Data Structures and Algorithms

- Algorithm Design Techniques -

School of Electrical Engineering
Korea University

Algorithm Design

- So far, we focused on the efficient implementation of algorithms
  - Actual data structures ignored
  - The programmer is in charge

- Shift to the design of algorithms
  - Five common types of algorithms to solve problems
  - At least one of them works for many problems

Algorithm Design Types

1. Greedy Algorithms
2. Divide and Conquer
3. Dynamic Programming
4. Randomized Algorithms
5. Backtracking Algorithms

Greedy Algorithms

- Work in phases.
- In each phase, a decision is made that appears to be good, ignoring future consequences
- Take local optimum now, hoping that it is equal to the global optimum.
  - If this is the case, the algorithm is correct
  - Otherwise, it produced a suboptimal solution
- Simple greedy algorithms for approximate answers.
- More complicated algorithms for exact answer

Greedy Algorithms: Examples

- Dijkstra's, Prim's, and Kruskal's algorithms
- Coin-changing problem
  - To make change in U.S. currency, repeatedly dispense the largest denomination

(Ex) 17.61 dollars

one ten-dollar bill

one five-dollar bill

two one-dollar bills

two quarters, one dime, one penny

→ minimize the number of bills and coins

A Simple Scheduling Problem

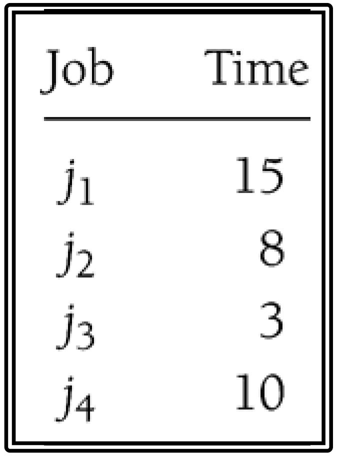
Input:

- Jobs $j_1, j_2, \cdots, j_N$, all with known running times $t_1, t_2, \cdots, t_N$, respectively.
- A single processor

Goal:

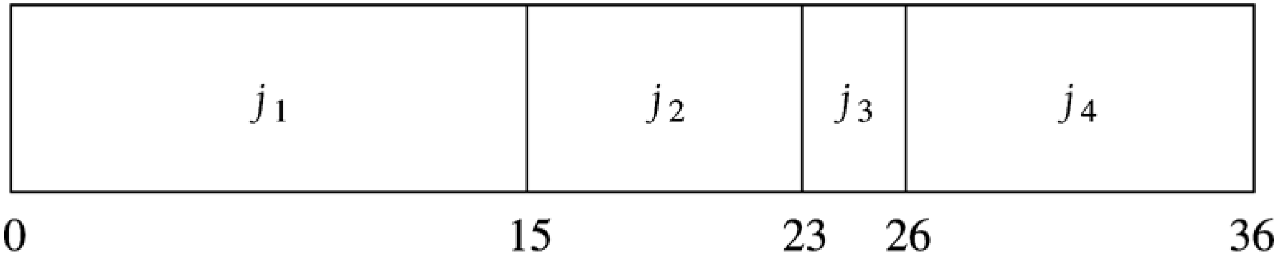
- A best schedule to minimize the average completion time of jobs
- Assuming *non-preemptive scheduling*

Example: Four jobs



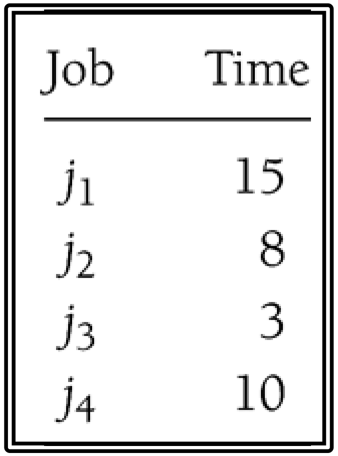
Job	Time
j_1	15
j_2	8
j_3	3
j_4	10

Figure 10.2 Schedule #1



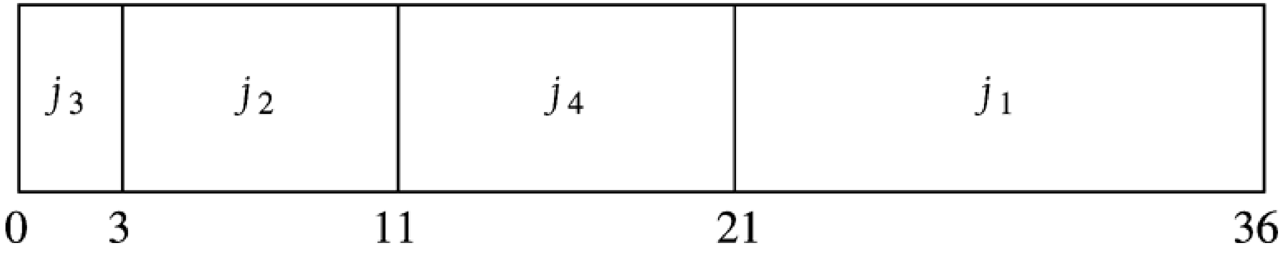
Average completion time
= (15 + 23 + 26 + 36) / 4
= 100 / 4
= 25

Example: Four jobs



Job	Time
j_1	15
j_2	8
j_3	3
j_4	10

Figure 10.3 Schedule #2 (optimal)



Average completion time
= (3 + 11 + 21 + 36) / 4
= 71 / 4
= 17.75

Example: Four jobs

- The second one is arranged by shortest job first, which always yields an optimal schedule.
- Generally, the total cost C of the schedule is defined by the following equation:

$$\mathrm{C} = \sum_{k=1}^{n} (N - k + 1) t_{i_k}$$

$$\mathrm{C} = (N + 1) \sum_{k=1}^{n} t_{i_k} - \sum_{k=1}^{n} k * t_{i_k}$$

- The first sum is independent of the job ordering.
- The second sum affects the total cost.

The Multiprocessor Case

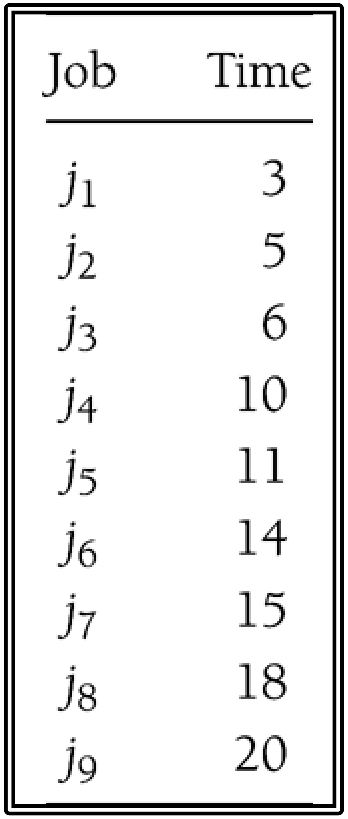
Input:

- Jobs $j_1, j_2, \cdots, j_N$, all with known running times $t_1, t_2, \cdots, t_N$, respectively.
- A number P of processors

Goal:

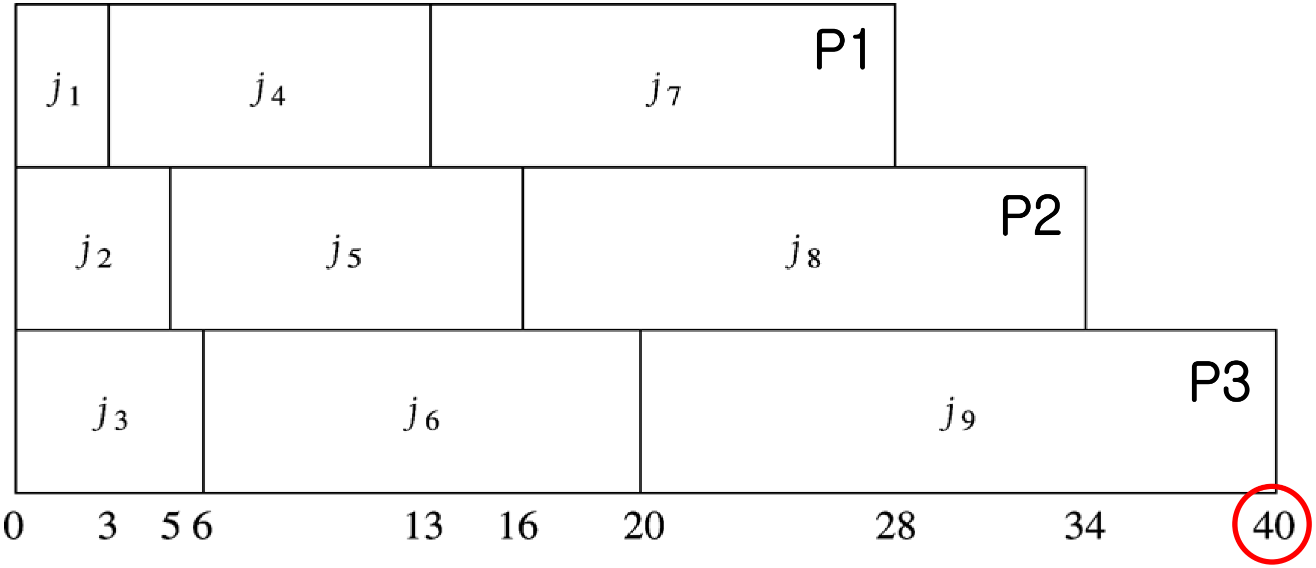
- A best schedule to minimize the average completion time of jobs
- Assuming *non-preemptive scheduling*

Example



Job	Time
j_1	3
j_2	5
j_3	6
j_4	10
j_5	11
j_6	14
j_7	15
j_8	18
j_9	20

Fig 10.5 An optimal solution

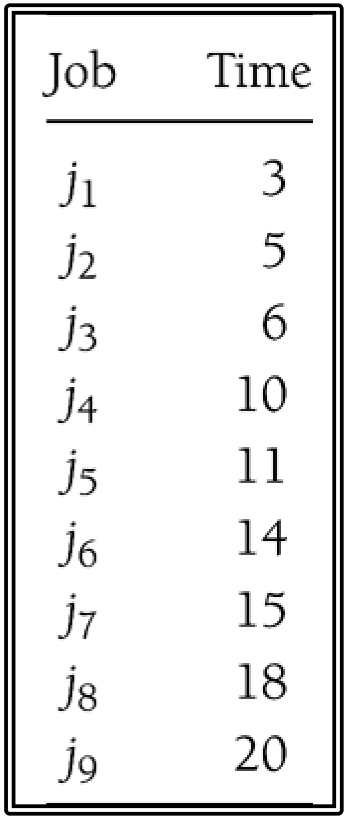


Mean completion time

= (3+5+6+13+16+20+28+34+40) / 9

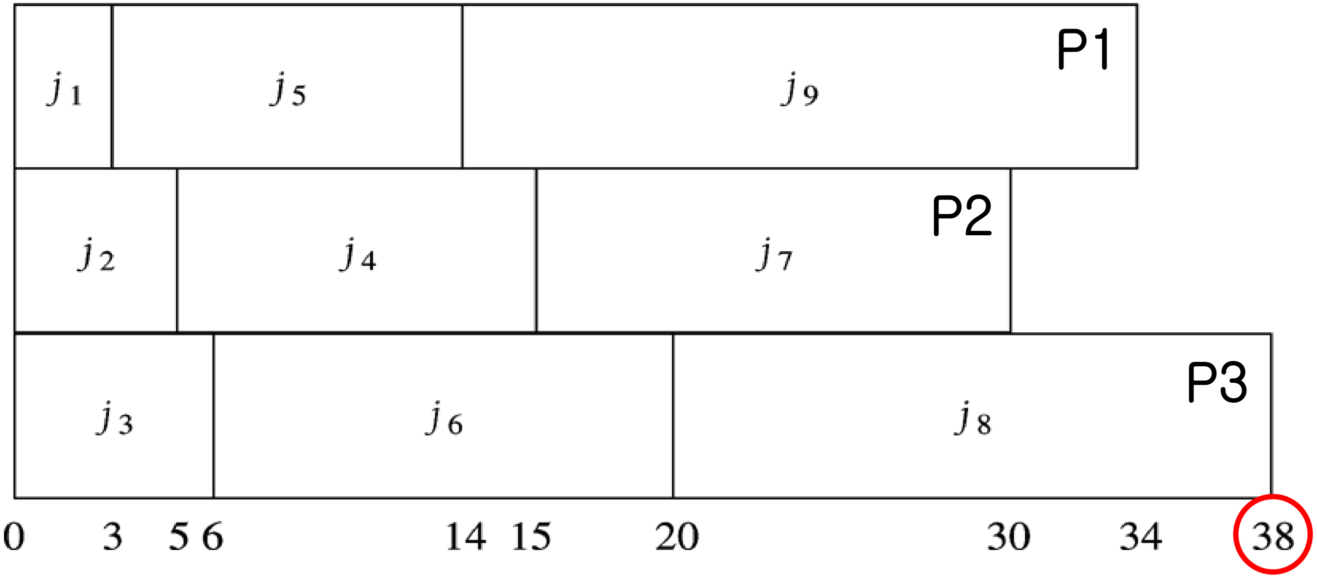
= 165 / 9 = 18.33

Example



Job	Time
j_1	3
j_2	5
j_3	6
j_4	10
j_5	11
j_6	14
j_7	15
j_8	18
j_9	20

Fig 10.6 A second optimal solution



Mean completion time

= (3+5+6+14+15+20+30+34+38) / 9

= 165 / 9 = 18.33

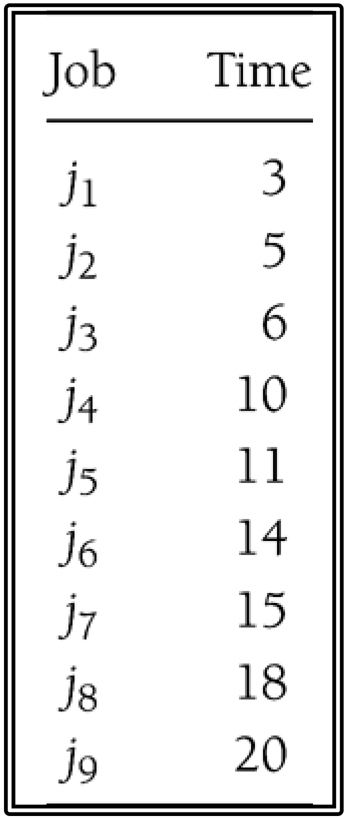
Final completion time

- What if we are only interested in when the last job finishes: final completion time
- In the previous two schedules, these completion times are 40 and 38.
- In the next schedule,

  the final completion time is 34. However,

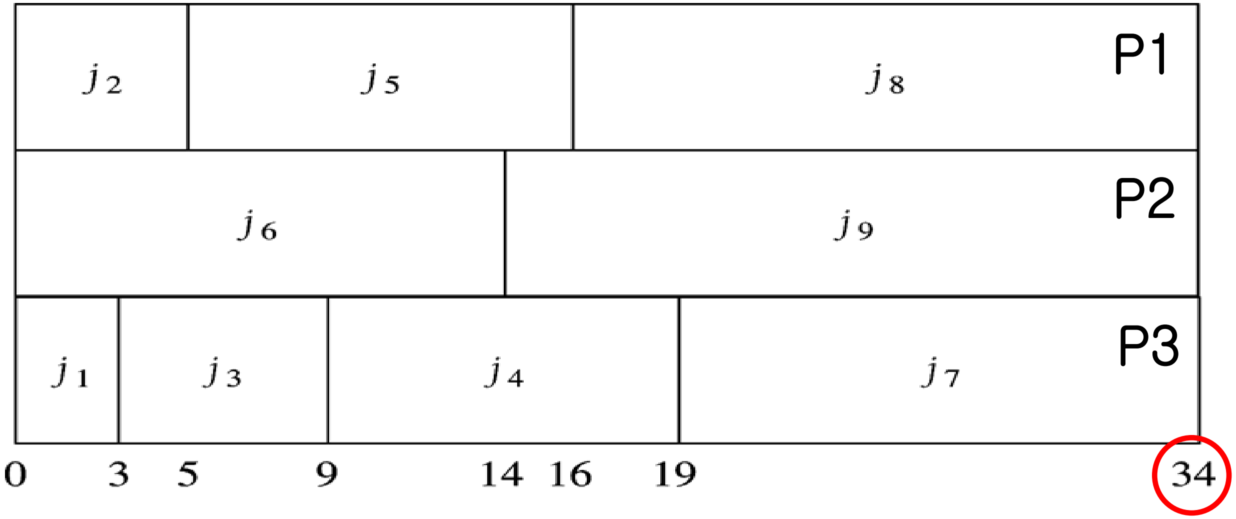
  its total completion time is 168.
- Generally, total completion time and final completion time do not go together.

