# DATA STRUCTURE AND ALGORITHM

Dr. Khine Thin Zar
Professor
Computer Engineering and Information Technology Dept.
Yangon Technological University

# Lecture 13

## Algorithm Design Techniques

# Outlines of Class (Lecture 13)

- Introduction
- Greedy Algorithms
- A Simple Scheduling Problem
- Huffman Codes
- Approximate Bin Packing
- Divide and Conquer
- Dynamic Programming
- Backtracking Algorithms

# Introduction

- The efficient implementation of algorithms
- Up to the programmer to choose the appropriate data structure in order to make the running time as small as possible
- Focus on five of the common types of algorithms used to solve problems.
- For each type of algorithm we will . . .
  - See the general approach,
  - Look at several examples,
  - Discuss, in general terms, the time and space complexity

# Greedy Algorithms

- Dijkstra's, Prim's, and Kruskal's algorithms
- Work in phases
- A decision is made that appears to be good, without regard for future consequences
- When the algorithm terminates, the local optimum is equal to the global optimum.
- The coin changing problem, Traffic problems

# A Simple Scheduling Problem

- First application of greedy algorithms
- Given jobs j1, j2, . . . , jN,
- With known running times t1, t2, . . . , tN
- A single processor
- To schedule these jobs in order to minimize the average completion time
- Assume nonpreemptive scheduling: Once a job is started, it must run to completion.

# A Simple Scheduling Problem (Cont.)

| Job | Time |
| --- | --- |
| $j_1$ | 15 |
| $j_2$ | 8 |
| $j_3$ | 3 |
| $j_4$ | 10 |

**Figure:** Jobs and times

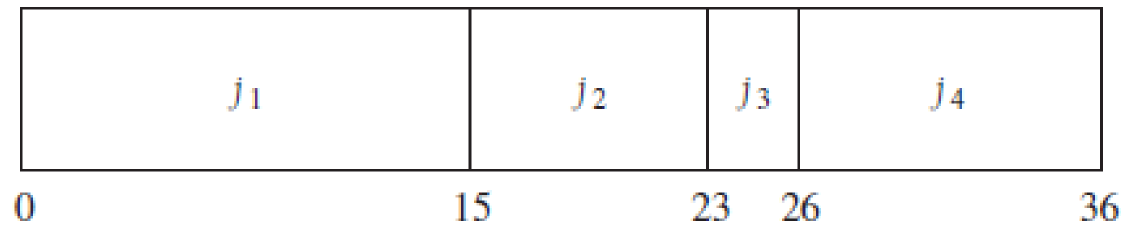


**Figure:** Schedule #1

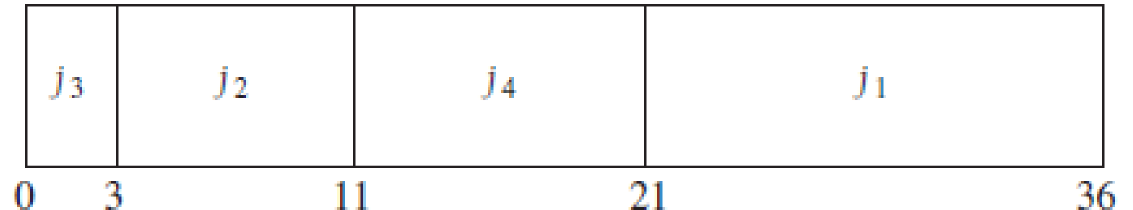


**Figure:** Schedule #2 (optimal)

the total cost, $C$, of the schedule is:

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

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

# Huffman Codes

- Second application of greedy algorithms: file compression.
- ASCII character set consists of roughly 100 “printable” characters
- log 100 = 7 bits are required
- Seven bits allow the representation of 128 characters
- In real life, a big disparity between the most frequent and least frequent characters
- Reducing the file size
- To provide a better code and reduce the total number of bits required.

# Huffman Codes (Cont.)

| 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 | | | 174 |

Figure: Example using a standard coding scheme

# Huffman Codes (Cont.)

- to allow the code length to vary from character to character
- to ensure that the frequently occurring characters have short codes.

