# 22. Dynamic Programming III

Optimal Search Tree [Ottman/Widmayer, Kap. 5.7]

## 22.1 Optimal Search Trees

# Optimal binary Search Trees

**Given**: $n$ keys $k_1, k_2 \dots k_n$ (wlog $k_1 < k_2 <$ ... $< k_n$) with weights (search probabilities[34]) $p_1, p_2, \dots, p_n$.

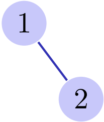
**Wanted**: optimal search tree $T$ with key depths[35] $\mathrm{d}(\cdot)$, that minimizes the expected search costs

$$C(T) = \sum_{i=1}^{n} (\mathrm{d}(k_i) + 1) \cdot p_i$$

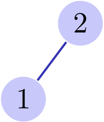
[34] It is possible to model unsuccesful search additionally, omitted for brevity here

[35] $d(k)$: Length of the path from the root to the node $k$

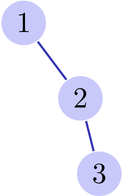
# Examples



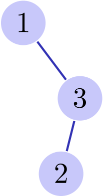

$2p_1 + p_2$



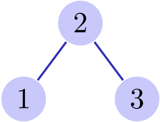

$p_1 + 2p_2$



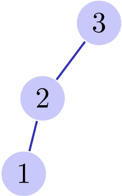

$p_1 + 2p_2 + 3p_3$



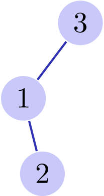

$p_1 + 3p_2 + 2p_2$




$2p_1 + p_2 + 2p_3$




$3p_1 + 2p_2 + p_3$

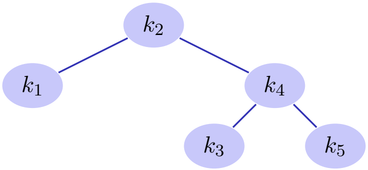



$2p_1 + 3p_3 + 1p_3$

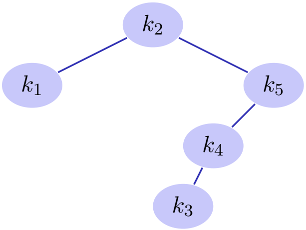
# Example

## Expected Frequencies

| $i$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $p_i$ | 0.25 | 0.10 | 0.05 | 0.20 | 0.40 |

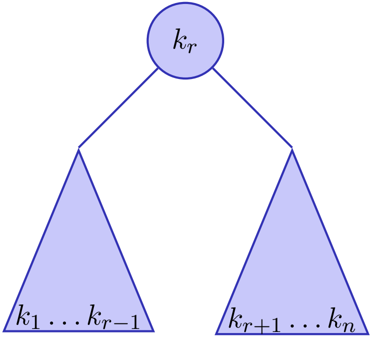



Search tree with expected costs 2.35

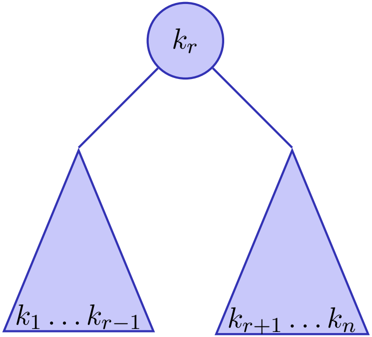



Search tree with expected costs 2.2

# Sub-trees for Searching



# Sub-trees for Searching



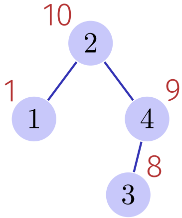

Which $r$ to choose?