Final completion time



Job	Time
j_1	3
j_2	5
j_3	6
j_4	10
j_5	11
j_6	14
j_7	15
j_8	18
j_9	20

Fig 10.7 Minimizing final completion time



Mean completion time

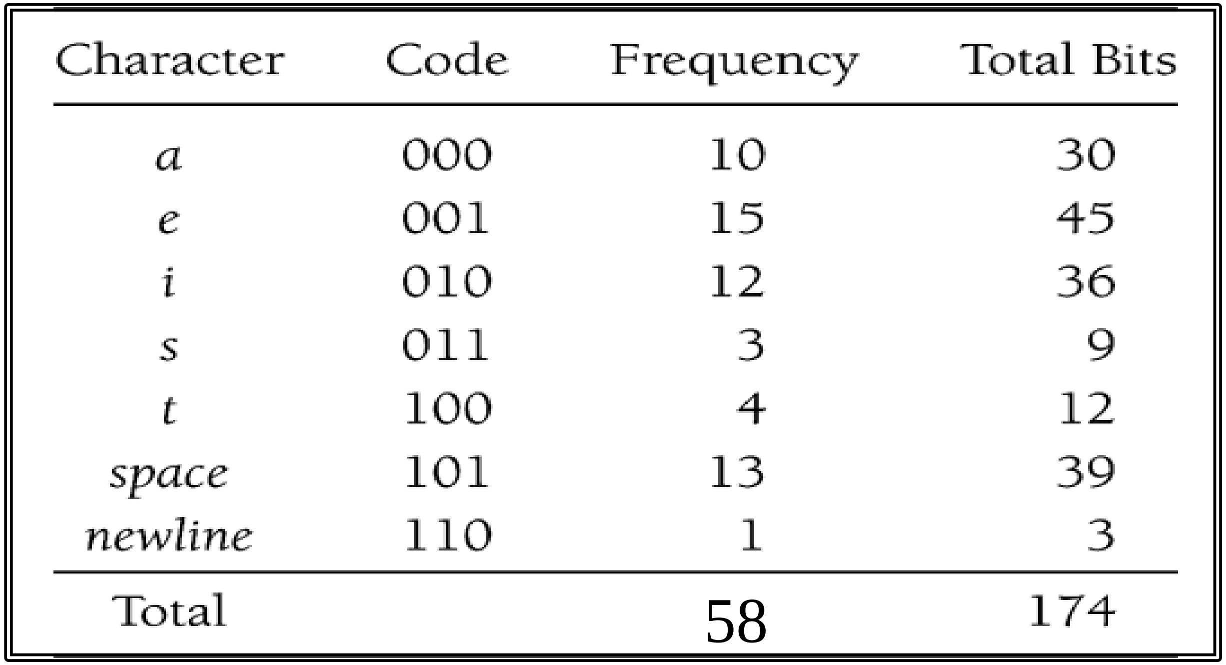
= 168 / 9 = 18.66

Huffman Codes

- Known as file compression
- The normal ASCII character set consists of roughly 100 printable characters
- 7 bits are required to distinguish them.
- An eighth bit is added as a parity check.
- If the size of the character set is C, then $|\log C|$ bits are needed in a standard encoding

Example

- Suppose that a file contains only ***a*, *e*, *i*, *s*, *t*, *blanks*** and ***newlines*** with the following frequency:



Character	Code	Frequency	Total Bits
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
space	101	13	39
newline	110	1	3
Total		58	174

Fig 10.8 Using a standard coding scheme

Huffman Codes

- In real life, files can be very large.
- There is usually a big disparity between the most frequent and least frequent characters.
- Reducing the file size might be preferred in some cases such as transmitting over a slow network line.
- Can achieve 25% or more savings on typical large files
- The general strategy is to use short codes for frequently occurring characters

Tree Representation

- The binary code for the alphabet can be represented by the binary tree.

Total bits = 174

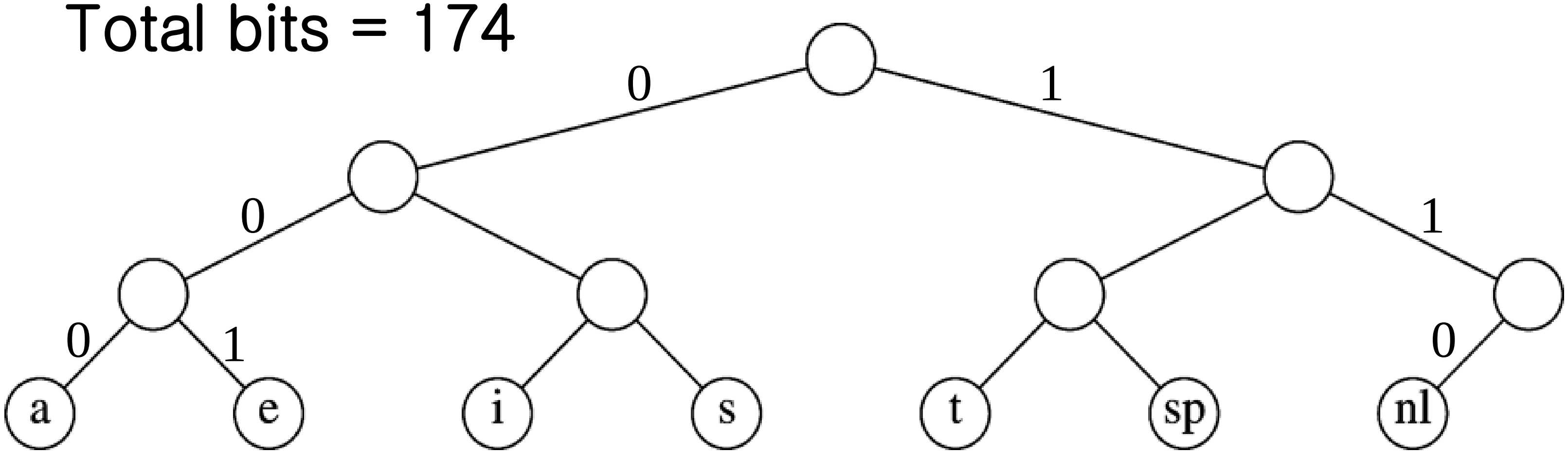


Fig 10.9 Representation of the original code in a tree

Tree Representation

Total bits = 173

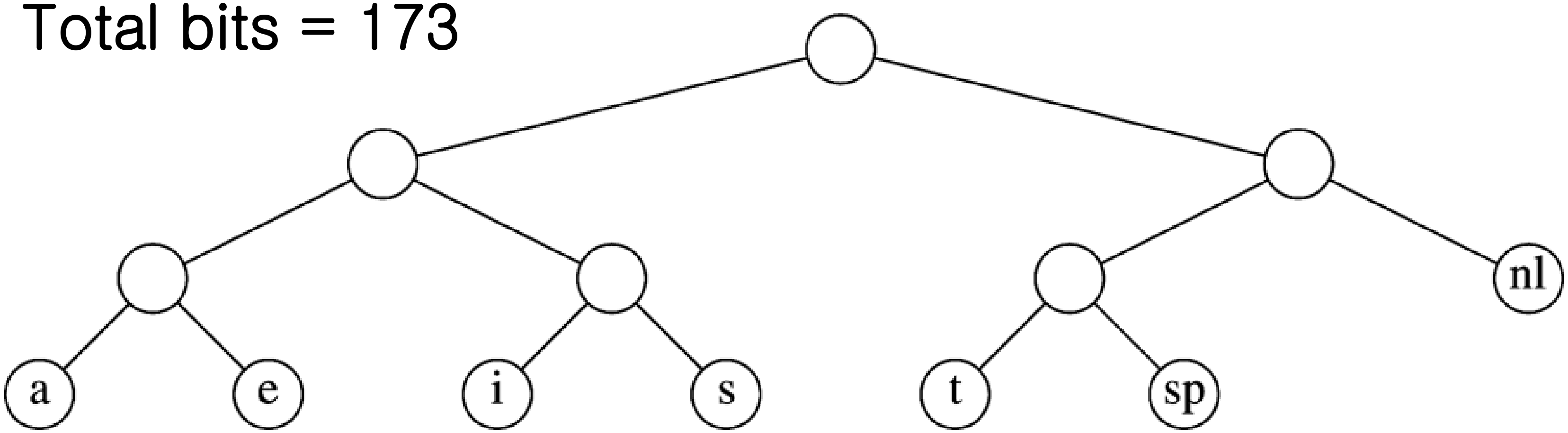


Fig 10.10 A slightly better tree

Observations

1. The optimal tree should be a full tree: All nodes either are leaves or have two children
   - Otherwise, nodes with only one child could move up a level.

2. The characters should be placed only at the leaves: Any sequence of bits can be decoded unambiguously.
   - If a character is contained in a nonleaf node, it is not possible to guarantee that the decoding will be unambiguous.

Optimal prefix code

- Prefix code: No character code is a prefix of another character code.

Total bits = 146

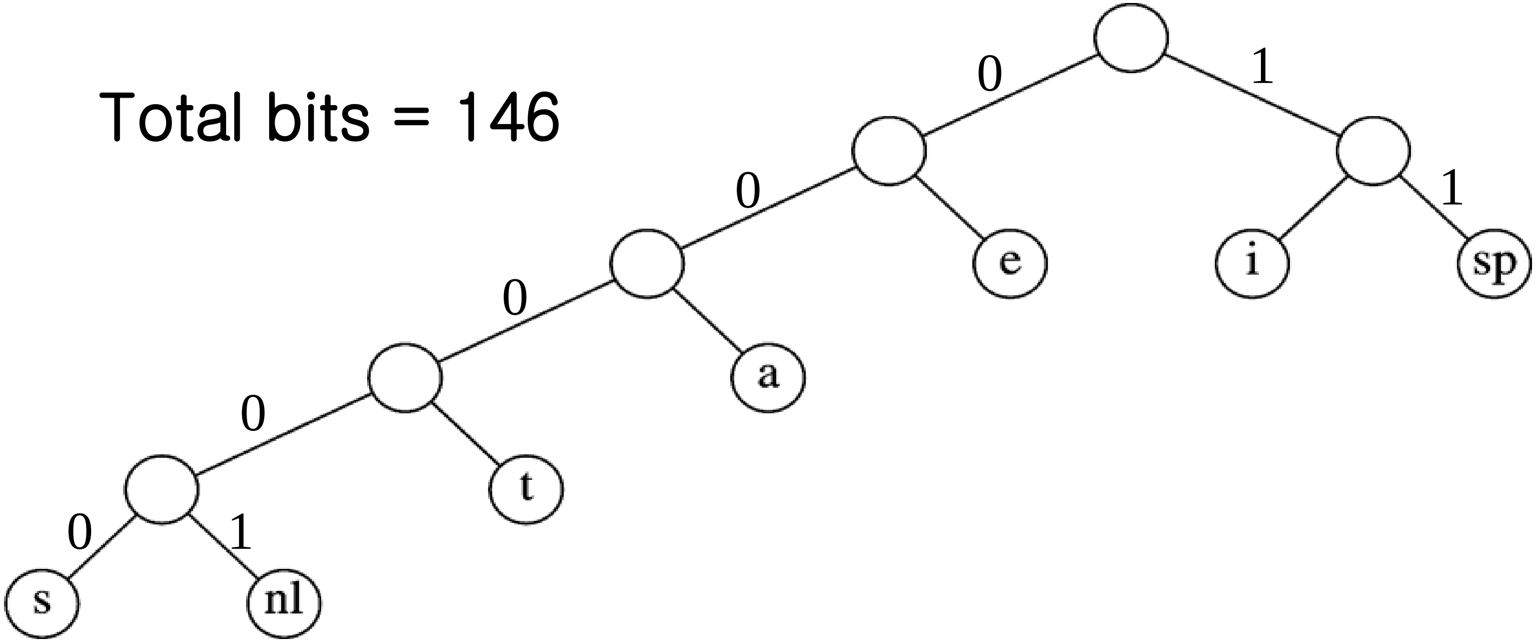
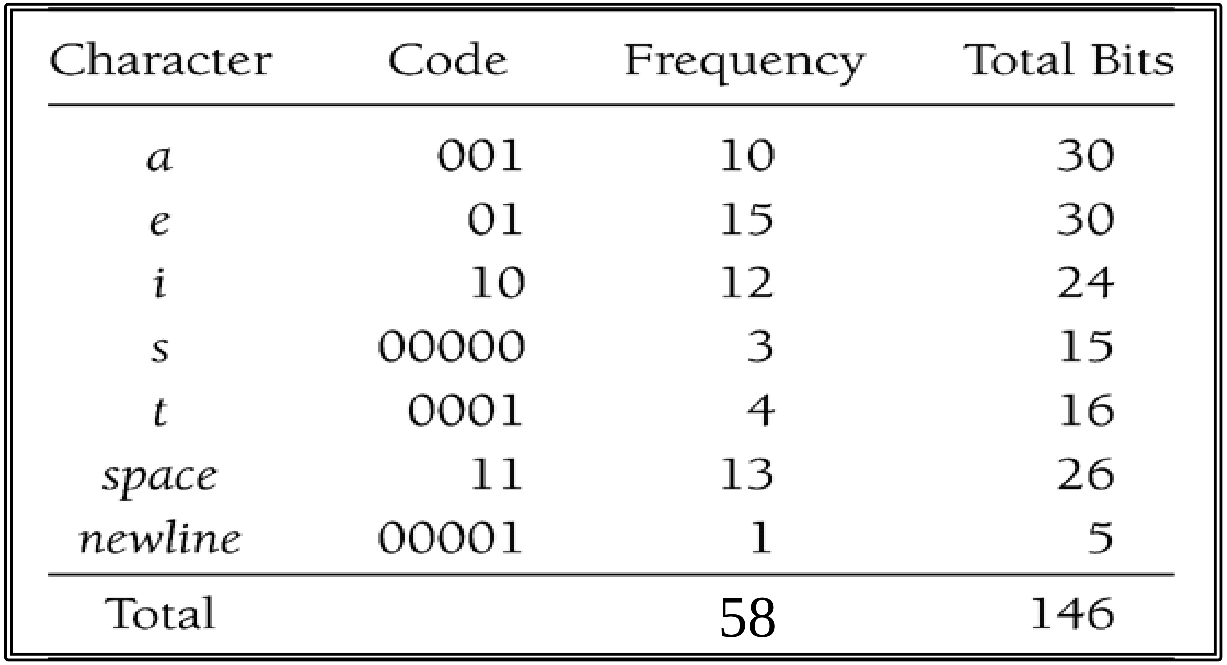