# Huffman's Algorithm

- Huffman's algorithm can be described as follows:
- The weight of a tree is equal to the sum of the frequencies of its leaves. C−1 times, select the two trees, T1 and T2, of smallest weight, breaking ties arbitrarily, and form a new tree with subtrees T1 and T2.
- At the beginning of the algorithm, there are C single-node trees—one for each character.
- At the end of the algorithm there is one tree, and this is the optimal Huffman coding tree.

# Huffman's Algorithm (Cont.)

**Figure:** Initial stage of Huffman's algorithm

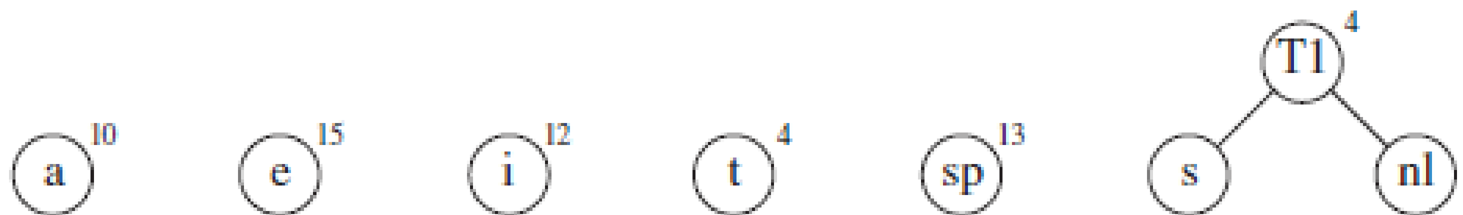


**Figure:** Huffman's algorithm after the first merge

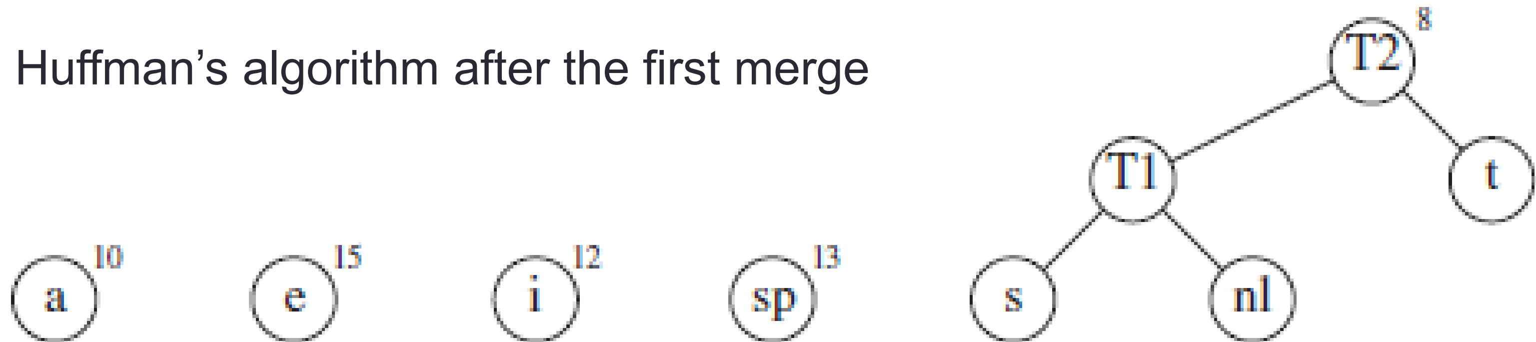


**Figure:** Huffman's algorithm after the second merge

# Huffman’s Algorithm (Cont.)

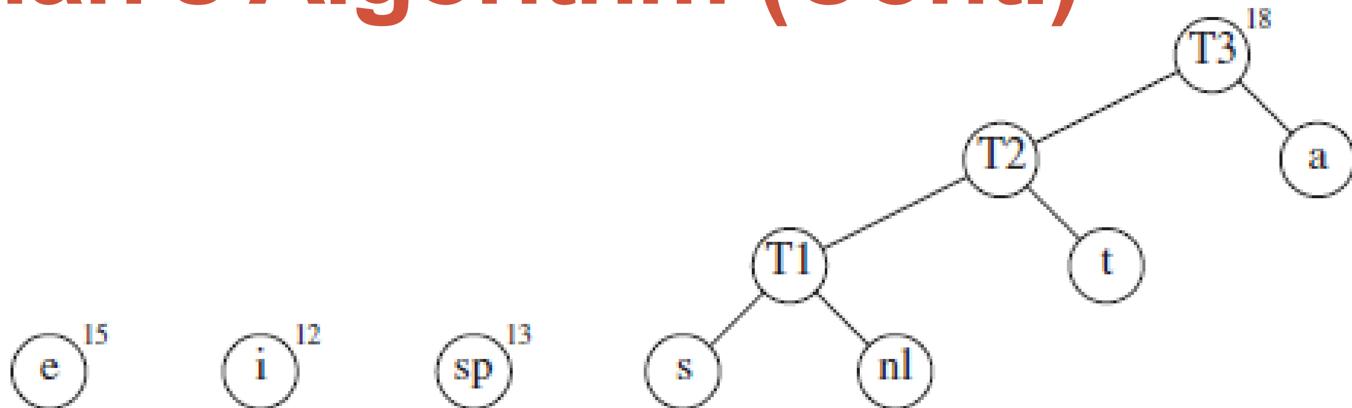


**Figure:** Huffman’s algorithm after the third merge

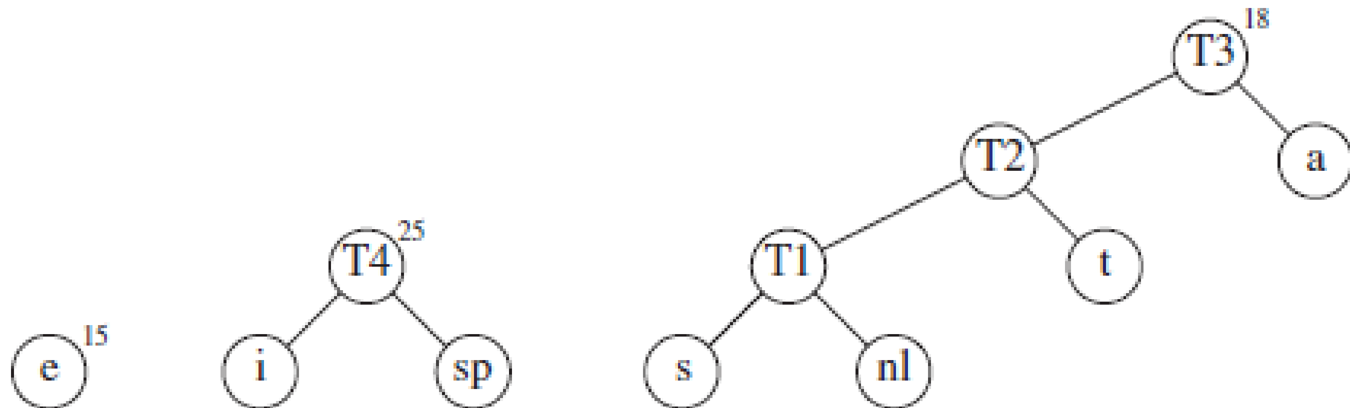


**Figure:** Huffman’s algorithm after the fourth merge

# Huffman's Algorithm (Cont.)

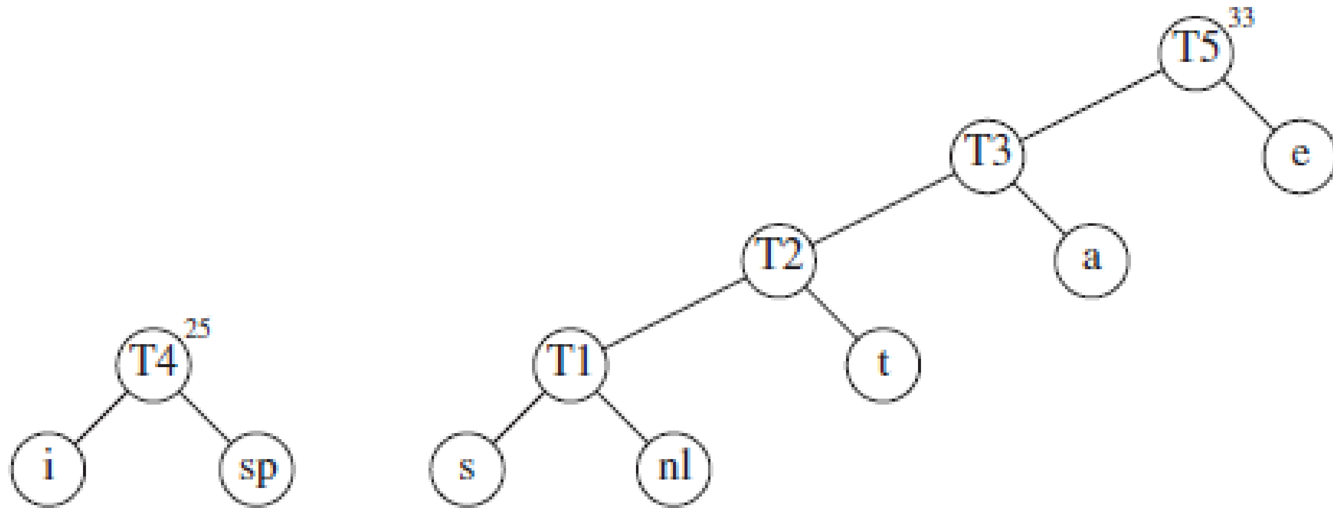