# Greedy?

Scenario $p_1 = 1, p_2 = 10, p_3 = 8, p_4 = 9$

# Greedy?

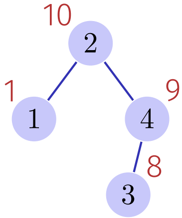
Scenario $p_1 = 1, p_2 = 10, p_3 = 8, p_4 = 9$



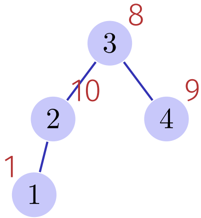

$c(T) = 54$

# Greedy?

Scenario $p_1 = 1, p_2 = 10, p_3 = 8, p_4 = 9$




$c(T) = 54$




$c(T) = 49$

# Structure of a optimal binary search tree

- Consider all subtrees with roots $k_r$ and optimal subtrees for keys $k_i, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j$
- Subtrees with keys $k_i, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j$ must be optimal for the respective sub-problems.[36]

$$E(i,j) = \text{Costs of optimal search tree with nodes } k_i, k_{i+1}, \ldots, k_j$$

---

[36]The usual argument: if it was not optimal, it could be replaced by a better solution improving the overal solution.

# Rekursion

With

$$p(i,j) := p_i + p_{i+1} + \cdots + p_j \qquad i \leq j$$

it holds that

$$E(i,j) = \begin{cases} 0 & \text{if } i > j \\ p(i) & \text{if } i = j \\ p(i,j) + \min\{E(i,k-1) + E(k+1,j), i \leq k \leq j\} & \text{otherwise.} \end{cases}$$

# DP

0. $E(1,n)$: Costs of optimal search tree with nodes $k_1,\ldots,k_n$ with search frequencies $p_1,\ldots,p_n$
1. $E(i,j), 1 \leq i \leq j \leq n$ — $\#$ sub-problems $\Theta(n^2)$
2. Enumerate roots of subtree of $k_i,\ldots,k_j$, $\#$ possibilities: $j-i+1$
3. Dependencies $E(i,j)$ depend on $E(i,k), E(k,j)$ $i<k<j$. Computation of the off-diagonals of $E$, starting with the diagonal of $E$
4. Solution is in $E(1,n)$, Reconstruction: store the arg-mins of the recursion in a separate table $V$.
5. Running time $\Theta(n^3)$. Memory $\Theta(n^2)$.

# Example

| $i$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $p_i$ | 0.25 | 0.10 | 0.05 | 0.20 | 0.40 |

$E$

| $i$ \ $j$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 1 | 0 | 0.25 | 0.45 | 0.60 | 1.15 | 2.00 |
| 2 | | 0 | 0.10 | 0.20 | 0.55 | 1.30 |
| 3 | | | 0 | 0.05 | 0.30 | 0.95 |
| 4 | | | | 0 | 0.20 | 0.80 |
| 5 | | | | | 0 | 0.40 |
| 6 | | | | | | 0 |

$p$

| $i$ \ $j$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0.25 | 0.35 | 0.40 | 0.60 | 1.00 |
| 2 | | 0.10 | 0.15 | 0.35 | 0.75 |
| 3 | | | 0.05 | 0.25 | 0.65 |
| 4 | | | | 0.20 | 0.60 |
| 5 | | | | | 0.40 |

$V$

| $i$ \ $j$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 4 |
| 2 | | 2 | 2 | 4 | 5 |
| 3 | | | 3 | 4 | 5 |
| 4 | | | | 4 | 5 |
| 5 | | | | | 5 |

# 23. Greedy Algorithms

Fractional Knapsack Problem, Huffman Coding [Cormen et al, Kap. 16.1, 16.3]

# Greedy Choice

A problem with a recursive solution can be solved with a **greedy algorithm** if it has the following properties:

- The problem has **optimal substructure**: the solution of a problem can be constructed with a combination of solutions of sub-problems.
- The problem has the **greedy choice property**: The solution to a problem can be constructed, by using a local criterion that is not depending on the solution of the sub-problems.

Examples: fractional knapsack, Huffman-Coding (below)
Counter-Example: knapsack problem, Optimal Binary Search Tree

# Huffman-Codes

Goal: memory-efficient saving of a sequence of characters using a binary code with code words..

# Huffman-Codes

Goal: memory-efficient saving of a sequence of characters using a binary code with code words..

## Example

File consisting of 100.000 characters from the alphabet $\{a, \ldots, f\}$.

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| Frequency (Thousands) | 45 | 13 | 12 | 16 | 9 | 5 |
| Code word with fix length | 000 | 001 | 010 | 011 | 100 | 101 |
| Code word variable length | 0 | 101 | 100 | 111 | 1101 | 1100 |

# Huffman-Codes

Goal: memory-efficient saving of a sequence of characters using a binary code with code words..