Fig 10.11 Optimal prefix code

Optimal prefix code



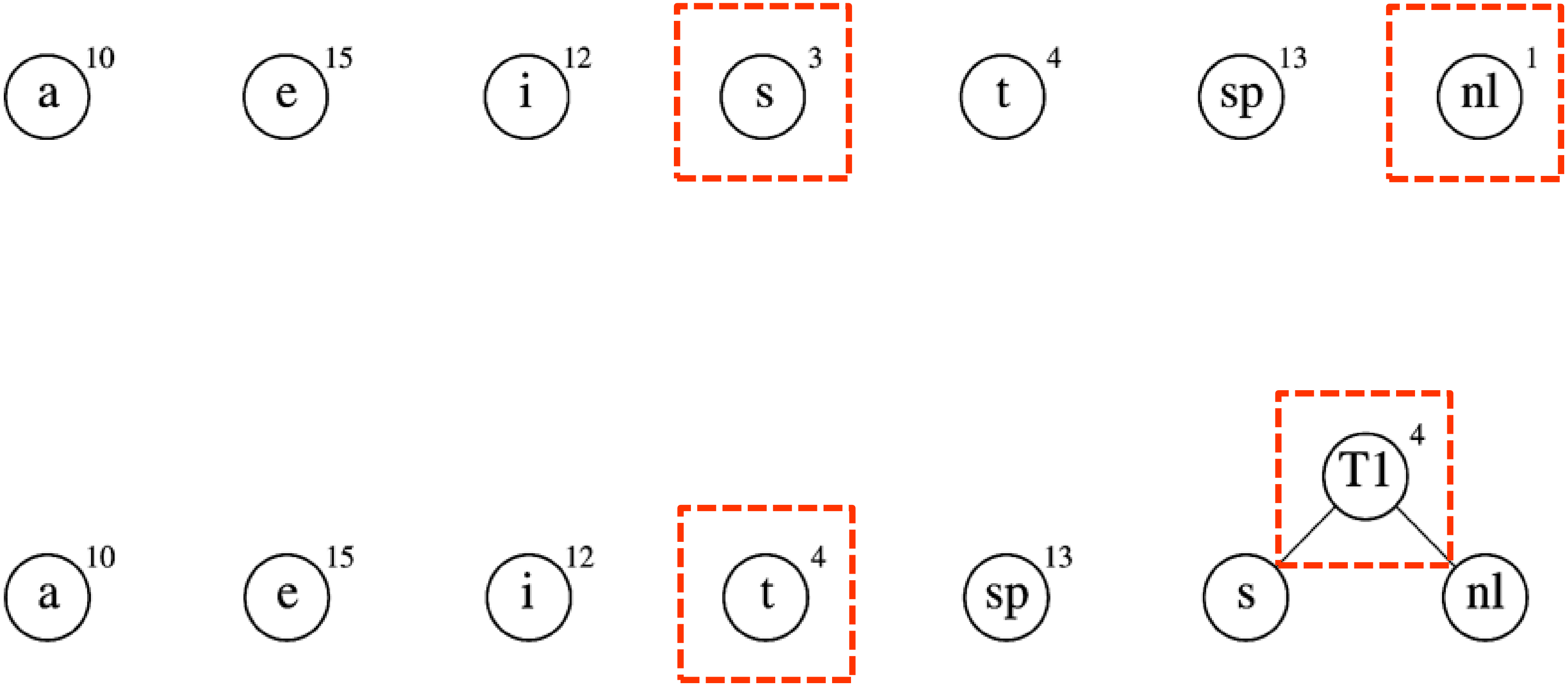
Character	Code	Frequency	Total Bits
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
space	11	13	26
newline	00001	1	5
Total		58	146

Fig 10.12 Optimal prefix code

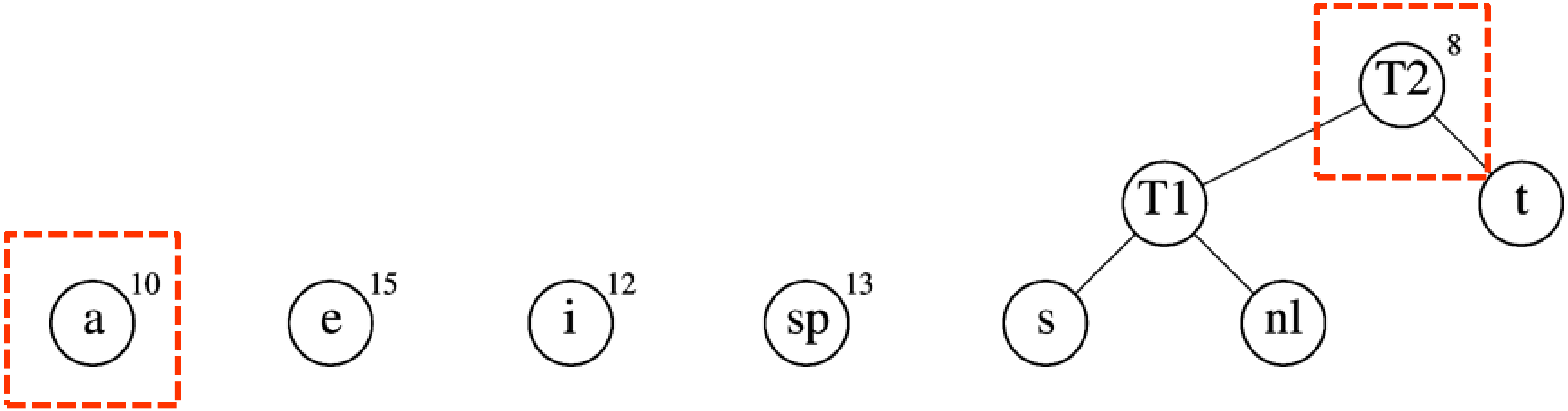
Huffman Codes

- How the coding tree is constructed?
- By Huffman in 1952.
- The coding system is called as Huffman code
- Algorithm sketch: Given a forest of C single node trees–one for each character. The weight of a tree is equal to the sum of the frequencies of its leaves. C–1 times, select the two trees, T_1 and T_2, of smallest weight, breaking ties arbitrarily, and form a new tree with subtrees $T_1$and T_2

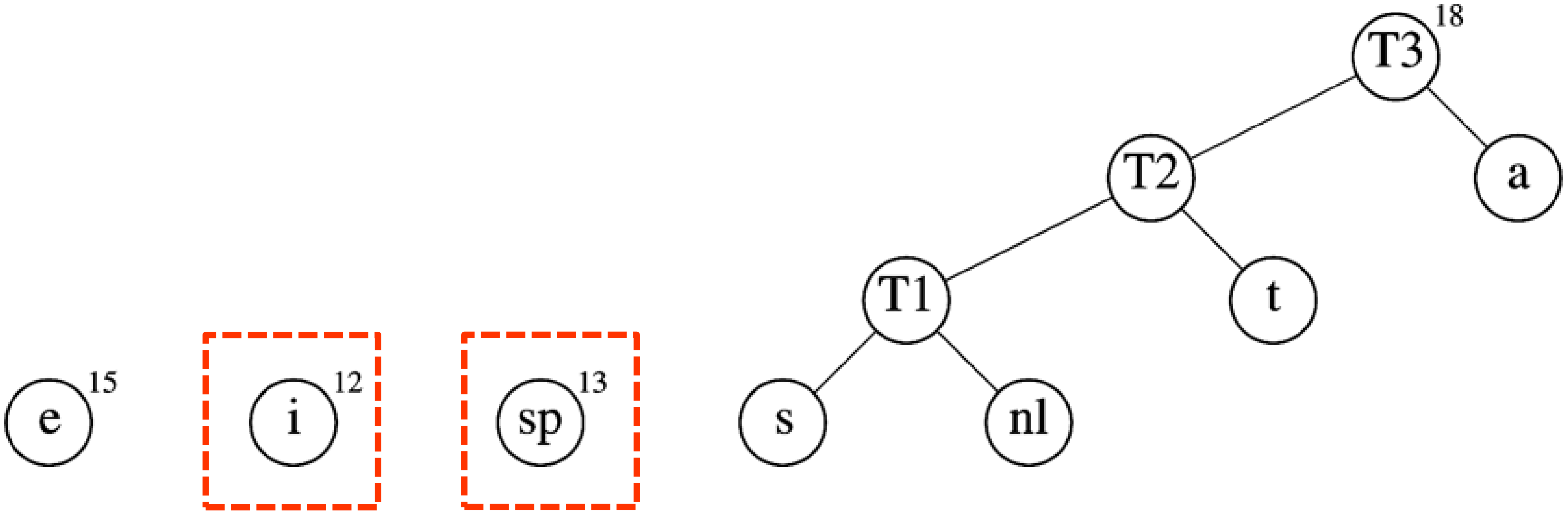
Example (1/6)



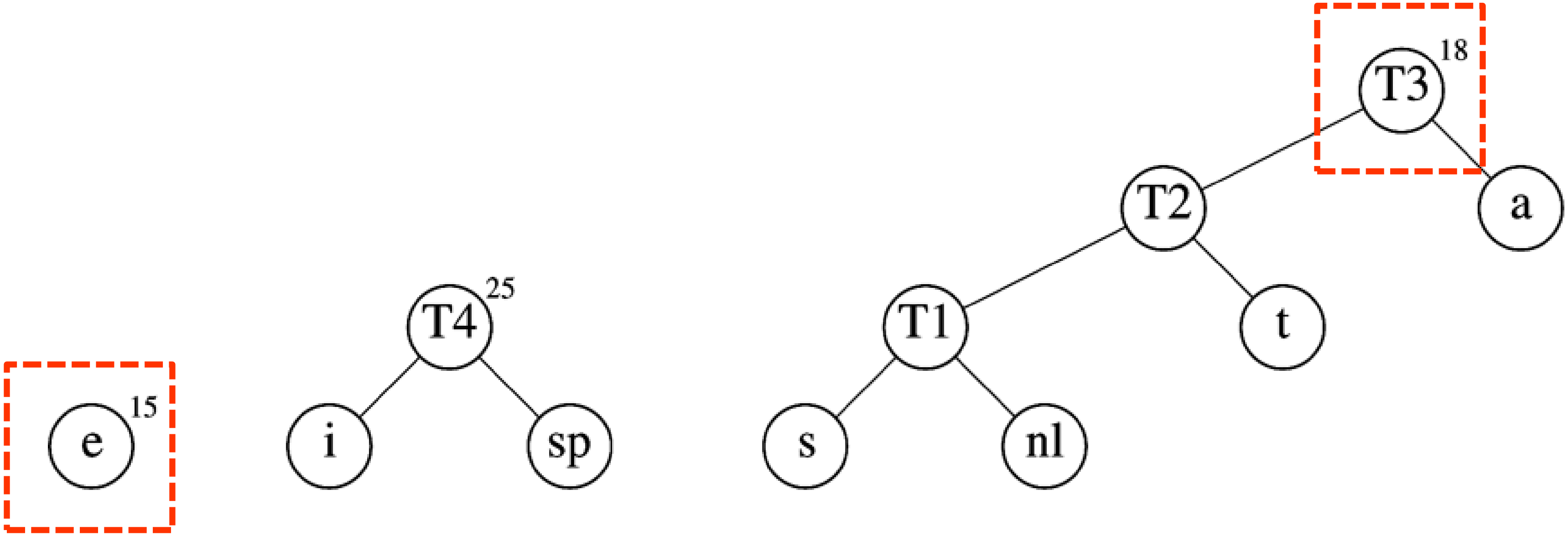
Example (2/6)



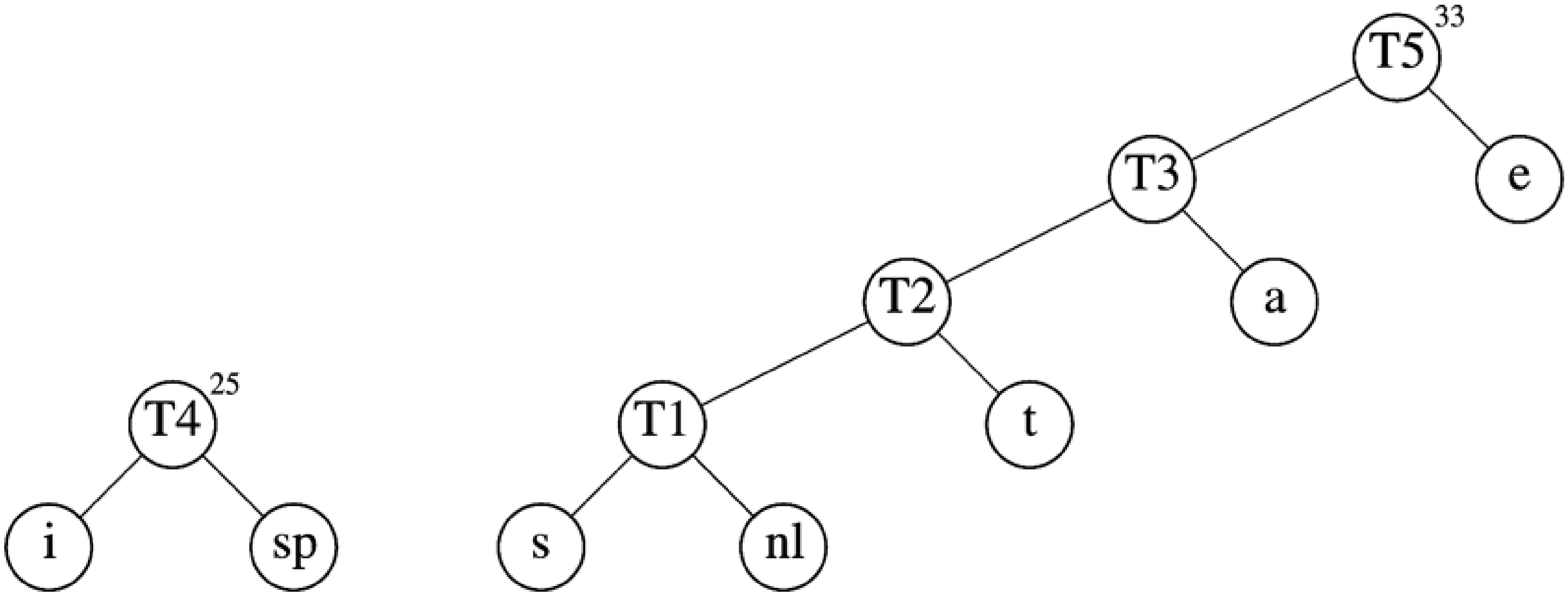
Example (3/6)



