20	45	12	25	18
Acc. Frequency	5	10	8	7	9	6	12	13
Cost to Access	10	25	30	50	95	107	132	150
Cost*Freq	50	250	240	350	855	642	1584	1950

$$\text{Average cost to access any file} = \frac{(50 + 250 + 240 + 350 + 855 + 642 + 1584 + 1950)}{(5 + 10 + 8 + 7 + 9 + 6 + 12 + 13)}$$

$$= 84.58$$

Designing a Tape for File Read: Ex 2

File Index	1	2	3	4	5	6	7	8
File Size	10	15	5	20	45	12	25	18
Acc. Frequency	5	10	8	7	9	6	12	13
Size/Frequency	2	1.5	0.625	2.857	5	2	2.083	1.385

Sorting based on the increasing order of File Size only

File Index	3	1	6	2	8	4	7	5
File Size	5	10	12	15	18	20	25	45
Acc. Freq.	8	5	6	10	13	7	12	9
Cost to Access	5	15	27	42	60	80	105	150
Cost*Freq	40	75	162	420	780	560	1260	1350

$$\text{Average cost to access any file} = \frac{(40 + 75 + 162 + 420 + 780 + 560 + 1260 + 1350)}{(8 + 5 + 6 + 10 + 13 + 7 + 12 + 9)}$$

$$= 66.38$$

Proof of Optimality

Given:

File Index	**1**	**2**
File Size	**10**	**15**
Acc. Frequency	**5**	**10**
Size/Frequency	**2**	**1.5**

File Size / Frequency

	b	a
File Index	**2**	**1**
File Size	**15**	**10**
Acc. Frequency	**10**	**5**
Cost to Access	**15**	**25**
Cost * Freq	**150**	**125**

File Size Only

a	b
1	**2**
10	**15**
5	**10**
10	**25**
50	**250**

- We want to prove that we get an optimal solution, when:

$$\frac{L[\pi(i)]}{F[\pi(i)]} \leq \frac{L[\pi(i+1)]}{F[\pi(i+1)]} \text{ for all } i$$

- Where, π(i) is the position of File i in the sorted order of Size/Frequency; L[π(i)] is the length of File i.
- Suppose L[π(i)] / F[π(i)] > L[π(i+1)] / F[π(i+1)] for some i.
- To simplify notation, let a = π(i) and b = π(i+1). L[a]/F[a] > L[b]/F[b]
- If we swap files a and b, then the cost of accessing a increases by L[b] and the cost of accessing b decreases by L[a]. Overall, the swap changes the total cost by L[b]F[a] – L[a]F[b] < 0. This is an improvement! We do this for all consecutive pairs a and b.