DESIGN AND ANALYSIS OF ALGORITHMS (DAA)

GREEDY ALGORITHMS

Course Instructor: Dr. Shreya Ghosh

OPTIMIZATION PROBLEMS

- A problem that may have many feasible solutions.
- Each solution has a value
  - In maximization problem, we wish to find a solution to maximize the value
  - In the minimization problem, we wish to find a solution to minimize the value

OPTIMIZATION PROBLEMS

- For most optimization problems you want to find, not just ***a*** solution, but the ***best*** solution.
- A ***greedy algorithm*** sometimes works well for optimization problems. It works in phases. At each phase:
  - You take the best you can get right now, without regard for future consequences.
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum.

EXAMPLE: COUNTING MONEY

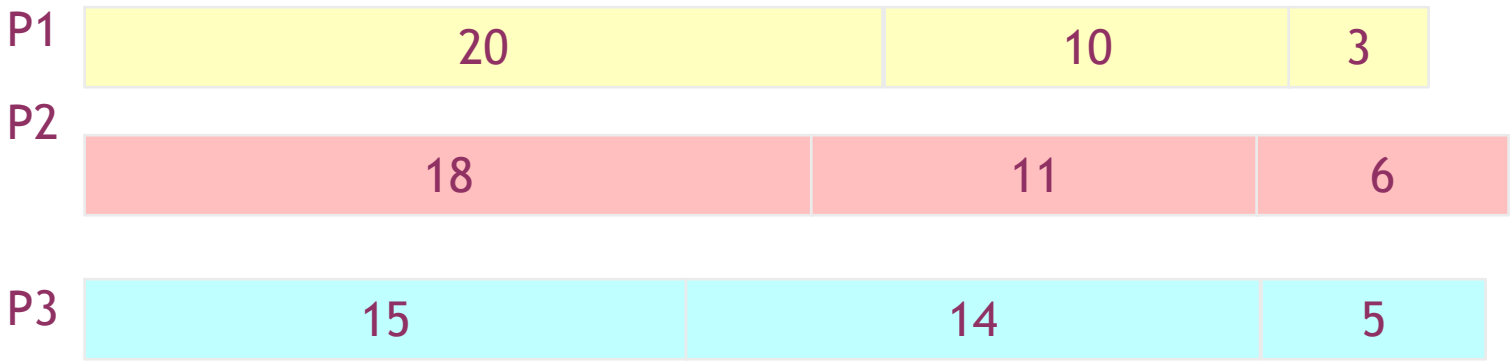
- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm to do this would be:
  At each step, take the largest possible bill or coin that does not overshoot
  - Example: To make $6.39, you can choose:
    - a $5 bill
    - a $1 bill, to make $6
    - a 25¢ coin, to make $6.25
    - A 10¢ coin, to make $6.35
    - four 1¢ coins, to make $6.39
- For US money, the greedy algorithm always gives the optimum solution

DOES GREEDY ALGORITHM WORK HERE?

- In some (fictional) monetary system, "krons" come in 1 kron, 7 kron, and 10 kron coins
- Using a greedy algorithm to count out 15 krons, you would get
  - A 10 kron piece
  - Five 1 kron pieces, for a total of 15 krons
  - This requires six coins
- A better solution would be to use two 7 kron pieces and one 1 kron piece
  - This only requires three coins
- The greedy algorithm results in a solution, but not in an optimal solution

