Algorithm Design using GREEDY Method

Partha P Chakrabarti

Indian Institute of Technology Kharagpur

Algorithm Design by Recursion Transformation

- Algorithms and Programs
- Pseudo-Code
- Algorithms + Data Structures = Programs
- Initial Solutions + Analysis + Solution Refinement + Data Structures = Final Algorithm
- Use of Recursive Definitions as Initial Solutions
- Recurrence Equations for Proofs and Analysis
- Solution Refinement through Recursion Transformation and Traversal
- Data Structures for saving past computation for future use

1. Initial Solution
   a. Recursive Definition – A set of Solutions
   b. Inductive Proof of Correctness
   c. Analysis Using Recurrence Relations
2. Exploration of Possibilities
   a. Decomposition or Unfolding of the Recursion Tree
   b. Examination of Structures formed
   c. Re-composition Properties
3. Choice of Solution & Complexity Analysis
   a. Balancing the Split, Choosing Paths
   b. Identical Sub-problems
4. Data Structures & Complexity Analysis
   a. Remembering Past Computation for Future
   b. Space Complexity
5. Final Algorithm & Complexity Analysis
   a. Traversal of the Recursion Tree
   b. Pruning
6. Implementation
   a. Available Memory, Time, Quality of Solution, etc

Overview of Algorithm Design

1. Initial Solution
   a. Recursive Definition – A set of Solutions
   b. Inductive Proof of Correctness
   c. Analysis Using Recurrence Relations
2. Exploration of Possibilities
   a. Decomposition or Unfolding of the Recursion Tree
   b. Examination of Structures formed
   c. Re-composition Properties
3. Choice of Solution & Complexity Analysis
   a. Balancing the Split, Choosing Paths
   b. Identical Sub-problems
4. Data Structures & Complexity Analysis
   a. Remembering Past Computation for Future
   b. Space Complexity
5. Final Algorithm & Complexity Analysis
   a. Traversal of the Recursion Tree
   b. Pruning
6. Implementation
   a. Available Memory, Time, Quality of Solution, etc

1. Core Methods
   a. Divide and Conquer
   b. Greedy Algorithms
   c. Dynamic Programming
   d. Branch-and-Bound
   e. Analysis using Recurrences
   f. Advanced Data Structuring
2. Important Problems to be addressed
   a. Sorting and Searching
   b. Strings and Patterns
   c. Trees and Graphs
   d. Combinatorial Optimization
3. Complexity & Advanced Topics
   a. Time and Space Complexity
   b. Lower Bounds
   c. Polynomial Time, NP-Hard
   d. Parallelizability, Randomization

Recursive Definitions and Greedy Choices

- ❑ Revisiting the 1-D Knapsack Problem:

Given a set of granular items of n different types, each having a profit per unit weight, and a Knapsack of capacity C, how to fill up the Knapsack to maximize the total profit.

Concept of Greedy Choice

Variants:

- ➢ Total amount of each item is limited
- ➢ Minimum and maximum amounts of any item are given
- ➢ Items have volume in addition to weight and a total volume limit V is also to be satisfied
- ➢ VC Funding, Land lease Problem – various issues involved

Optimal Merge Sequence: Problem

Merge Sequence (L)

{ $L = \{L_1, L_2, \ldots, L_n\}$

each L_i has $|L_i| = l_i$ elements

which are already sorted

GENERAL RECURSIVE METHOD

for each pair (L_i, L_j)