**Figure:** Huffman's algorithm after the fifth merge

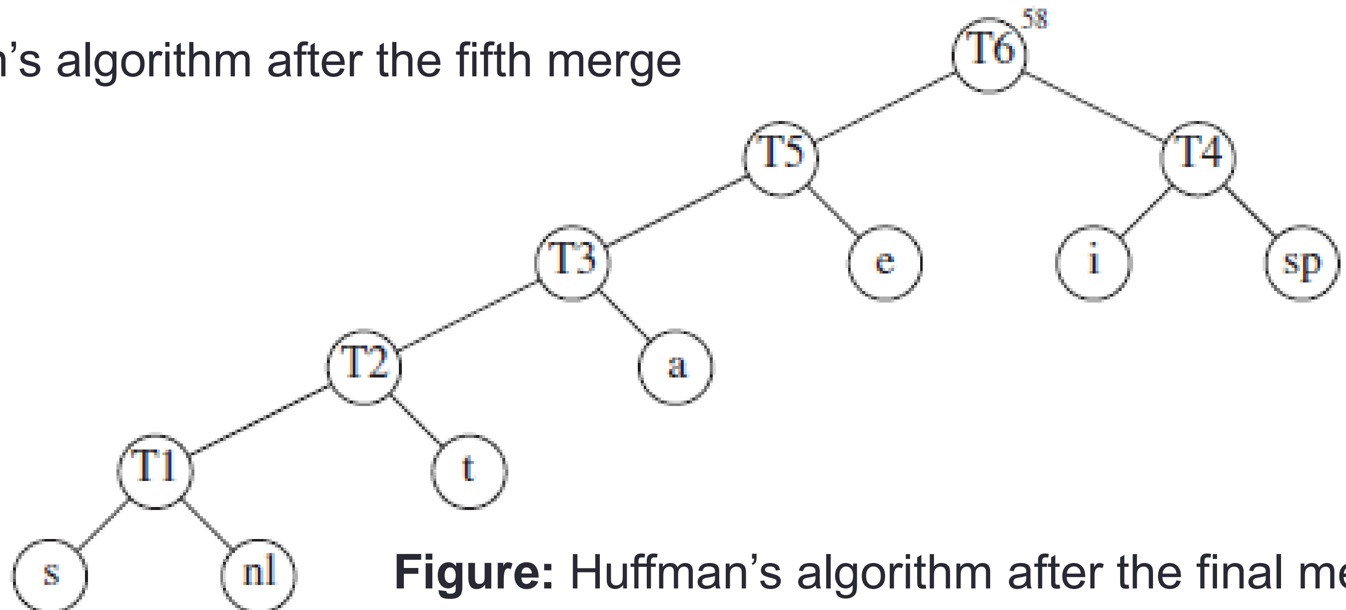


**Figure:** Huffman's algorithm after the final merge

# Approximate Bin Packing

- Third application of greedy algorithms: bin-packing problem
- The solutions that are produced are not too far from optimal
- Given N items of sizes s1, s2, . . . , sN.
- All sizes satisfy $0 < s_i \leq 1$.
- To pack these items in the fewest number of bins (each bin has unit capacity)