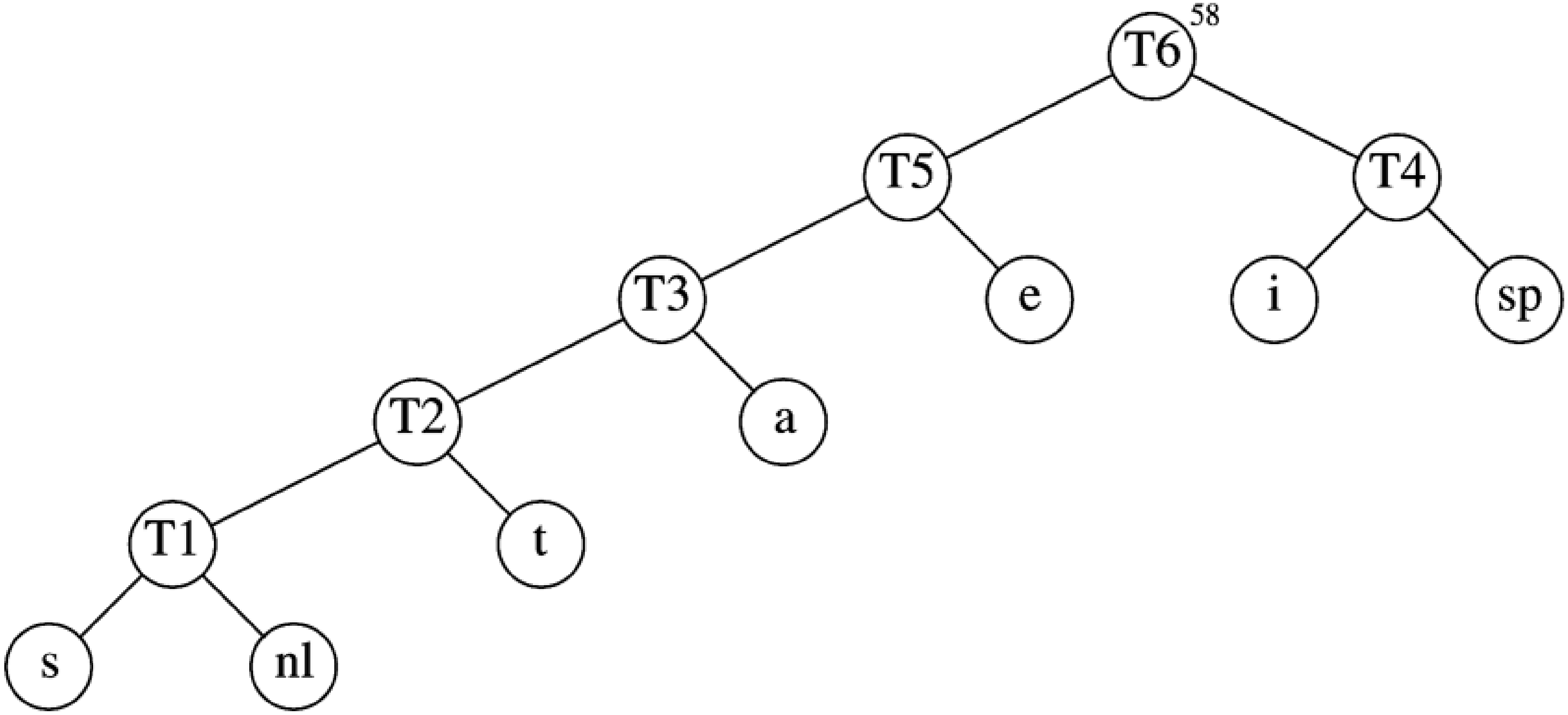
Example (4/6)



Example (5/6)



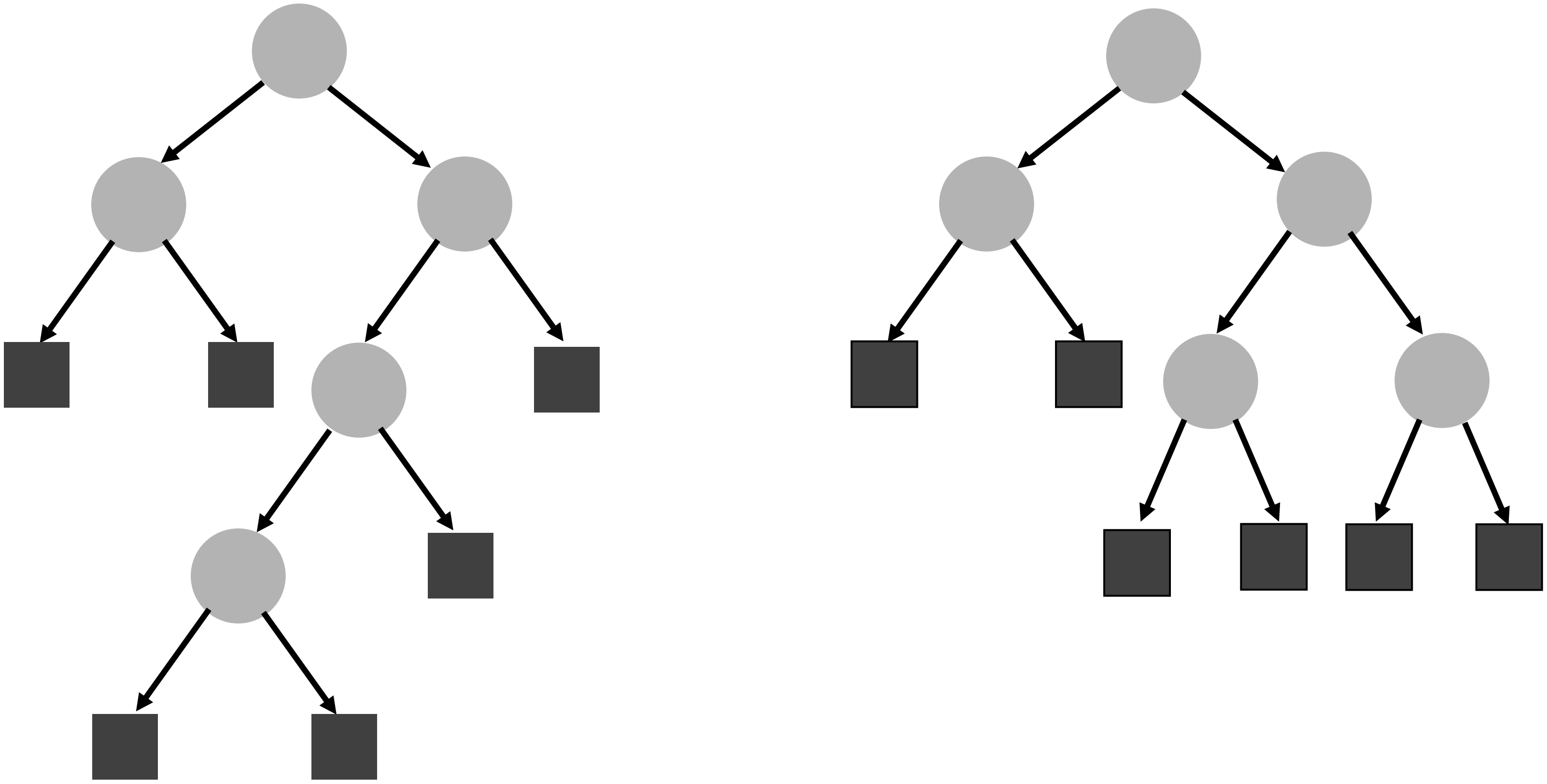
Example (6/6)



Extended Binary Tree

- Augment binary tree with a special “square” node at every place there is a null link: external node
- Every binary tree with n nodes has n+1 null links.
- Every binary tree with n nodes has n+1 external nodes.
- External (Internal) path length $E(I)$of a binary tree is the sum of the lengths of the paths from the root to all external (internal) nodes

Extended Binary Tree



I = 0 + 1 + 1 + 2 + 3 = 7

E = 2 + 2 + 2 + 3 + 4 + 4 = 17

● **internal node**

■ **external node**

Properties

- The internal and external path lengths I and E of a binary tree with n internal nodes are related by the formula $E = I + 2n$.
- It follows that binary trees with the maximum E also have maximum I.
- Question: Over all binary trees with n internal nodes, what is the maximum and minimum possible values for I ?
- The worst case is when the tree is

$$I = \sum_{i=0}^{n-1} i = n*(n-1) / 2$$

Properties

- For minimum I, put as many internal nodes as close to the root as possible.

$$0 + 2 * 1 + 4 * 2 + 8 * 3 + \ldots$$

$$\rightarrow \sum_{1}^{n} \lfloor \log k \rfloor = O(n * \log n)$$

- One such example: Complete binary tree

Weighted External Path Length

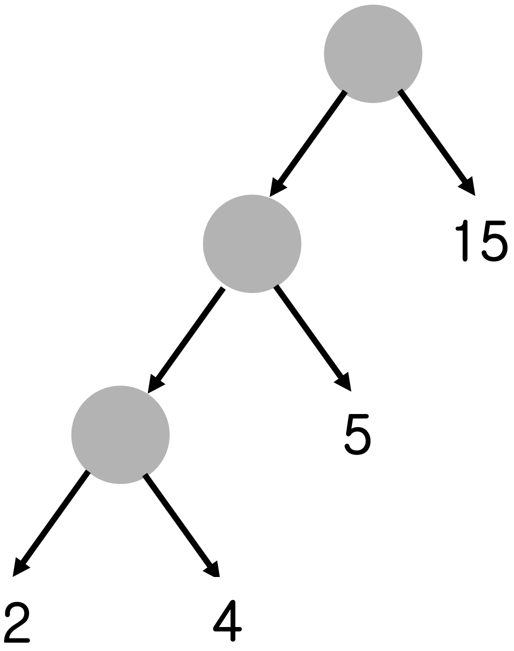
- From a set of $n+1$ positive weights $q_1, q_2, \ldots,$ q_{n+1}, each of the $n+1$ external nodes in a binary tree is associated with one of the weights.

- Weighted External Path Length

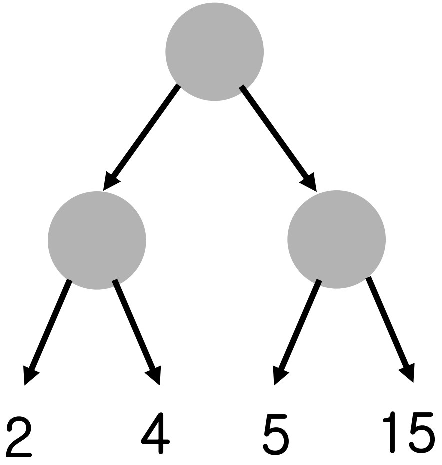
$$WE = \sum_{1}^{n+1} q_i * k_i$$

where k_i is the distance from the root node to the external node with weight q_i

Example



WE = 2*3 +4*3+5*2+15*1
= 43

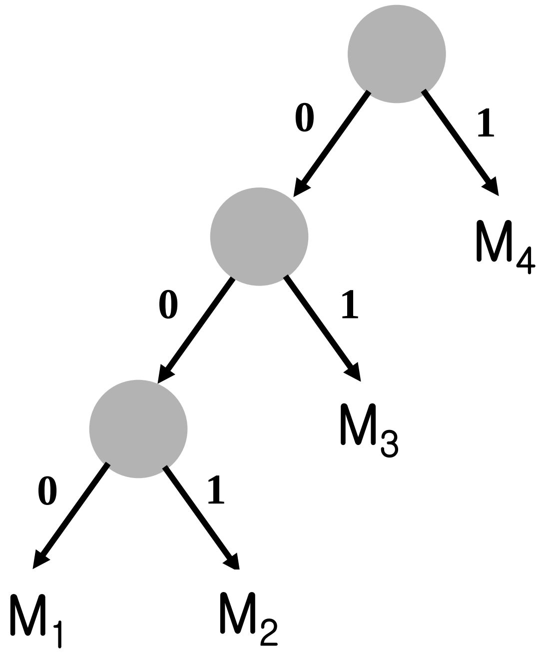


WE = 2*2 +4*2+5*2+15*2
= 52

Application

- An optimal set of codes for messages $M_1, \ldots, M_{n+1}$ to transmit the corresponding messages.
- At the receiving end, the code will be decoded using a decode tree.
- A decode tree is a binary tree in which external nodes represent messages
- The binary bits in the codes determine the branching needed at each level of the decode tree to reach the correct external node.

Decode tree



- Codes for messages

M_1 : 0 0 0
M_2 : 0 0 1
M_3 : 0 1
M_4 : 1

Huffman codes

- The cost of decoding a code word is proportional to the number of bits in the code
- Is equal to the distance of the corresponding external node from the root node.

Problem Formalism

- Assume q_i is the relative frequency with which message M_i will be transmitted, then the expected decode time is

$$T = \sum_{1}^{n+1} q_i * d_i$$

where d_i is the distance of the external node for the message M_i from the root node

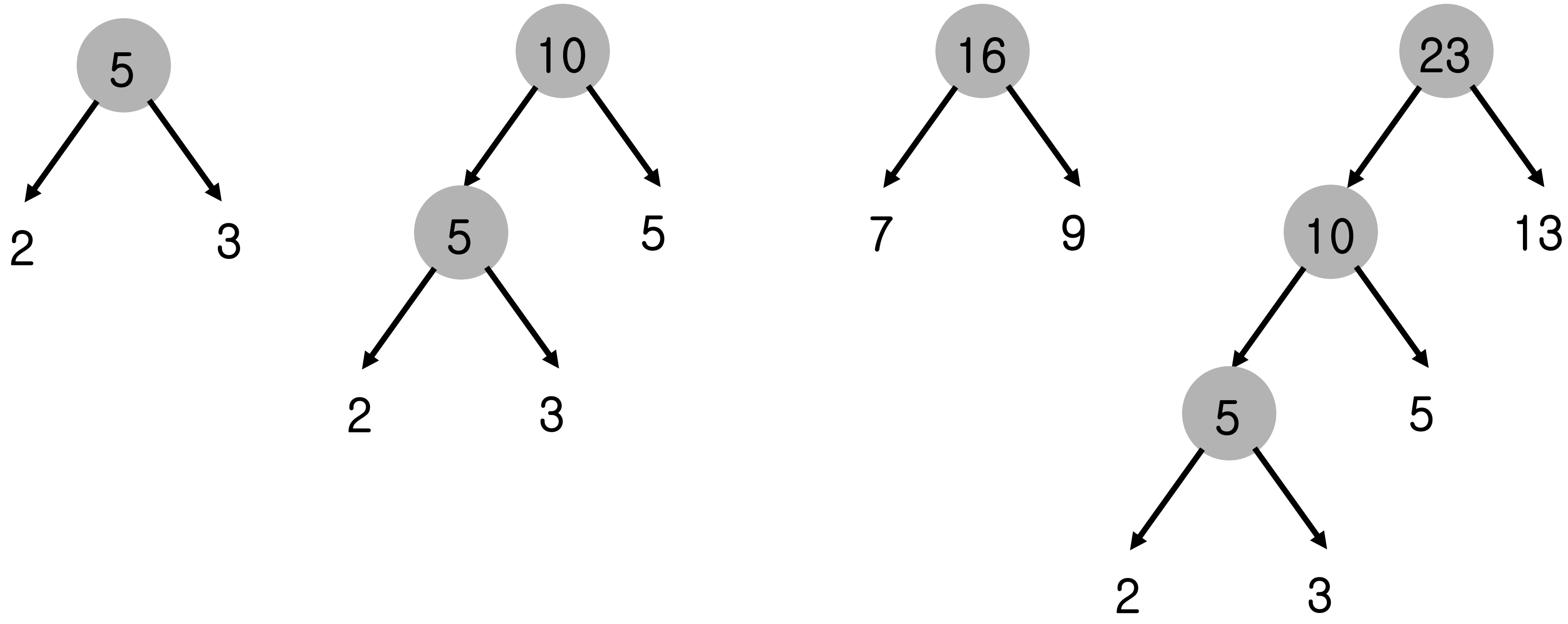
- The expected decode time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length

Algorithm

```
procedure HUFFMAN (L, n) {
   //L is a list of n single node binary trees
   for i = 1 to n−1 do {
       GETNODE(T);   //create a new binary tree by
       LCHILD(T) ← LEAST(L); //combining the trees with
       RCHILD(T) ← LEAST(L); //the two smallest weights
       WT(T) ← WT(LCHILD(T)) + WT(RCHILD(T));
       INSERT (L, T)
     }
}
```

Example

$q_1 = 2$, $q_2 = 3$, $q_3 = 5$, $q_4 = 7$, $q_5 = 9$, and $q_6 = 13$



Approximate Bin Packing

- Solve the bin packing problem
- Run quickly but will not necessarily produce optimal solutions
- The solutions are not too far from optimal

Approximate Bin Packing

- Input

  N items of size s_1, s_2, ⋯, s_N where $0 < s_i \leq 1$

- Goal

  Pack the items in the fewest no. of bins.