## Example

File consisting of 100.000 characters from the alphabet $\{a, \ldots, f\}$.

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| Frequency (Thousands) | 45 | 13 | 12 | 16 | 9 | 5 |
| Code word with fix length | 000 | 001 | 010 | 011 | 100 | 101 |
| Code word variable length | 0 | 101 | 100 | 111 | 1101 | 1100 |

File size (code with fix length): $300.000$ bits.
File size (code with variable length): $224.000$ bits.

# Huffman-Codes

- Consider prefix-codes: no code word can start with a different codeword.

# Huffman-Codes

- Consider prefix-codes: no code word can start with a different codeword.
- Prefix codes can, compared with other codes, achieve the optimal **data compression** (without proof here).

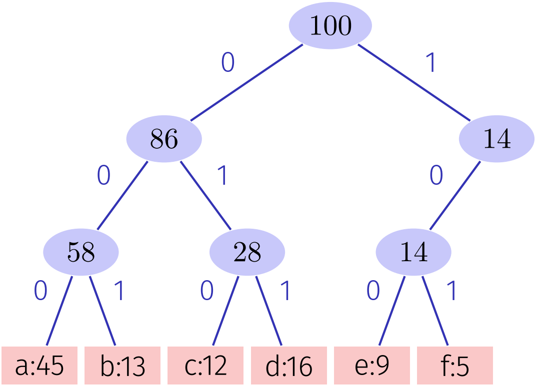
# Huffman-Codes

- Consider prefix-codes: no code word can start with a different codeword.
- Prefix codes can, compared with other codes, achieve the optimal **data compression** (without proof here).
- Encoding: concatenation of the code words without stop character (difference to morsing).
  affe $\rightarrow 0 \cdot 1100 \cdot 1100 \cdot 1101 \rightarrow 0110011001101$

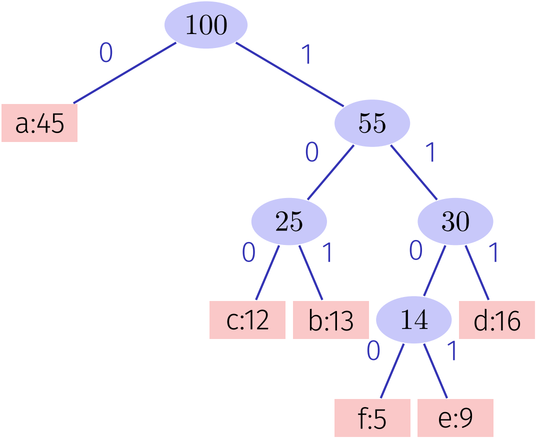
# Huffman-Codes

- Consider prefix-codes: no code word can start with a different codeword.
- Prefix codes can, compared with other codes, achieve the optimal **data compression** (without proof here).
- Encoding: concatenation of the code words without stop character (difference to morsing).
  affe $\rightarrow 0 \cdot 1100 \cdot 1100 \cdot 1101 \rightarrow 0110011001101$
- Decoding simple because prefixcode
  $0110011001101 \rightarrow 0 \cdot 1100 \cdot 1100 \cdot 1101 \rightarrow$ affe

# Code trees




Code words with fixed length




Code words with variable length

# Properties of the Code Trees

- An optimal coding of a file is alway represented by a complete binary tree: every inner node has two children.

# Properties of the Code Trees

- An optimal coding of a file is alway represented by a complete binary tree: every inner node has two children.
- Let $C$ be the set of all code words, $f(c)$ the frequency of a codeword $c$ and $d_T(c)$ the depth of a code word in tree $T$. Define the cost of a tree as

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c).$$

(cost = number bits of the encoded file)

# Properties of the Code Trees

- An optimal coding of a file is alway represented by a complete binary tree: every inner node has two children.
- Let $C$ be the set of all code words, $f(c)$ the frequency of a codeword $c$ and $d_T(c)$ the depth of a code word in tree $T$. Define the cost of a tree as

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c).$$

  (cost = number bits of the encoded file)

In the following a code tree is called optimal when it minimizes the costs.