A SCHEDULING PROBLEM

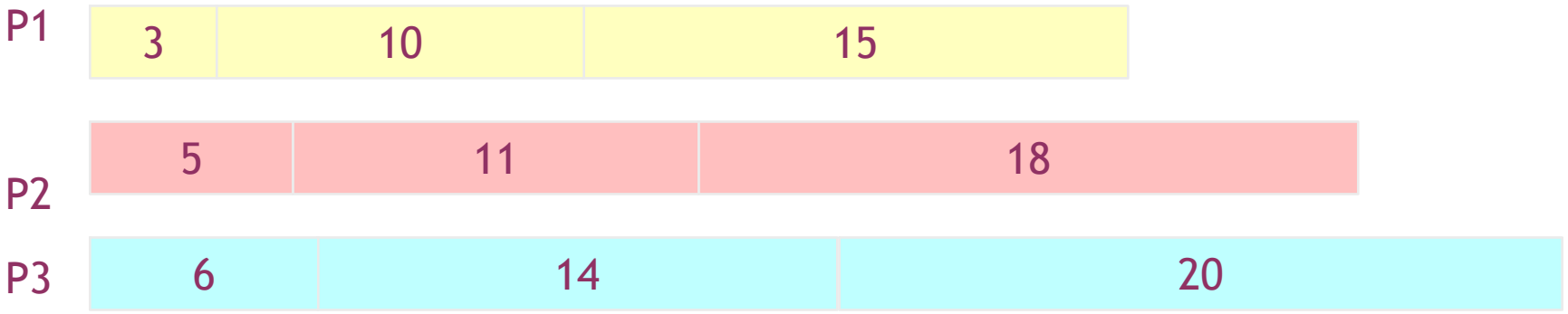
- You have to run nine jobs, with running times of 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes.
- You have three processors on which you can run these jobs.
- You decide to do the longest-running jobs first, on whatever processor is available.



- Time to completion: 18 + 11 + 6 = 35 minutes
- This solution isn't bad, but we might be able to do better

ANOTHER APPROACH

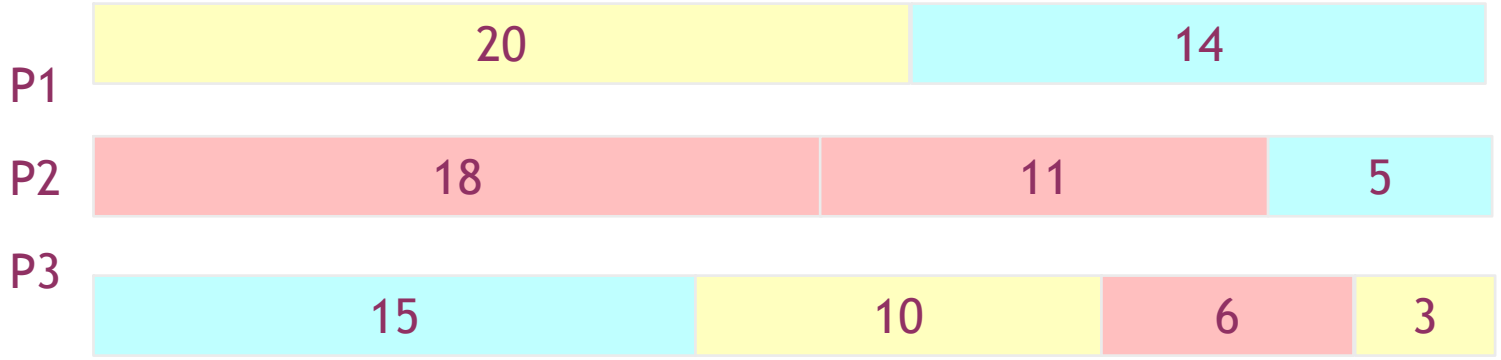
- What would be the result if you ran the *shortest* job first?
- Again, the running times are 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes



- That wasn't such a good idea; time to completion is now
  6 + 14 + 20 = 40 minutes
- Note, however, that the greedy algorithm itself is fast
  - All we had to do at each stage was pick the minimum or maximum

AN OPTIMUM SOLUTION

- Better solutions do exist:



- This solution is clearly optimal
- Clearly, there are other optimal solutions
- How do we find such a solution?
  - One way: Try all possible assignments of jobs to processors
  - Unfortunately, this approach can take exponential time