Optimal Packing

- 7 items with sizes 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

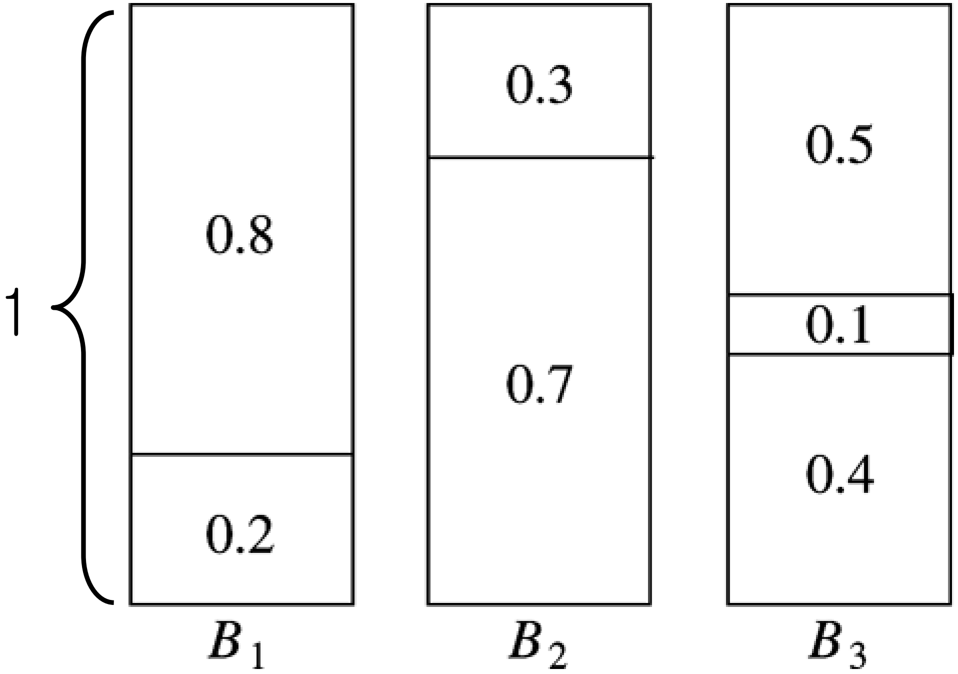


Fig 10.20 Optimal packing

Bin Packing Algorithm

- Two versions
  - On-line bin packing : each item must be placed in a bin before the next item can be processed and the decision can’t be changed
  - Off-line bin packing: it is not necessary to do anything until all the input has been read

On-line Algorithms

- An on–line algorithm cannot always give an optimal solution.
- Theorem: There are inputs that force any on–line bin packing algorithm to use at least 4/3 the optimal number of bins.
- Three simple algorithms that guarantee that the number of bins used is no more than twice optimal.

Next Fit

- Probably the simplest algorithm
- When processing any item, check whether it fits in the same bin as the last item.
  - If it does, it is placed there
  - Otherwise, a new bin is created.

Next fit

- 7 items with sizes 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

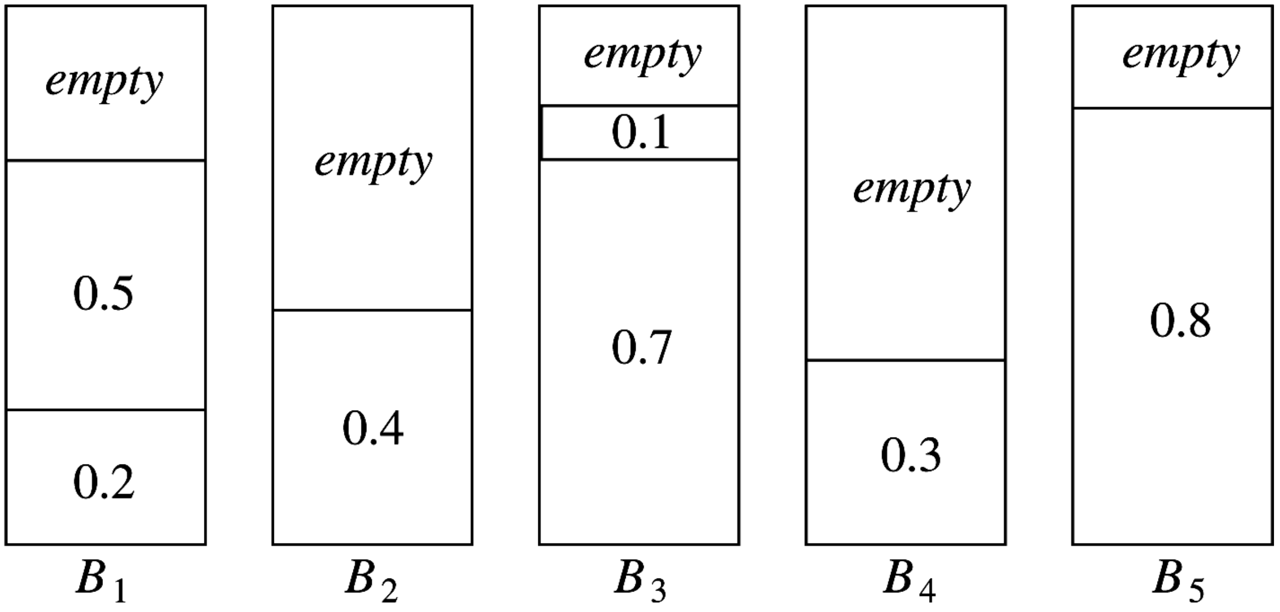


Fig 10.21 Using Next fit

Next Fit

- (Theorem10.2) Let $\boldsymbol{M}$ be the optimal number of bins required to pack a list $\boldsymbol{I}$ of items. Then next fit never uses more than $2\boldsymbol{M}$ bins. There exist sequences such that next fit uses $2\boldsymbol{M} - 2$ bins.

Example for Theorem 10.2

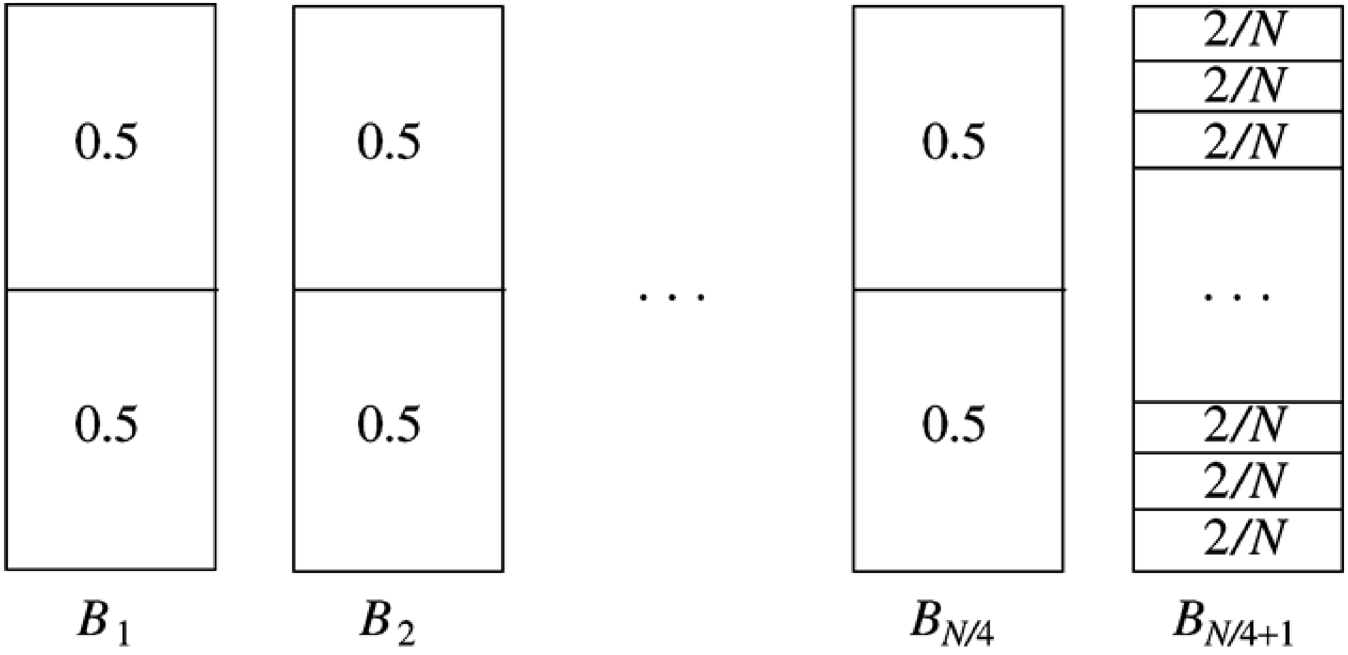


Fig 10.22 Optimal packing for 0.5, 2/N, 0.5, 2/N,… where N is divisible by 4

Example for Theorem 10.2

Fig 10.23 Next fit packing for 0.5, 2/N, 0.5, 2/N,⋯

First Fit

- To scan the bins in order and place the new item in the first bin that is large enough to hold it.
- A new bin is created only when the results of previous placements have left no other alternative.
- Processing each item by scanning down the list of bins sequentially, which would take $O(N^2)$

First Fit

- Items with sizes 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

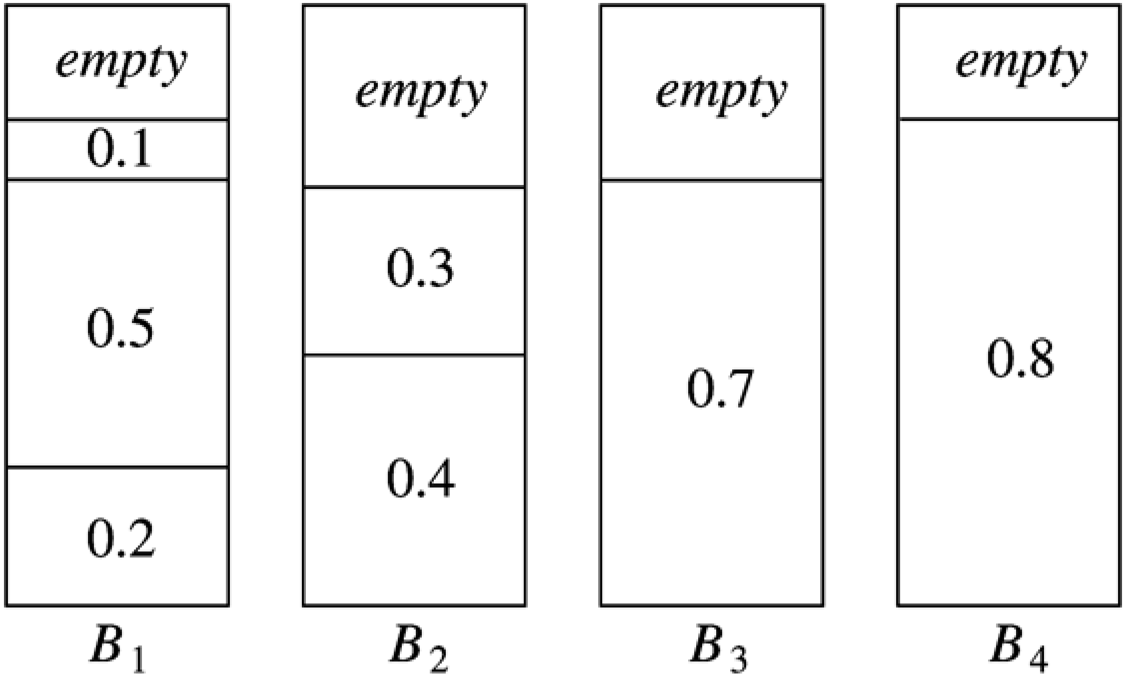


Fig 10.24 Using First fit (4 bins)

First Fit

(Theorem 10.3)

Let M be the optimal number of bins required to pack a list I of items. Then first fit never uses more than $[{}^{17}/_{10} * N]$ bins

(Ex) The input consists of 6M items of size 1/7+e, followed by 6M items of size 1/3+e, followed by 6M items of size ½+e. One simple packing places one item of each size in a bin and requires 6M bins.