$L_{ij} = \text{Merge}(L_i, L_j)$

$L_R = L - \{L_i\} - \{L_i\} + L_{ij}$

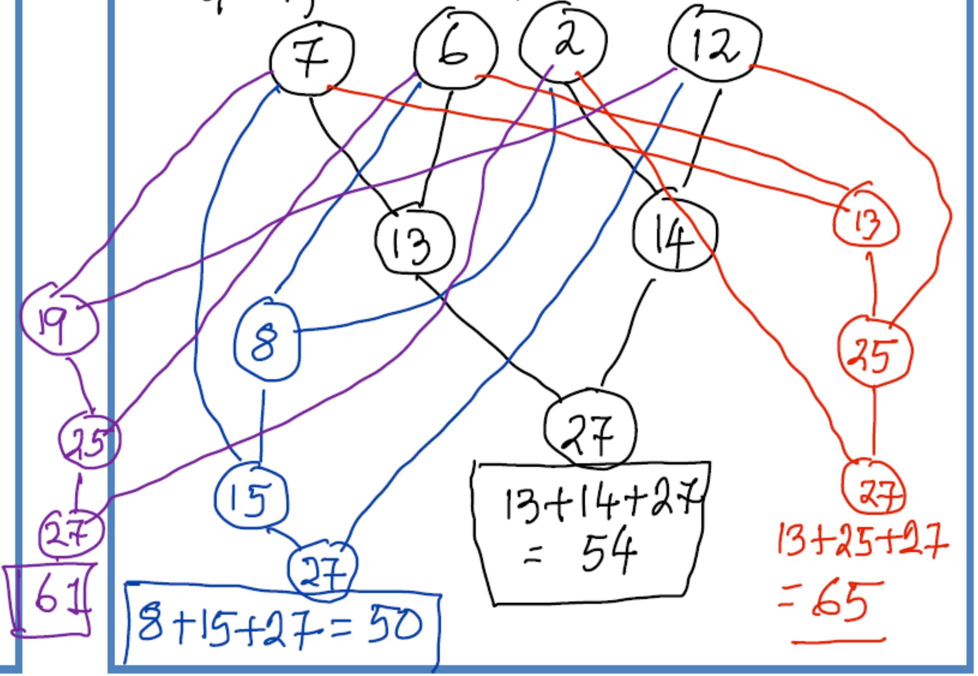
$M_{ij} = \text{MergeSequence}(L_R)$

Best Solution/option of choosing L_i & L_j

Example

Four Lists L_1, L_2, L_3, L_4

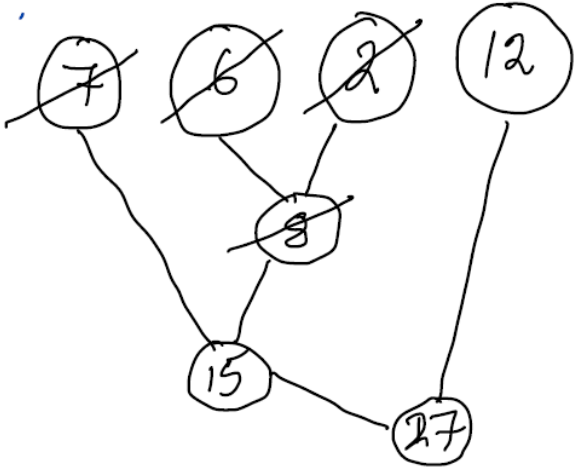
$l_1 = 7$, $l_2 = 6$, $l_3 = 2$, $l_4 = 12$



Optimal Merge Sequence: Algorithm & Choice

$L = \{L_1, L_2, \ldots, L_n\}$

Choose: L_i and L_j such that l_i and l_j $(|L_i| = l_i)$ $(|L_j| = l_j)$ are the smallest sized sets in L.



1. Prove that this GREEDY choice always yields the optimal value.

2. Analyze the Time Complexity of this Algorithm

[Assignments / Homework / Tutorial]

Use of a GREEDY CHOICE METHOD

↳ is also another way of looking at bottom-up evaluation in Mergesort

Activity Selection Problem

- Suppose A set of activities $S=\{a_1, a_2,\ldots, a_n\}$
  - They use resources, such as lecture hall, one lecture at a time
  - Each a_i, has a start time s_i, and finish time f_i, with $0\leq s_i< f_i<\infty$.
  - a_i and a_j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap
- Goal: select maximum-size subset of mutually compatible activities.

i	1	2	3	4	5	6	7	8	9	10	11
$start\_time_i$	1	3	0	5	3	5	6	8	8	2	12
$finish\_time_i$	4	5	6	7	8	9	10	11	12	13	14

Sample Schedulable Subsets:

{a3, a9, a11}, {a1, a4, a8, a11}, (a2, a4, a9, a11}

RECURSIVE DECOMPOSITION

- Assume that $f_1 \leq\ldots \leq f_n$.
- Define $S_{ij}=\{a_k: f_i\leq s_k<f_k\leq s_j\}$, i.e., all activities starting after a_i finished and ending before a_j begins.
- Define two fictitious activities a_0 with $f_0=0$ and a_{n+1} with $s_{n+1}=\infty$
  - So $f_0 \leq f_1 \leq\ldots \leq f_{n+1}$.
- Then an optimal solution including a_k to S_{ij} contains within it the optimal solution to S_{ik} and S_{kj}.
- So Apply a Recursive Decomposition over all possible a_k S_{ij} in and choose the one which maximizes the size of the set $\{a_k$ U Result(S_{ik}) U Result(S_{kj}.)} and return this maximum set as the Result(S_{ij})

Activity Selection Problem: DP Solution

RECURSIVE DECOMPOSITION

- Assume that $f_1 \leq \ldots \leq f_n$.
- Define $S_{ij}=\{a_k: f_i \leq s_k < f_k \leq s_j\}$, i.e., all activities starting after a_i finished and ending before a_j begins.
- Define two fictitious activities a_0 with $f_0=0$ and a_{n+1} with $s_{n+1}=\infty$
  - So $f_0 \leq f_1 \leq \ldots \leq f_{n+1}$.
- Then an optimal solution including a_k to S_{ij} contains within it the optimal solution to S_{ik} and S_{kj}.
- So Apply a Recursive Decomposition over all possible a_k S_{ij} in and choose the one which maximizes the size of the set $\{a_k$ U Result(S_{ik}) U Result(S_{kj}.)$\}$ and return this maximum set as the Result(S_{ij})

A Dynamic Programming based recursive definition to be used with memoization:

- Assume c[n+1,n+1] with c[i,j] is the number of activities in a maximum-size subset of mutually compatible activities in S_{ij}. So the solution is c[0,n+1]=$S_{0,n+1}$.
- $$c[i,j]= \begin{cases} 0, & \text{if } S_{ij}=\varnothing \\ \max\{c[i,k]+c[k,j]+1\}, & \text{if } S_{ij}\neq\varnothing,\ i<k<j \text{ and } a_k \in S_{ij} \end{cases}$$

Activity Selection Problem: Greedy Choice

RECURSIVE DECOMPOSITION

- Assume that $f_1 \leq \ldots \leq f_n$.
- Define $S_{ij}=\{a_k: f_i \leq s_k < f_k \leq s_j\}$, i.e., all activities starting after a_i finished and ending before a_j begins.
- Define two fictitious activities a_0 with $f_0=0$ and a_{n+1} with $s_{n+1}=\infty$
  - So $f_0 \leq f_1 \leq \ldots \leq f_{n+1}$.
- Then an optimal solution including a_k to S_{ij} contains within it the optimal solution to S_{ik} and S_{kj}.
- So Apply a Recursive Decomposition over all possible a_k S_{ij} in and choose the one which maximizes the size of the set $\{a_k \cup \text{Result}(S_{ik}) \cup \text{Result}(S_{kj}.)\}$ and return this maximum set as the $\text{Result}(S_{ij})$

A Dynamic Programming based recursive definition to be used with memoization:

- Assume c[n+1,n+1] with c[i,j] is the number of activities in a maximum-size subset of mutually compatible activities in S_{ij}. So the solution is $c[0,n+1]=S_{0,n+1}$.
- $c[i,j]= \begin{cases} 0, & \text{if } S_{ij}=\emptyset \\ \max\{c[i,k]+c[k,j]+1\}, & \text{if } S_{ij}\neq\emptyset,\ i<k<j \text{ and } a_k \in S_{ij} \end{cases}$

Consider any nonempty subproblem S_{ij}, and let a_m be the activity in S_{ij} with earliest finish time: $f_m=\min\{f_k : a_k \in S_{ij}\}$, then

- Activity a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.
- The subproblem S_{im} is empty, so that choosing a_m leaves S_{mj} as the only one that may be nonempty.