DATA COMPRESSION

- Suppose we have 1000000000 (1G) character data file that we wish to include in an email.
- Suppose file only contains 26 letters {a,…,z}.
- Suppose each letter α in {a,…,z} occurs with frequency f_α.
- Suppose we encode each letter by a binary code
- If we use a fixed length code, we need 5 bits for each character
- The resulting message length is $5(f_a + f_b + \ldots + f_z)$
- **Can we do better?**

HUFFMAN CODES

- Most character code systems (ASCII, unicode) use fixed length encoding
- If frequency data is available and there is a wide variety of frequencies, variable length encoding can save 20% to 90% space
- Which characters should we assign shorter codes; which characters will have longer codes?

DATA COMPRESSION: A SMALLER EXAMPLE

- Suppose the file only has 6 letters {a,b,c,d,e,f} with frequencies

a	b	c	d	e	f	
.45	.13	.12	.16	.09	.05	
000	001	010	011	100	101	Fixed length
0	101	100	111	1101	1100	Variable length

- Fixed length 3G=3000000000 bits
- Variable length

$$(.45 \bullet 1 + .13 \bullet 3 + .12 \bullet 3 + .16 \bullet 3 + .09 \bullet 4 + .05 \bullet 4) = 2.24G$$

HOW TO DECODE?

- At first it is not obvious how decoding will happen, but this is possible if we use prefix codes

WHAT IS PREFIX CODE?

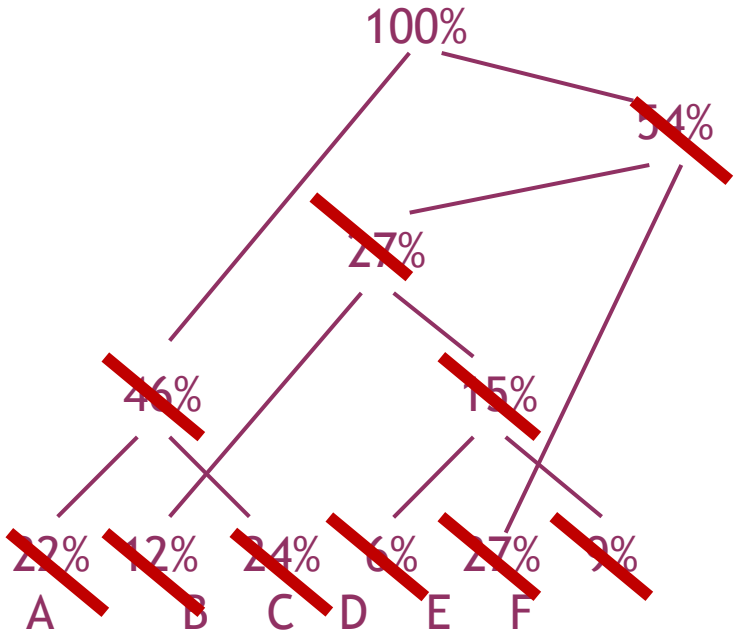
- No encoding of a character can be the prefix of the longer encoding of another character, for example, we could not encode *t* as 01 and *x* as 01101 since 01 is a prefix of 01101

- By using a binary tree representation we will generate prefix codes provided all letters are leaves