# Approximate Bin Packing (Cont.)

- Two versions of the bin packing problem
- Online bin packing:
  - ✓ each item must be placed in a bin before the next item can be processed.
- Offline bin packing:
  - ✓ do not need to do anything until all the input has been read.

# Approximate Bin Packing (Cont.)

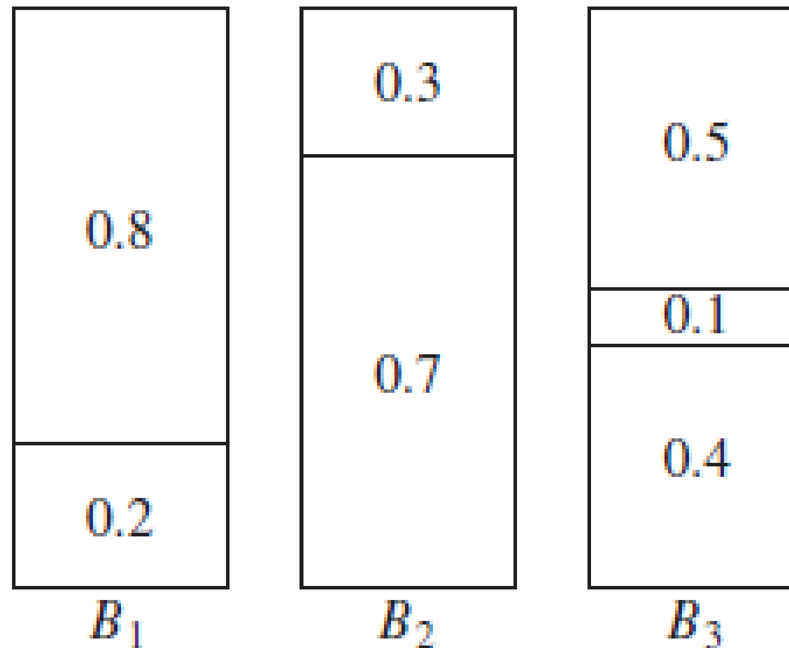


**Figure:** Optimal packing for 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8

# Divide and Conquer

- Divide-and conquer algorithms consist of two parts:
- Divide: Smaller problems are solved recursively (except, of course, base cases).
- Conquer: The solution to the original problem is then formed from the solutions to the subproblems.
- Computational geometry: Given N points in a plane, show that the closest pair of points can be found in O(N logN) time.
- The selection problem: to find the kth smallest element in a collection S of N elements.

# Dynamic Programming

- Any recursive mathematical formula could be directly translated to a recursive algorithm
- The compiler will not do justice to the recursive algorithm, and an inefficient program results.
- Help to the compiler, by rewriting the recursive algorithm as a nonrecursive algorithm

# Using a Table Instead of Recursion

- running time, T(N), that satisfies T(N) ≥ T(N−1)+T(N−2).

```
1   /**
2    * Compute Fibonacci numbers as described in Chapter 1.
3    */
4   long long fib( int n )
5   {
6       if( n <= 1 )
7           return 1;
8       else
9           return fib( n - 1 ) + fib( n - 2 );
10  }
```

**Figure:** Inefficient algorithm to compute Fibonacci numbers

# Using a Table Instead of Recursion (Cont.)

```
1   /**
2    * Compute Fibonacci numbers as described in Chapter 1.
3    */
4   long long fibonacci( int n )
5   {
6       if( n <= 1 )
7           return 1;
8
9       long long last = 1;
10      long long last nextToLast = 1;
11      long long answer = 1;
12
13      for( int i = 2; i <= n; ++i )
14      {
15          answer = last + nextToLast;
16          nextToLast = last;
17          last = answer;
18      }
19      return answer;
20  }
```