Activity Selection Problem: Final Algorithm

- Given a set of tasks, with Start and End Times, to be scheduled without conflict in a single resource, find the maximum number of tasks that can be scheduled.

i	1	2	3	4	5	6	7	8	9	10	11
$start\_time_i$	1	3	0	5	3	5	6	8	8	2	12
$finish\_time_i$	4	5	6	7	8	9	10	11	12	13	14

Consider any nonempty subproblem S_{ij}, and let a_m be the activity in S_{ij} with earliest finish time: $f_m = \min\{f_k : a_k \in S_{ij}\}$, then

- Activity a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.
- The subproblem S_{im} is empty, so that choosing a_m leaves S_{mj} as the only one that may be nonempty.

Activity Selection Problem: All Task Scheduling

- ❑ Given a set of tasks, with Start and End Times, to be scheduled without conflict in a single resource, find the minimum number of resources to schedule all tasks.

i	1	2	3	4	5	6	7	8	9	10	11
$start\_time_i$	1	3	0	5	3	5	6	8	8	2	12
$finish\_time_i$	4	5	6	7	8	9	10	11	12	13	14

- ❑ Relationship with Colouring Interval Graphs

Huffman Coding

Suppose we have a 100,000 character data file that we wish to store. The file contains only 6 characters, with the following frequencies:

	a	b	c	d	e	f
Frequency in '000s	45	13	12	16	9	5

A *binary code* encodes each character as a binary string or *codeword*. We would like to find a binary code that encodes the file using as few bits as possible, ie., *compresses it* as much as possible.

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
a fixed-length	000	001	010	011	100	101
a variable-length	0	101	100	111	1101	1100

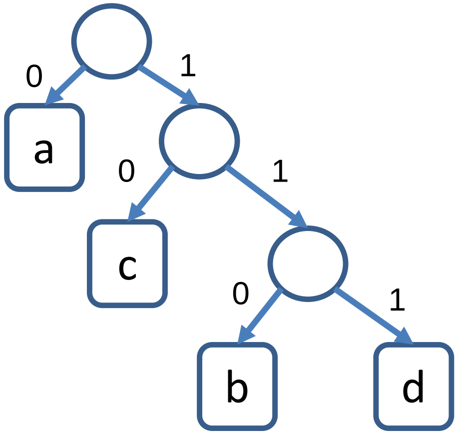
The fixed length-code requires 3,00,000 bits to store the file. The variable-length code uses only (45*1+13*3+12*3+16*3+9*4+5*4)*1000 = 2,24,000 bits, saving a lot of space! How to find the optimal coding?

Coding: Alphabet = {a, b, c, d}

Fixed Length Coding: If the code is a = 00; b = 01; c = 10; d = 11: then the word bad is encoded into 010011

Prefix-Coding: If the code is a = 0; b = 110; c = 10; d = 111, then the word bad is encoded into 1100111

Decoding: For Fixed Length Code it is a Table Lookup. For Prefix Coding, it is a Tree-Based Search



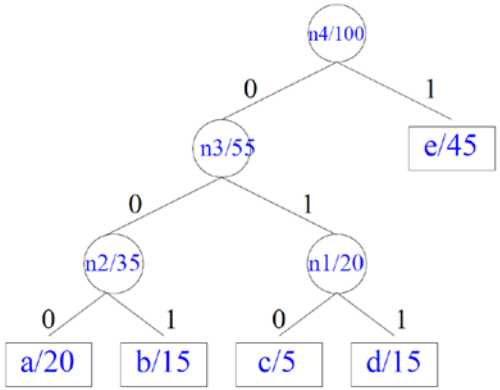
Huffman Coding Problem: Designing the Best Tree

The problem: Given an alphabet $A = \{a_1, \ldots, a_n\}$ with frequency distribution $f(a_i)$ find a binary prefix code C for A that minimizes the number of bits

$$B(C) = \sum_{a=1}^{n} f(a_i) L(c(a_i))$$

needed to encode a message of $\sum_{a=1}^{n} f(a)$ characters, where $c(a_i)$ is the codeword for encoding a_i, and $L(c(a_i))$ is the length of the codeword $c(a_i)$.