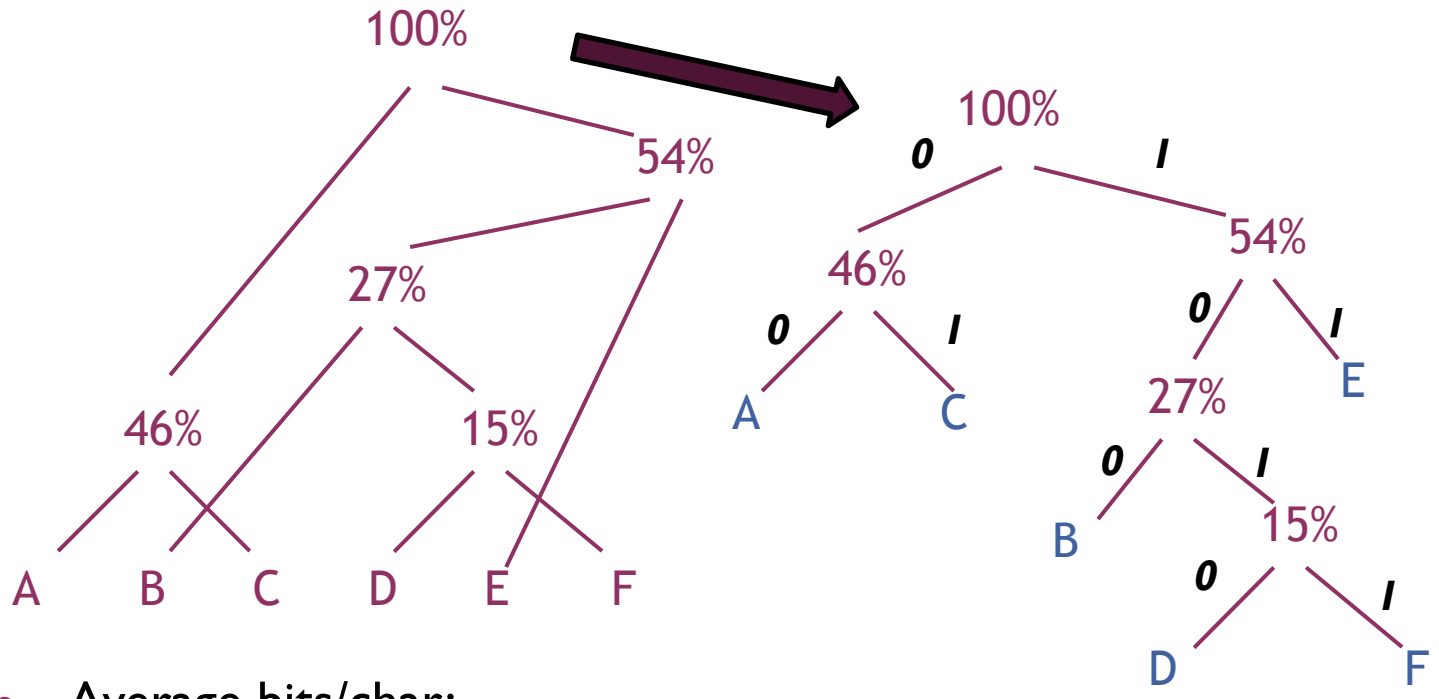
HUFFMAN CODING

- The Huffman encoding algorithm is a greedy algorithm
- Given the percentage the each character appears in a corpus, determine a variable-bit pattern for each char.
- You always pick the two smallest percentages to combine.

HUFFMAN CODING: AN EXAMPLE



HUFFMAN CODING



A=00
B=100
C=01
D=1010
E=11
F=1011

- Average bits/char:
  0.22*2 + 0.12*3 + 0.24*2 + 0.06*4 + 0.27*2 + 0.09*4
  = 2.42
- The solution found doing this is an optimal solution.

HUFFMAN CODING ALGORITHM

Input: a collection C of objects containing a character and its frequency.
Output: the root of a Huffman tree
Uses: priority queue Q
$n = |C|$
$Q = C$
for i=1 to n-1
    z = new treenode
    left(z) = ExtractMin(Q)
    right(z) = ExtractMin(Q)
    frequency(z) = frequency(left(z)) + frequency(right(z))
    Insert(Q, z)
return ExtractMin(Q)

Running time: $(O(n \lg n))$

ANOTHER EXAMPLE – HUFFMAN CODING

Letter freqency table

Letter	Z	K	M	C	U	D	L	E
Frequency	2	7	24	32	37	42	42	120

SOLUTION

Huffman code

Letter	Freq	Code	Bits
E	120	0	1
D	42	101	3
L	42	110	3
U	37	100	3
C	32	1110	4
M	24	11111	5
K	7	111101	6
Z	2	111100	6