First Fit: Worst Case

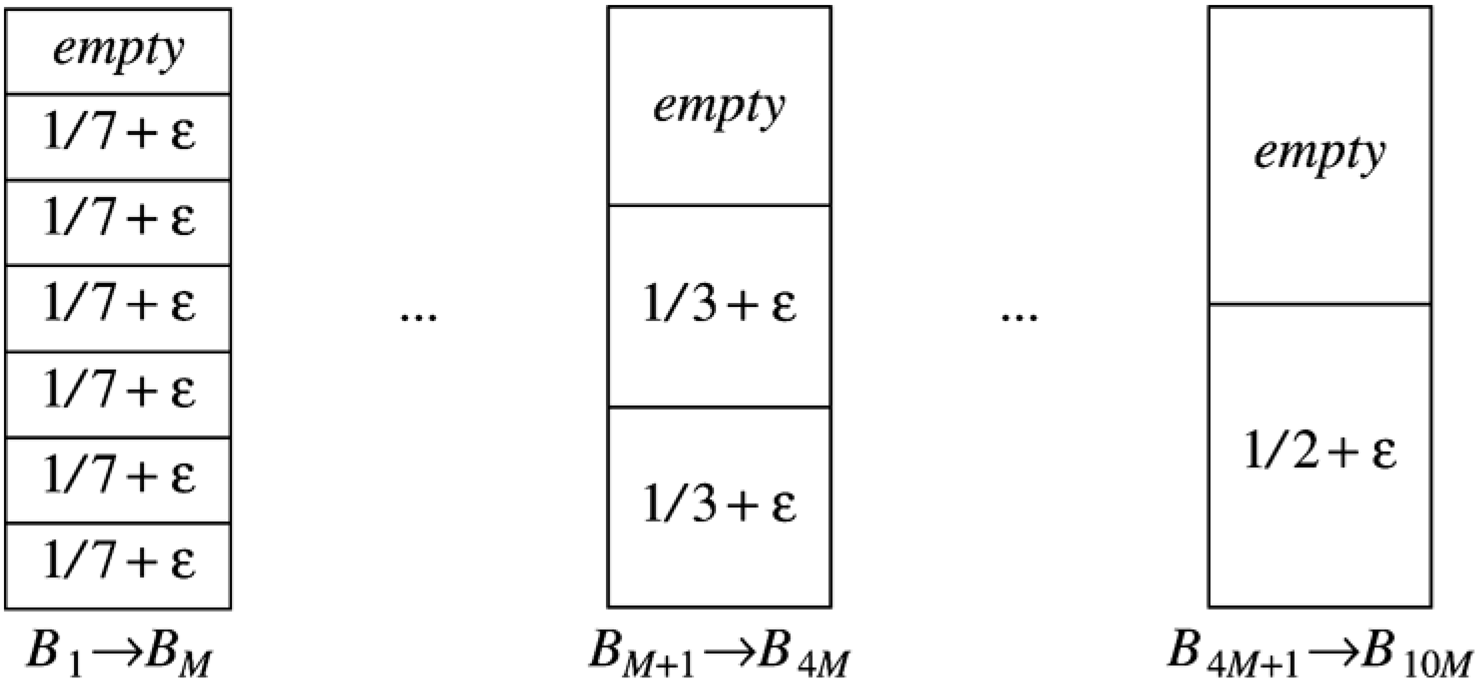


Fig 10.25 A case where FF uses 10M bins instead of 6M

Best Fit

- Instead of placing a new item in the fist spot that is found, it is placed in the tightest spot among all bins.
- Even though we make a more educated choice of bins, the generic bad cases are the same
- Best fit is never more than roughly 1.7 times as bad as optimal.

Best Fit

- Items with sizes 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

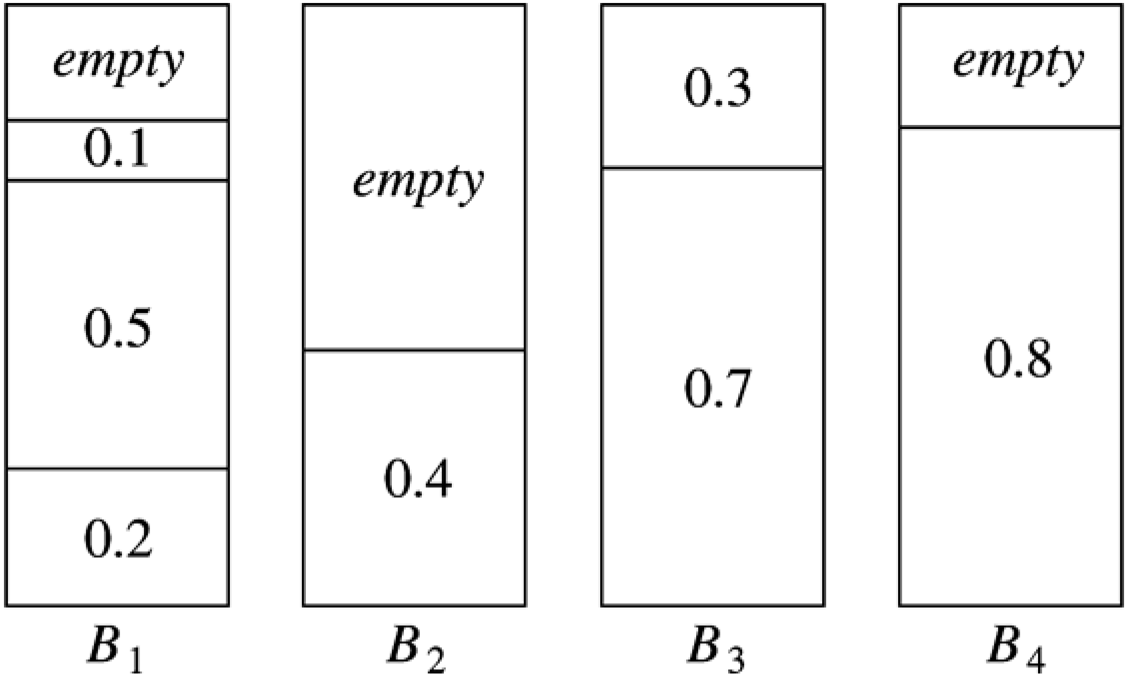


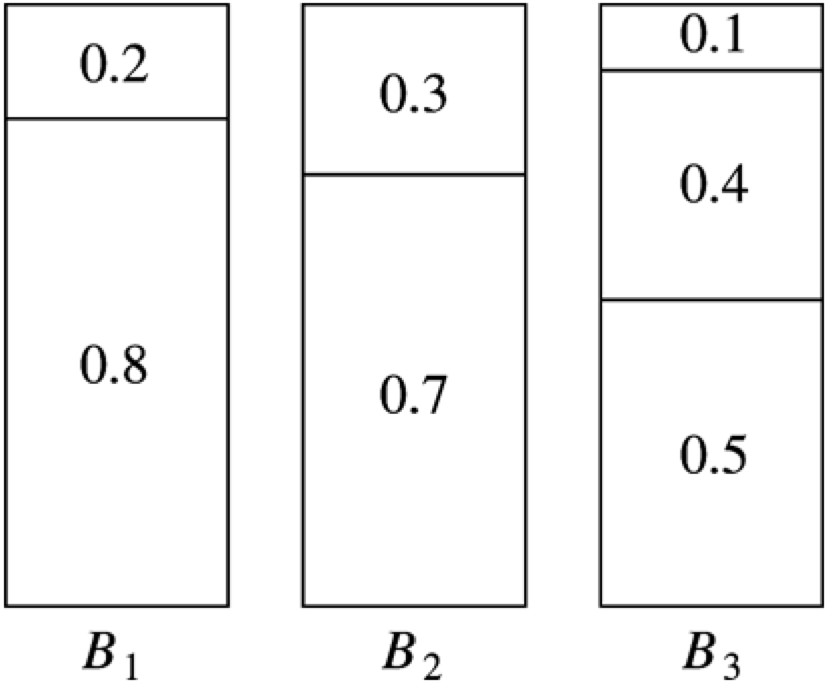
Fig 10.26 Best Fit

Off-line Algorithms

- Can view the entire item list before producing an answer.
- All the on-line algorithms have difficulty in packing the large items, especially when they occur later in the input.
- This can be solved by sorting the items and placing the largest items first.
- We can then apply first fit or best fit, yielding <u>first fit decreasing</u> and best fit decreasing, respectively.

First Fit Decreasing

- Items with sizes 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1

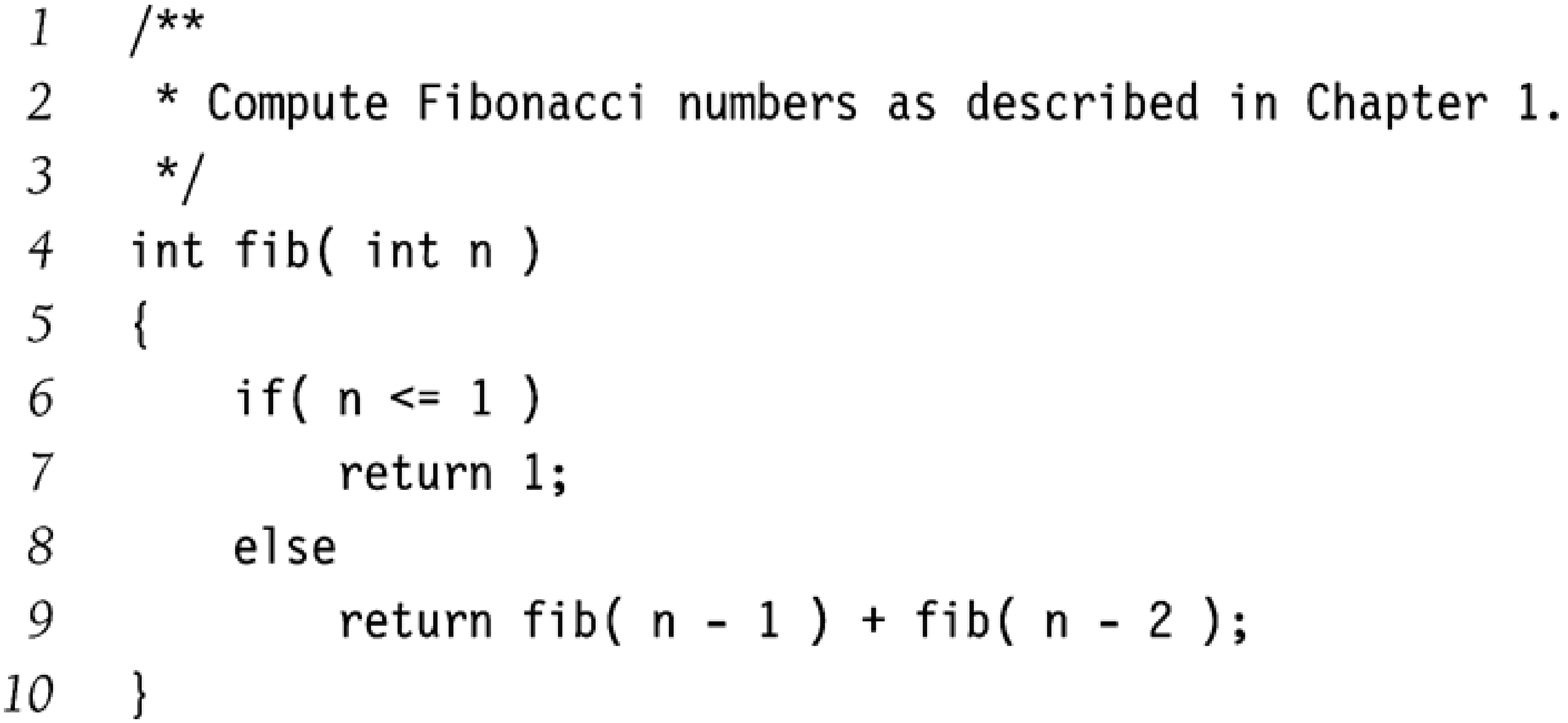


- Optimal, but not true in general.

Dynamic Programming

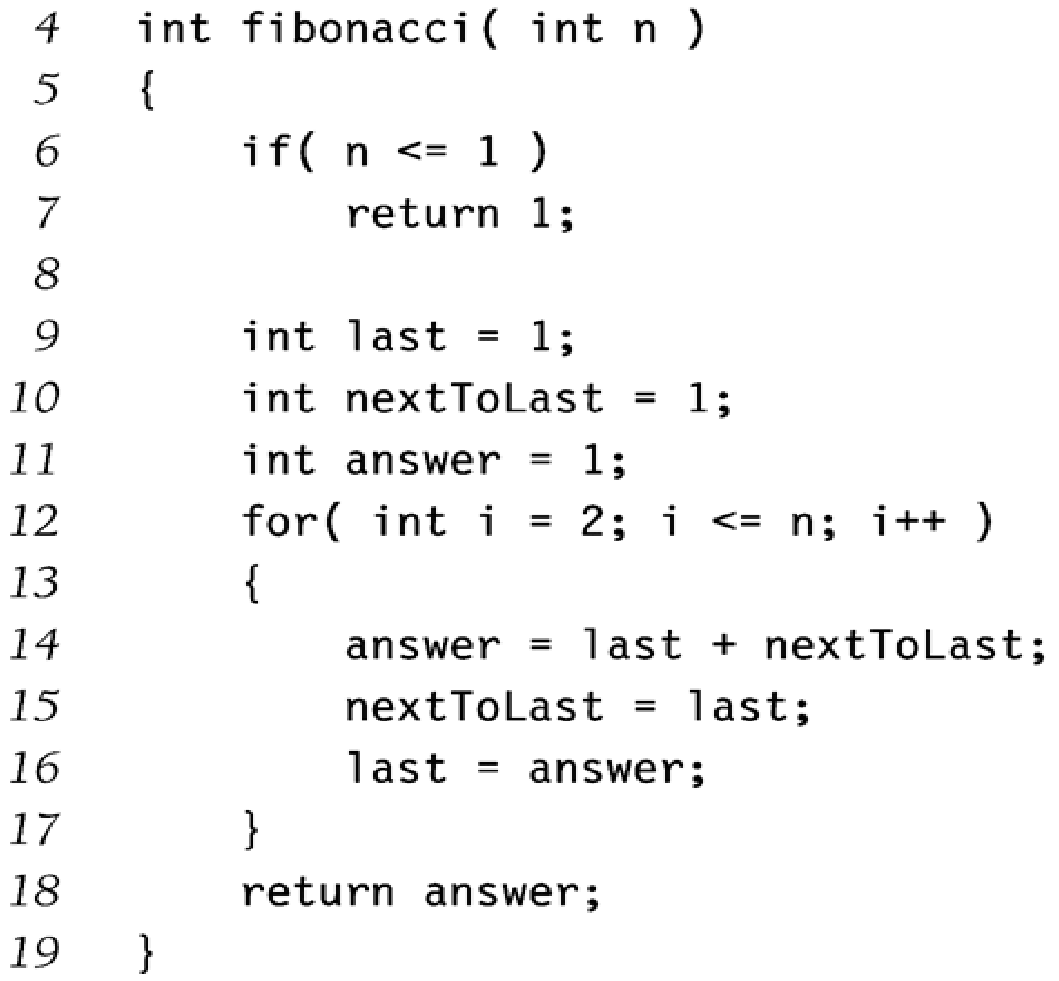
- A problem that can be mathematically expressed recursively can also be expressed as a recursive algorithm.
- In case a recursive algorithm is not efficient, the recursive algorithm can be rewritten as a non-recursive algorithm that systematically records the answers to the subproblems in a table.
- Dynamic programming makes use of this approach.