**Figure:** Linear algorithm to compute Fibonacci numbers (more efficient)

# Using a Table Instead of Recursion

- the recursive algorithm is so slow because of the algorithm used to simulate recursion.
- To compute $F_N$, there is one call to $F_{N-1}$ and $F_{N-2}$.
- $F_{N-3}$ is computed three times, $F_{N-4}$ is computed five times, $F_{N-5}$ is computed eight times, and so on.

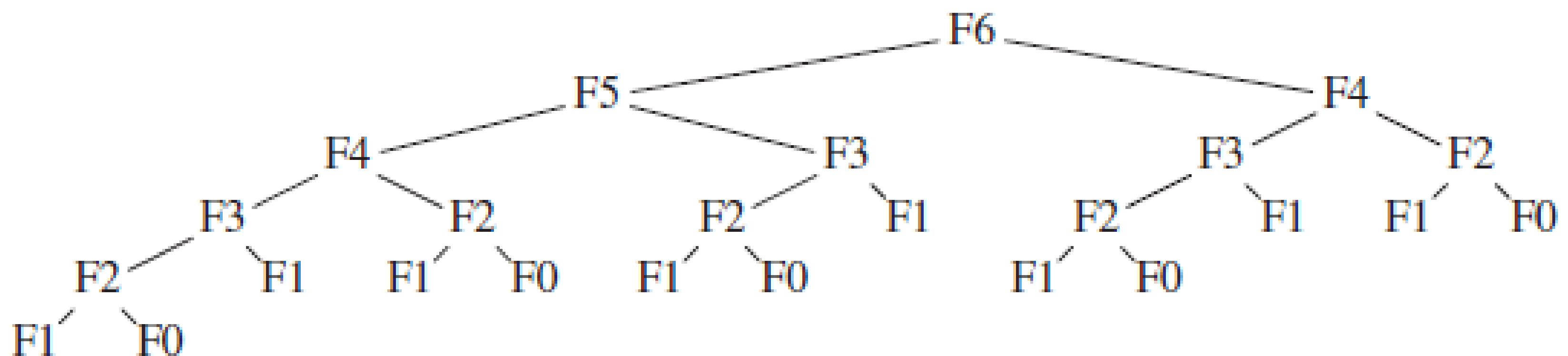


**Figure:** Trace of the recursive calculation of Fibonacci numbers

# Ordering Matrix Multiplications

- Given four matrices, A, B, C, and D, of dimensions A = 50×10, B = 10×40, C = 40×30, and D = 30×5.
- What is the best way to perform the three matrix multiplications required to compute ABCD?
- To solve the problem by exhaustive search
- There are only five ways to order the multiplications
- The best ordering uses roughly one-ninth the number of multiplications as the worst ordering

# Ordering Matrix Multiplications (Cont.)

- **(A((BC)D))**: Evaluating **BC** requires 10 × 40 × 30 = 12,000 multiplications. Evaluating **(BC)D** requires the 12,000 multiplications to compute **BC**, plus an additional 10 × 30 × 5 = 1,500 multiplications, for a total of 13,500. Evaluating **(A((BC)D))** requires 13,500 multiplications for **(BC)D**, plus an additional 50 × 10 × 5 = 2,500 multiplications, for a grand total of 16,000 multiplications.
- **(A(B(CD)))**: Evaluating **CD** requires 40×30×5 = 6,000 multiplications. Evaluating **B(CD)** requires the 6,000 multiplications to compute **CD**, plus an additional 10×40× 5 = 2,000 multiplications, for a total of 8,000. Evaluating **(A(B(CD)))** requires 8,000 multiplications for **B(CD)**, plus an additional 50 × 10 × 5 = 2,500 multiplications, for a grand total of 10,500 multiplications.
- **((AB)(CD))**: Evaluating **CD** requires 40×30×5 = 6,000 multiplications. Evaluating **AB** requires 50 × 10 × 40 = 20,000 multiplications. Evaluating **((AB)(CD))** requires 6,000 multiplications for **CD**, 20,000 multiplications for **AB**, plus an additional 50 × 40 × 5 = 10,000 multiplications for a grand total of 36,000 multiplications.