Remark: Huffman developed a nice greedy algorithm for solving this problem and producing a minimum-cost (optimum) prefix code. The code that it produces is called a *Huffman code* .



$\{a = 000,\ b = 001,\ c = 010,\ d = 011,\ e = 1\}$

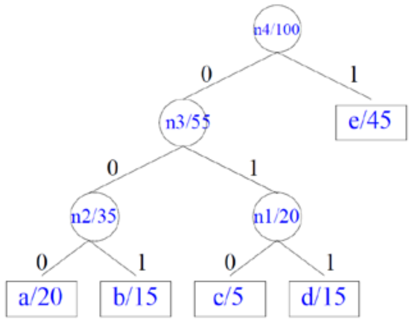
Huffman Coding: Greedy Algorithm

Step 1: Pick two letters x, y from alphabet A with the smallest frequencies and create a subtree that has these two characters as leaves. (greedy idea)
Label the root of this subtree as z.

Step 2: Set frequency $f(z) = f(x) + f(y)$.
Remove x, y and add z creating new alphabet
$A' = A \cup \{z\} - \{x, y\}$.
Note that $|A'| = |A| - 1$.

Repeat this procedure, called *merge*, with new alphabet A' until an alphabet with only one symbol is left.

The resulting tree is the Huffman code.



$$\{a = 000,\ b = 001,\ c = 010,\ d = 011,\ e = 1\}$$

SIMILAR TO OUR OPTIMAL MERGE SEQUENCE PROBLEM

Algorithm Design by Recursion Transformation

- Algorithms and Programs
- Pseudo-Code
- Algorithms + Data Structures = Programs
- Initial Solutions + Analysis + Solution Refinement + Data Structures = Final Algorithm
- Use of Recursive Definitions as Initial Solutions
- Recurrence Equations for Proofs and Analysis
- Solution Refinement through Recursion Transformation and Traversal
- Data Structures for saving past computation for future use

1. Initial Solution
   a. Recursive Definition – A set of Solutions
   b. Inductive Proof of Correctness
   c. Analysis Using Recurrence Relations
2. Exploration of Possibilities
   a. Decomposition or Unfolding of the Recursion Tree
   b. Examination of Structures formed
   c. Re-composition Properties
3. Choice of Solution & Complexity Analysis
   a. Balancing the Split, Choosing Paths
   b. Identical Sub-problems
4. Data Structures & Complexity Analysis
   a. Remembering Past Computation for Future
   b. Space Complexity
5. Final Algorithm & Complexity Analysis
   a. Traversal of the Recursion Tree
   b. Pruning
6. Implementation
   a. Available Memory, Time, Quality of Solution, etc

Overview of Algorithm Design

1. Initial Solution
   a. Recursive Definition – A set of Solutions
   b. Inductive Proof of Correctness
   c. Analysis Using Recurrence Relations
2. Exploration of Possibilities
   a. Decomposition or Unfolding of the Recursion Tree
   b. Examination of Structures formed
   c. Re-composition Properties
3. Choice of Solution & Complexity Analysis
   a. Balancing the Split, Choosing Paths
   b. Identical Sub-problems
4. Data Structures & Complexity Analysis
   a. Remembering Past Computation for Future
   b. Space Complexity
5. Final Algorithm & Complexity Analysis
   a. Traversal of the Recursion Tree
   b. Pruning
6. Implementation
   a. Available Memory, Time, Quality of Solution, etc

1. Core Methods
   a. Divide and Conquer
   b. Greedy Algorithms
   c. Dynamic Programming
   d. Branch-and-Bound
   e. Analysis using Recurrences
   f. Advanced Data Structuring
2. Important Problems to be addressed
   a. Sorting and Searching
   b. Strings and Patterns
   c. Trees and Graphs
   d. Combinatorial Optimization
3. Complexity & Advanced Topics
   a. Time and Space Complexity
   b. Lower Bounds
   c. Polynomial Time, NP-Hard
   d. Parallelizability, Randomization

Thank you

Any Questions?