Inefficient Fibonacci Algorithm



```
/**
 * Compute Fibonacci numbers as described in Chapter 1.
 */
int fib( int n )
{
    if( n <= 1 )
        return 1;
    else
        return fib( n - 1 ) + fib( n - 2 );
}
```

Linear Fibonacci Algorithm

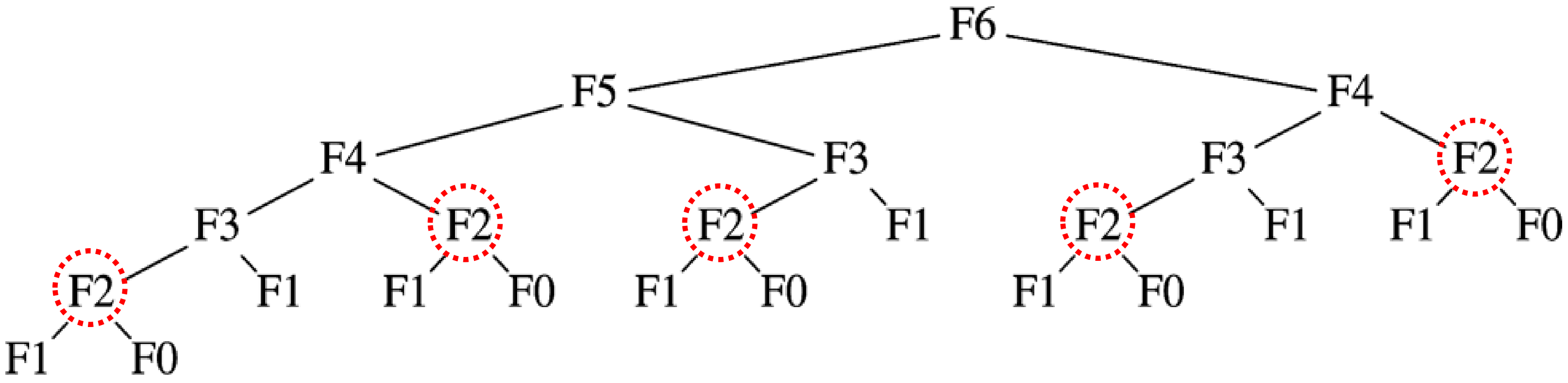


```
int fibonacci( int n )
{
    if( n <= 1 )
        return 1;

    int last = 1;
    int nextToLast = 1;
    int answer = 1;
    for( int i = 2; i <= n; i++ )
    {
        answer = last + nextToLast;
        nextToLast = last;
        last = answer;
    }
    return answer;
}
```

Recursive algorithm

- The recursive algorithm is slow due to repeated function calls



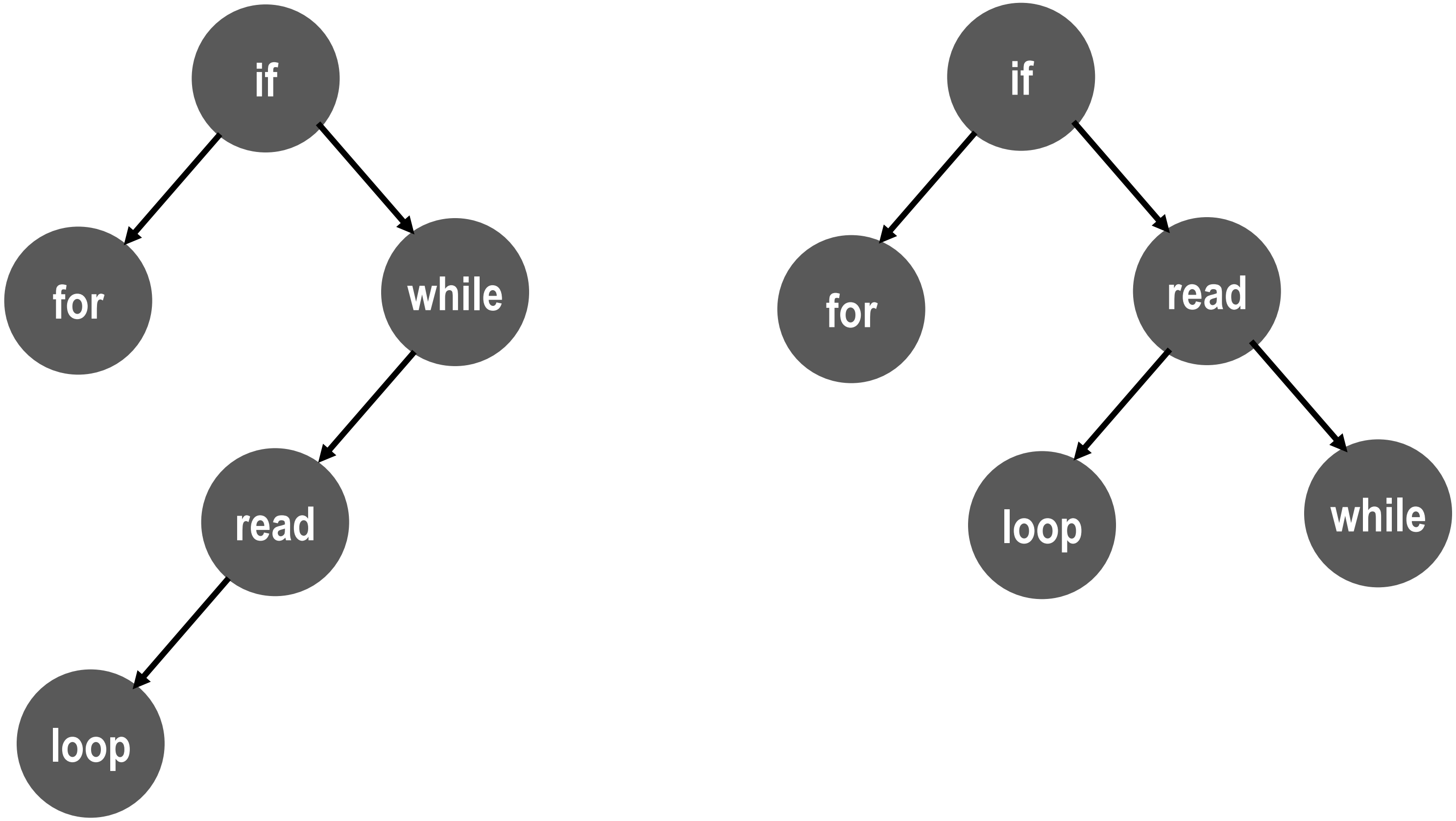
F_{N-3} 3 times / F_{N-4} 5 times / F_{N-5} 8 times

Binary Search Tree

- A binary search tree T is a binary tree; either it is empty or each node in the tree contains an identifier and:

  1. all identifiers in the left subtree of T are less than the identifier in the root node T;
  2. all identifiers in the right subtree of T are greater than the identifier in the root node T;
  3. the left and right subtrees of T are also binary search trees.

Two Examples



Which one is more desirable in terms of search?

Algorithm

```
procedure  SEARCH(T, X, $i$) {
// search binary search tree T for X
    $i \leftarrow$ T;
    while $i \neq 0$ do {
       case {
           :X < IDENT($i$): k ← LCHILD($i$) //search left tree
           :X = IDENT($i$): return
           :X > IDENT($i$): $i \leftarrow$ RCHILD($i$) //search right tree
        }
    }
}
```

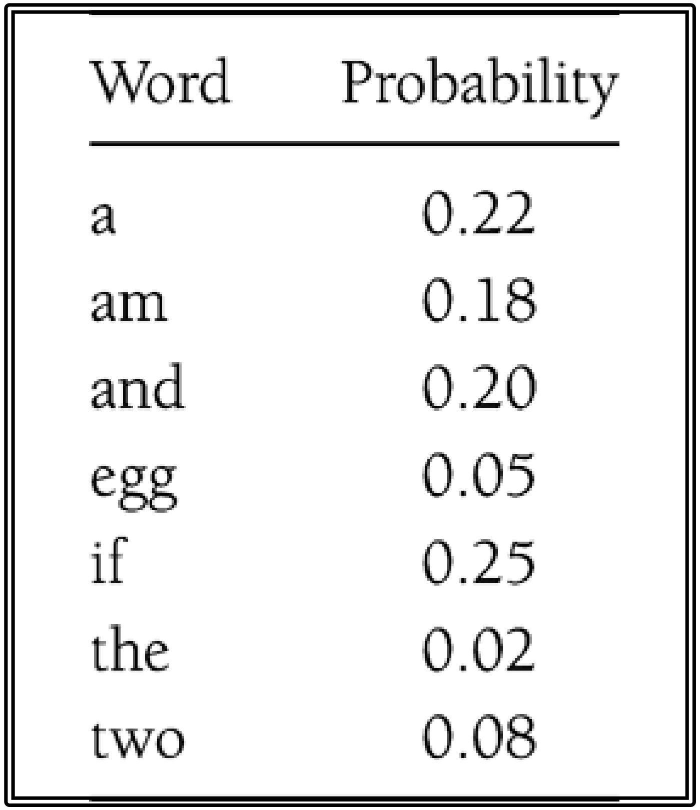
Optimal Binary Search Tree

- Input: a list of words, $w_1, w_2, \cdots, w_N$, and fixed probabilities $p_1, p_2, \cdots, p_N$ of their occurrence.
- Output: A binary search tree that minimizes the expected total access time or total number of comparisons required.
- Hence, the tree should minimize

$$T = \sum_{1}^{n} p_i * (1 + d_i)$$

where d_i is the depth of word w_i in the tree

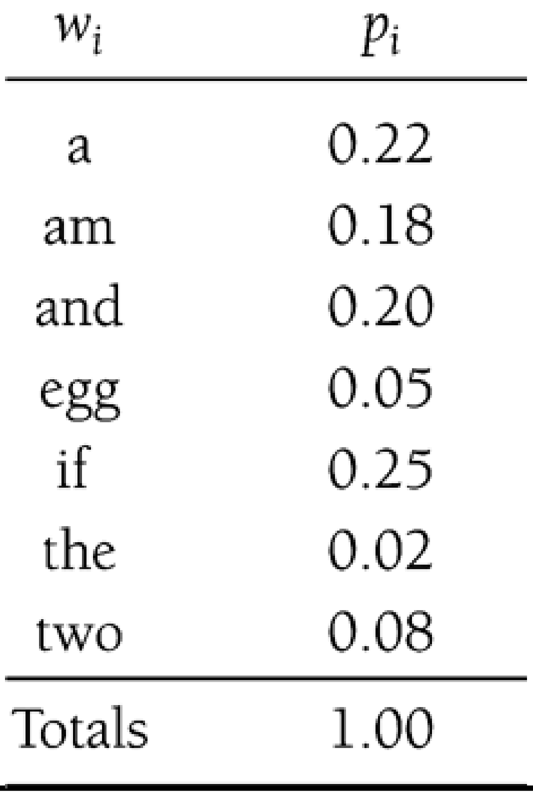
Sample Input



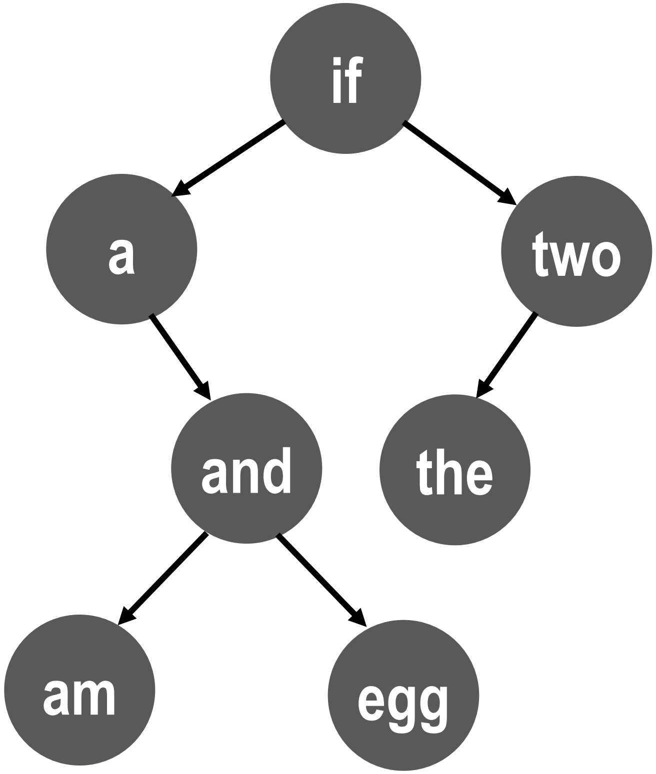
Word	Probability
a	0.22
am	0.18
and	0.20
egg	0.05
if	0.25
the	0.02
two	0.08

Possible BST #1

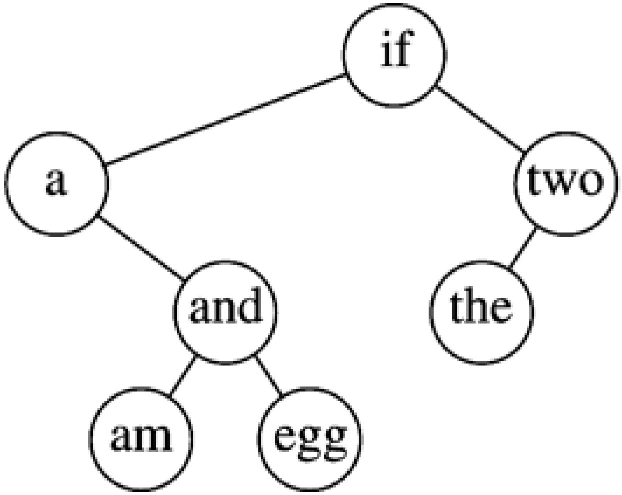
- Use a greedy approach where the word with the highest probability was placed at the root.



w_i	p_i
a	0.22
am	0.18
and	0.20
egg	0.05
if	0.25
the	0.02
two	0.08
Totals	1.00



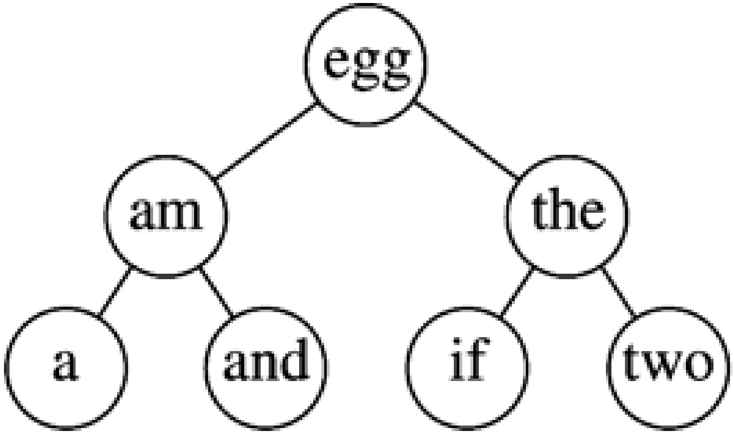
Possible BST #1



Input		Tree #1	
Word	Probability	Access Cost	
w_i	p_i	Once	Sequence
a	0.22	2	0.44
am	0.18	4	0.72
and	0.20	3	0.60
egg	0.05	4	0.20
if	0.25	1	0.25
the	0.02	3	0.06
two	0.08	2	0.16
Totals	1.00		2.43