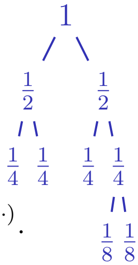
# Probabilitiy Distributions

The sum to be minimized

$$\sum_{c \in C} f(c) \cdot d_T(c)$$

can be written as

$$- \sum_{c \in C} f(c) \cdot \log_2 g_T(c), \text{ where } g_T(\cdot) := 2^{-d_T(\cdot)}.$$




$g_T(\cdot)$ can be understood as discrete probability distribution because it holds that $\sum_c g_T(c) = 1$. That is a property of a complete binary tree because each inner node has two child nodes.

# Probabilitiy Distributions

For two discrete proability distributions $f$ and $g$ over $C$ the **Gibbs inequality** holds

$$\underbrace{-\sum_{c \in C} f(c) \log f(c)}_{\text{Entropy of } f} \leq -\sum_{c \in C} f(x) \log g(c)$$

with equality if and only if $f(c) = g(c)$ for each $c \in C$.

# Probabilitiy Distributions

For two discrete proability distributions $f$ and $g$ over $C$ the **Gibbs inequality** holds

$$\underbrace{-\sum_{c\in C} f(c)\log f(c)}_{\text{Entropy of } f} \leq -\sum_{c\in C} f(x)\log g(c)$$

with equality if and only if $f(c) = g(c)$ for each $c \in C$.

**Consequence** if $f(c) \in \{2^{-k}, k \in \mathbb{N}\}$ for all $c \in C$, then the optimal code tree can be formed easily with $d_T(c) = -\log_2 f(c)$.

# Shannon Fano Coding

**Approximative algorithm of Shannon and Fano**

1. Sort the keys by frequency, wlog $p_1 \leq p_2 \leq \dots \leq p_n$
2. Partition the keys into two sets of almost equal weight, i.e. into sets $A = \{1, \dots, k\}$ and $B = \{k+1, \dots, n\}$ such that $\sum_{i \in A} p_i \approx \sum_{i \in B} p_i$. Recursion until all sets contain a single element.

Running Time:

# Shannon Fano Coding

**Approximative algorithm of Shannon and Fano**

1. Sort the keys by frequency, wlog $p_1 \leq p_2 \leq ... \leq p_n$
2. Partition the keys into two sets of almost equal weight, i.e. into sets $A = \{1, \ldots, k\}$ and $B = \{k+1, \ldots, n\}$ such that $\sum_{i \in A} p_i \approx \sum_{i \in B} p_i$. Recursion until all sets contain a single element.