# Ordering Matrix Multiplications (Cont.)

- **(((AB)C)D)**: Evaluating **AB** requires 50 × 10 × 40 = 20,000 multiplications. Evaluating **(AB)C** requires the 20,000 multiplications to compute **AB**, plus an additional 50 × 40 × 30 = 60,000 multiplications, for a total of 80,000. Evaluating **(((AB)C)D)** requires 80,000 multiplications for **(AB)C**, plus an additional 50 × 30 × 5 = 7,500 multiplications, for a grand total of 87,500 multiplications.

- **((A(BC))D)**: Evaluating **BC** requires 10 × 40 × 30 = 12,000 multiplications. Evaluating **A(BC)** requires the 12,000 multiplications to compute **BC**, plus an additional 50 × 10 × 30 = 15,000 multiplications, for a total of 27,000. Evaluating **((A(BC))D)** requires 27,000 multiplications for **A(BC)**, plus an additional 50 × 30 × 5 = 7,500 multiplications, for a grand total of 34,500 multiplications.

# Optimal Binary Search Tree

- A list of words, w1, w2, . . . , wN, and fixed probabilities, p1, p2, . . . , pN, of their occurrence
- To arrange these words in a binary search tree in a way that minimizes the expected total access time.
- The number of comparisons needed to access an element at depth d is d + 1

# Optimal Binary Search Tree (Cont.)

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

**Figure:** Sample input for optimal binary search tree problem

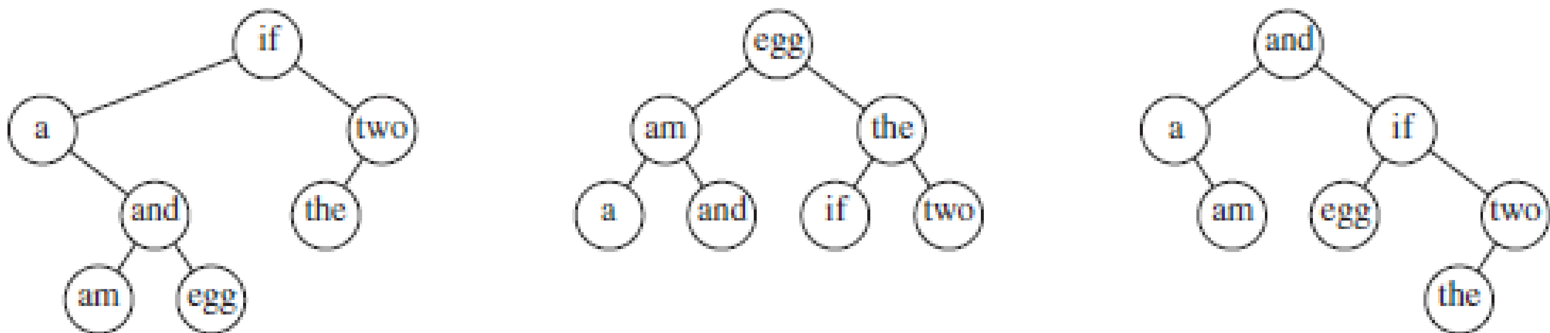


**Figure:** Three possible binary search trees for data in above table

# Optimal Binary Search Tree (Cont.)

| 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 |

**Figure:** Comparison of the three binary search trees

# Backtracking Algorithms

- ❑ Amounts to a clever implementation of exhaustive search, with generally unfavorable performance.
- ❑ The savings over a brute-force exhaustive search can be significant
- ❑ The problem of arranging furniture in a new house
  - ✓ Many possibilities to try
  - ✓ Many other bad arrangements are discarded early, because an undesirable subset of the arrangement is detected.
  - ✓ The elimination of a large group of possibilities in one step is known as pruning.

# Next Week Lecture (Week 14)

Lecture 14: Applications

Thank you!