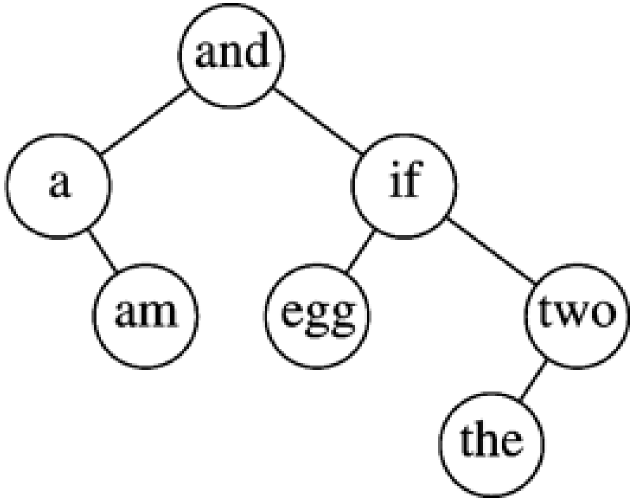
Possible BST #2

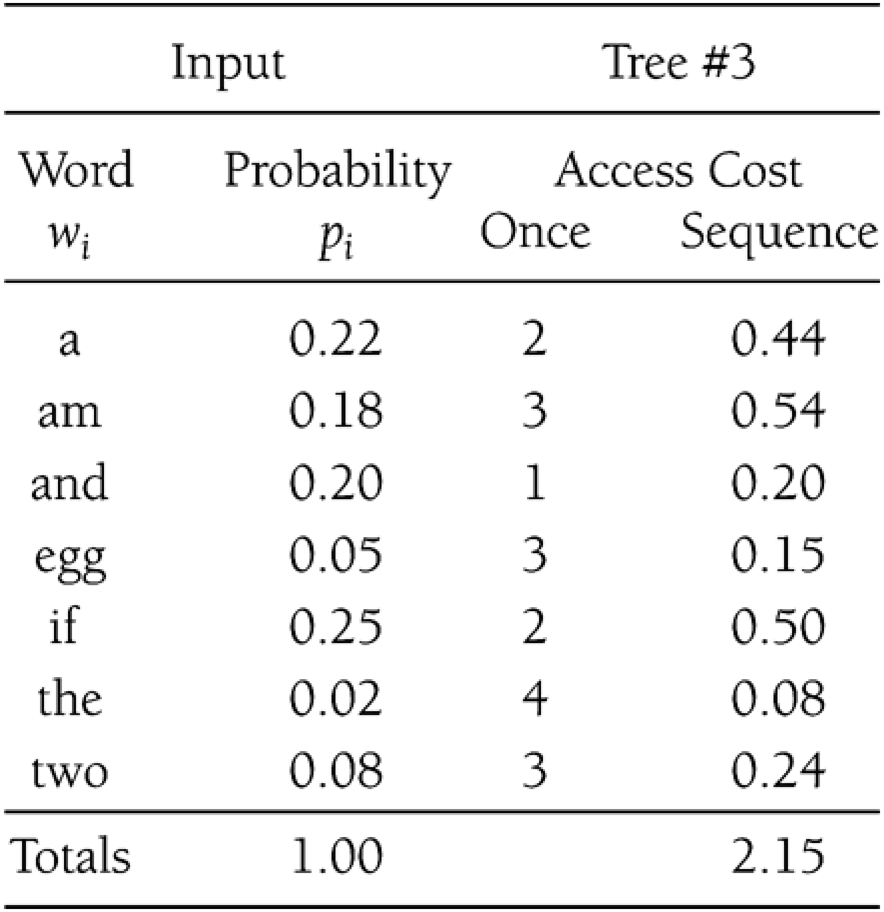
- Perfectly balanced search tree.



Input		Tree #2	
Word	Probability	Access Cost	
w_i	p_i	Once	Sequence
a	0.22	3	0.66
am	0.18	2	0.36
and	0.20	3	0.60
egg	0.05	1	0.05
if	0.25	3	0.75
the	0.02	2	0.04
two	0.08	3	0.24
Totals	1.00		2.70

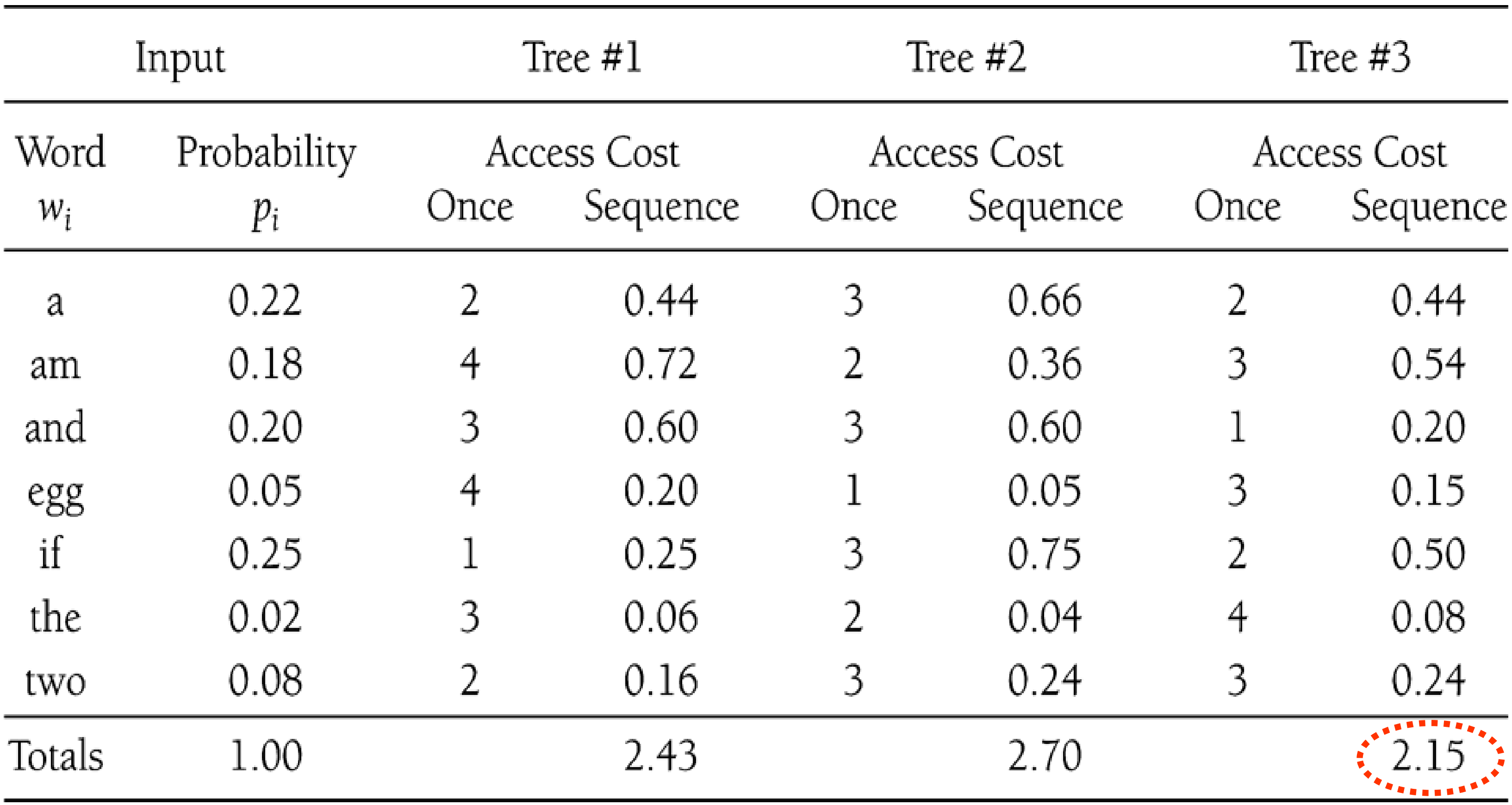
Possible BST #3





Input		Tree #3	
Word w_i	Probability p_i	Access Cost Once	 Sequence
a	0.22	2	0.44
am	0.18	3	0.54
and	0.20	1	0.20
egg	0.05	3	0.15
if	0.25	2	0.50
the	0.02	4	0.08
two	0.08	3	0.24
Totals	1.00		2.15

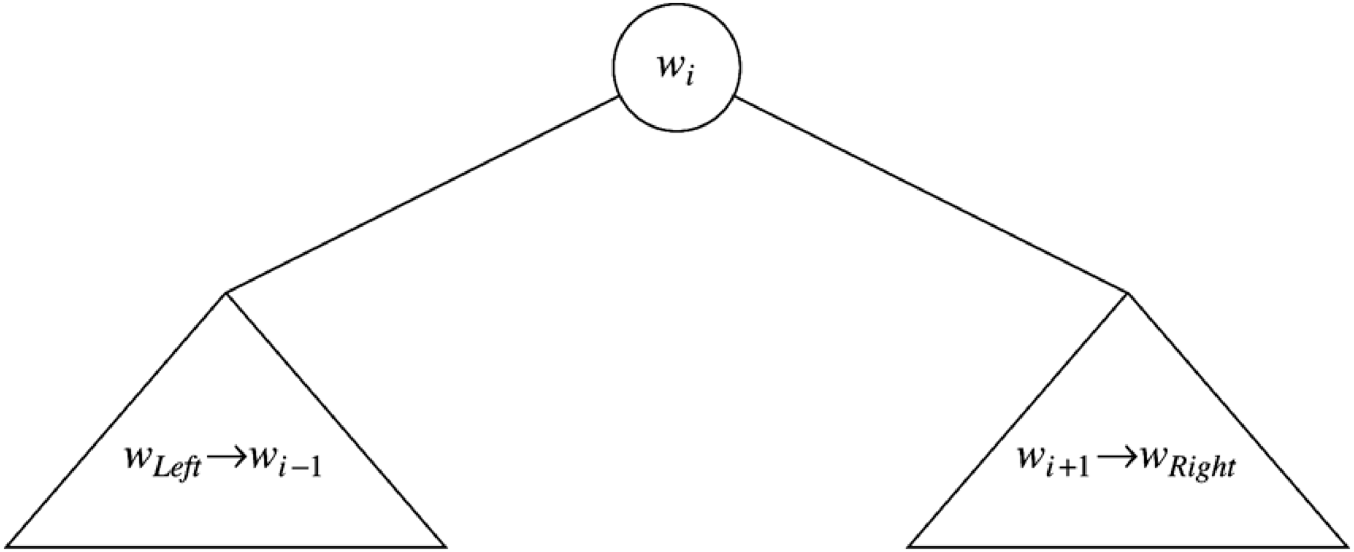
Comparison



Input		Tree #1		Tree #2		Tree #3	
Word	Probability	Access Cost		Access Cost		Access Cost	
w_i	p_i	Once	Sequence	Once	Sequence	Once	Sequence
a	0.22	2	0.44	3	0.66	2	0.44
am	0.18	4	0.72	2	0.36	3	0.54
and	0.20	3	0.60	3	0.60	1	0.20
egg	0.05	4	0.20	1	0.05	3	0.15
if	0.25	1	0.25	3	0.75	2	0.50
the	0.02	3	0.06	2	0.04	4	0.08
two	0.08	2	0.16	3	0.24	3	0.24
Totals	1.00		2.43		2.70		2.15

Structure of Optimal BST

- Place sorted words w_{Left}, $w_{Left+1}, \cdots, w_i, \cdots, w_{Right-1}, w_{Right}$ into a binary search tree.



Cost Formula

$$C_{Left,\ Right} = \min_{Left \le i \le Right} \{\ p_i + C_{Left,\ i-1} + C_{i+1,\ Right} + \sum_{j=Left}^{i-1} p_j + \sum_{j=i+1}^{Right} p_j\ \}$$

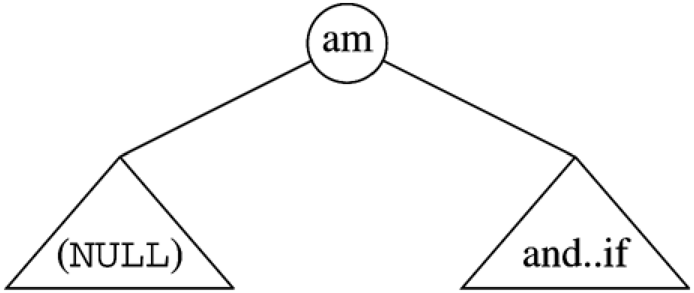
$$= \min_{Left \le i \le Right} \{\ C_{Left,\ i-1} + C_{i+1,\ Right} + \sum_{j=Left}^{Right} p_j\ \}$$

- For each subrange of words starting from a single word, the algorithm produces the cost and root of the optimal BST as in the following table.

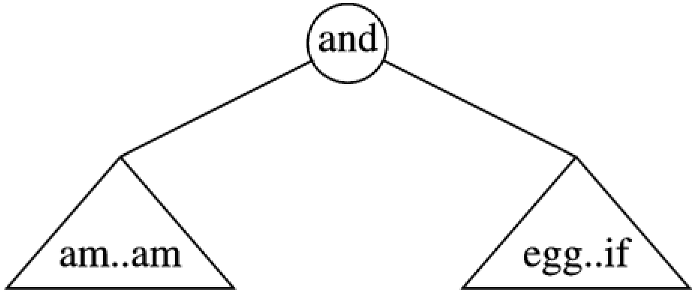
Computation for the sample input

	Left=1	Left=2	Left=3	Left=4	Left=5	Left=6	Left=7
Iteration=1	a..a .22 a	am..am .18 am	and..and .20 and	egg..egg .05 egg	if..if .25 if	the..the .02 the	two..two .08 two
Iteration=2	a..am .58 a	am..and .56 and	and..egg .30 and	egg..if .35 if	if..the .29 if	the..two .12 two	
Iteration=3	a..and 1.02 am	am..egg .66 and	and..if .80 if	egg..the .39 if	if..two .47 if		
Iteration=4	a..egg 1.17 am	am..if 1.21 and	and..the .84 if	egg..two .57 if			
Iteration=5	a..if 1.83 and	am..the 1.27 and	and..two 1.02 if				
Iteration=6	a..the 1.89 and	am..two 1.53 and					
Iteration=7	a..two 2.15 and						

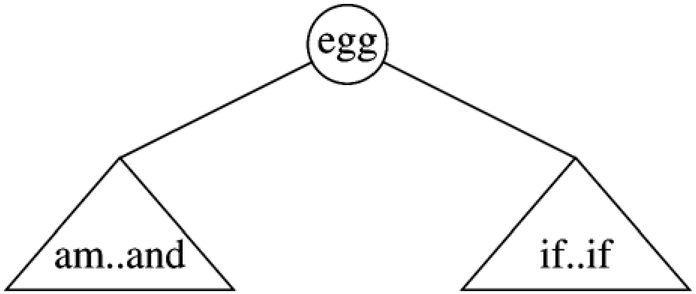
Computation of table entry for am..if



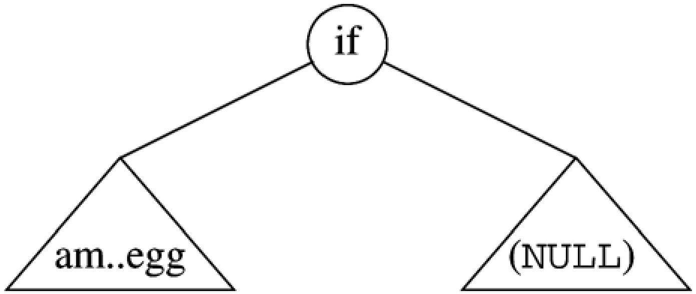
0 + 0.80 + 0.68 = 1.48



0.18 + 0.35 + 0.68 = 1.21



0.56 + 0.25 + 0.68 = 1.49



0.66 + 0 + 0.68 = 1.34