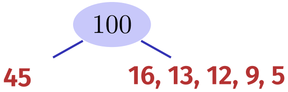
Running Time: $\Theta(n \log n)$

# Shannon Fano Coding

**45, 16, 13, 12, 9, 5**

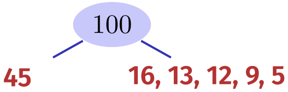
# Shannon Fano Coding

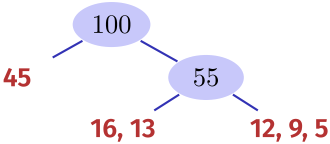
**45, 16, 13, 12, 9, 5**



# Shannon Fano Coding

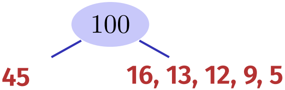
**45, 16, 13, 12, 9, 5**

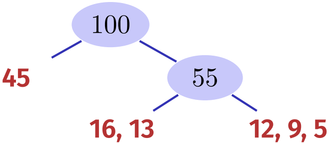


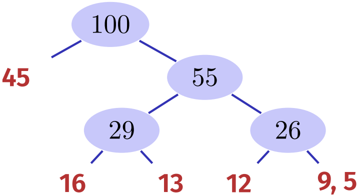



# Shannon Fano Coding

45, 16, 13, 12, 9, 5

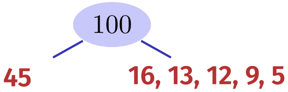


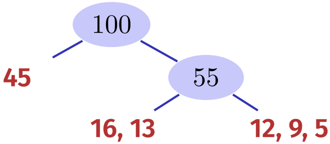



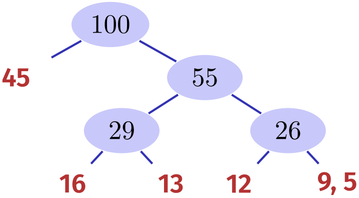



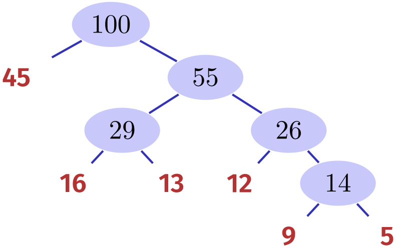
# Shannon Fano Coding

**45, 16, 13, 12, 9, 5**



# Problem

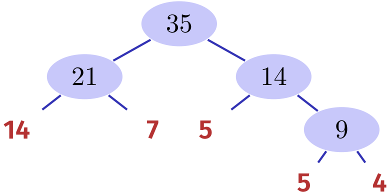
The approximate algorithm of Shannon and Fano does not always provide the optimal result

Example $\{14, 7, 5, 5, 4\}$ with lower bound (entropy) $B(T) \geq 75.35$

# Problem

The approximate algorithm of Shannon and Fano does not always provide the optimal result

Example $\{14, 7, 5, 5, 4\}$ with lower bound (entropy) $B(T) \geq 75.35$

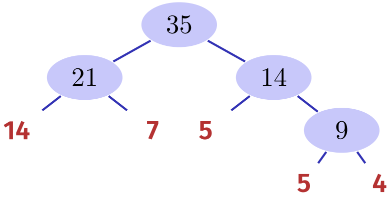



**Shannon-Fano Coding,** $B(T) = 79$

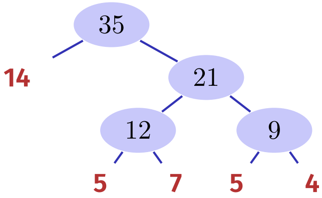
# Problem

The approximate algorithm of Shannon and Fano does not always provide the optimal result

Example $\{14, 7, 5, 5, 4\}$ with lower bound (entropy) $B(T) \geq 75.35$




**Shannon-Fano Coding,** $B(T) = 79$




**Optimal,** $B(T) = 77$

# Huffman's Idea

Tree construction bottom up

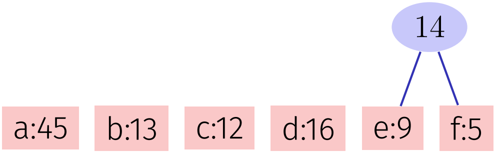
- Start with the set $C$ of code words

a:45 b:13 c:12 d:16 e:9 f:5

# Huffman's Idea

Tree construction bottom up

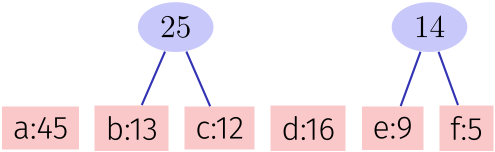
- Start with the set $C$ of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.



# Huffman's Idea

Tree construction bottom up

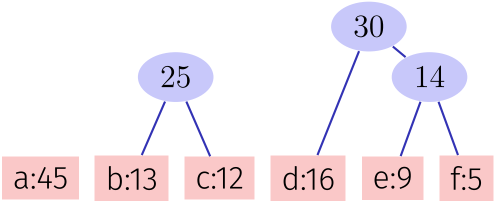
- Start with the set $C$ of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.



# Huffman's Idea

Tree construction bottom up

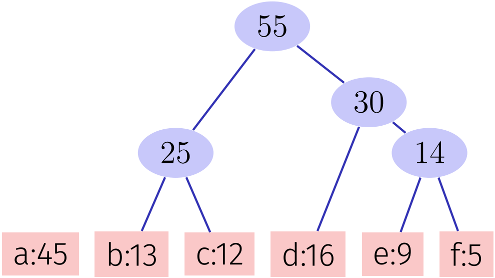
- Start with the set $C$ of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.



# Huffman's Idea

Tree construction bottom up

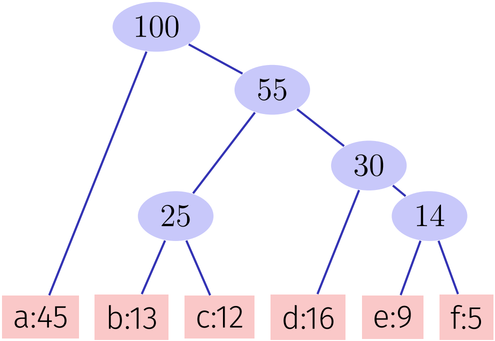
- Start with the set $C$ of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.



# Huffman's Idea

Tree construction bottom up

- Start with the set $C$ of code words
- Replace iteriatively the two nodes with smallest frequency by a new parent node.



# Algorithm Huffman($C$)

**Input**: code words $c \in C$
**Output**: Root of an optimal code tree

```
n ← |C|
Q ← C
for i = 1 to n − 1 do
    allocate a new node z
    z.left ← ExtractMin(Q)                    // extract word with minimal frequency.
    z.right ← ExtractMin(Q)
    z.freq ← z.left.freq + z.right.freq
    Insert(Q, z)
return ExtractMin(Q)
```

# Analyse

Use a heap: build Heap in $\mathcal{O}(n)$. Extract-Min in $O(\log n)$ for $n$ Elements. Yields a runtime of $O(n \log n)$.

# The greedy approach is correct

**Theorem 20**

*Let $x$, $y$ be two symbols with smallest frequencies in $C$ and let $T'(C')$ be an optimal code tree to the alphabet $C' = C - \{x, y\} + \{z\}$ with a new symbol $z$ with $f(z) = f(x) + f(y)$. Then the tree $T(C)$ that is constructed from $T'(C')$ by replacing the node $z$ by an inner node with children $x$ and $y$ is an optimal code tree for the alphabet $C$.*

# Proof

It holds that
$f(x) \cdot d_T(x) + f(y) \cdot d_T(y) = (f(x) + f(y)) \cdot (d_{T'}(z) + 1) = f(z) \cdot d_{T'}(x) + f(x) + f(y)$.
Thus $B(T') = B(T) - f(x) - f(y)$.

Assumption: $T$ is not optimal. Then there is an optimal tree $T''$ with $B(T'') < B(T)$. We assume that $x$ and $y$ are brothers in $T''$. Let $T'''$ be the tree where the inner node with children $x$ and $y$ is replaced by $z$. Then it holds that $B(T''') = B(T'') - f(x) - f(y) < B(T) - f(x) - f(y) = B(T')$. Contradiction to the optimality of $T'$.

The assumption that $x$ and $y$ are brothers in $T''$ can be justified because a swap of elements with smallest frequency to the lowest level of the tree can at most decrease the value of $B$.

# Recursive Problem-Solving Strategies

| Brute Force Enumeration | Backtracking | Divide and Conquer | Dynamic Programming | Greedy |
|---|---|---|---|---|

# Recursive Problem-Solving Strategies

| Brute Force Enumeration | Backtracking | Divide and Conquer | Dynamic Programming | Greedy |
|---|---|---|---|---|
| Recursive Enumerability | Constraint Satisfaction, Partial Validation | Optimal Substructure | Optimal Substructure, Overlapping Subproblems | Optimal Substructure, Greedy Choice Property |

# Recursive Problem-Solving Strategies

| Brute Force Enumeration | Backtracking | Divide and Conquer | Dynamic Programming | Greedy |
|---|---|---|---|---|
| Recursive Enumerability | Constraint Satisfaction, Partial Validation | Optimal Substructure | Optimal Substructure, Overlapping Subproblems | Optimal Substructure, Greedy Choice Property |
| DFS, BFS, all Permutations, Tree Traversal | n-Queen, Sudoku, m-Coloring, SAT-Solving, naive TSP | Binary Search, Mergesort, Quicksort, Hanoi Towers, FFT | Bellman Ford, Warshall, Rod-Cutting, LAS, Editing Distance, Knapsack Problem DP | Dijkstra, Kruskal, Huffmann Coding |