ANALYSIS

- A greedy algorithm typically makes (approximately) n choices for a problem of size n
  - (The first or last choice may be forced)

- Hence the expected running time is:
  O(n * O(choice(n))), where choice(n) is making a choice among n objects
  - Counting: Must find largest useable coin from among k sizes of coin (k is a constant), an O(k)=O(1) operation;
    - Therefore, coin counting is (n)

  - Huffman: Must sort n values before making n choices
    - Therefore, Huffman is O(n log n) + O(n) = O(n log n)

OTHER GREEDY ALGORITHMS

- **Dijkstra's** algorithm for finding the shortest path in a graph
  - Always takes the *shortest* edge connecting a known node to an unknown node
- **Kruskal's** algorithm for finding a minimum-cost spanning tree
  - Always tries the *lowest-cost* remaining edge
- **Prim's** algorithm for finding a minimum-cost spanning tree
  - Always takes the *lowest-cost* edge between nodes in the spanning tree and nodes not yet in the spanning tree

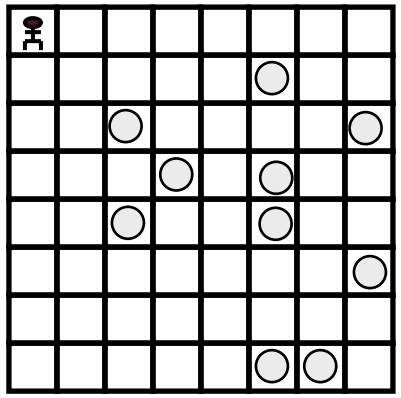
CONNECTING WIRES

- There are n white dots and n black dots, equally spaced, in a line
- You want to connect each white dot with some one black dot, with a minimum total length of "wire"
- Example:

- Total wire length above is 1 + 1 + 1 + 5 = 8
- Do you see a greedy algorithm for doing this?
- Does the algorithm guarantee an optimal solution?
  - Can you prove it?
  - Can you find a counterexample?

COLLECTING COINS

- A checkerboard has a certain number of coins on it
- A robot starts in the upper-left corner, and walks to the bottom left-hand corner
  - The robot can only move in two directions: right and down
  - The robot collects coins as it goes
- You want to collect *all* the coins using the *minimum* number of robots



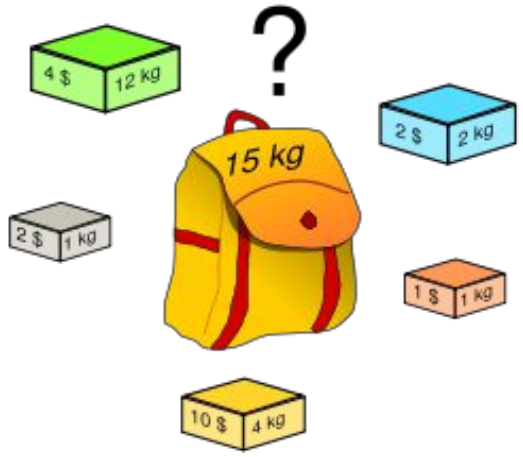
- Do you see a greedy algorithm for doing this?
- Does the algorithm guarantee an optimal solution?
  - Can you prove it?
  - Can you find a counterexample?

KNAPSACK

- The **knapsack problem** or **rucksack problem** is a problem in combinatorial optimization.
- It derives its name from the following maximization problem of the best choice of essentials that can fit into one bag to be carried on a trip.
- Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than a given limit and the total value is as large as possible.

THE ORIGINAL KNAPSACK PROBLEM (1)

- Problem Definition
  - Want to carry essential items in one bag
  - Given a set of items, each has
    - A cost (i.e., 12kg)
    - A value (i.e., 4$)



- Goal
  - To determine the # of each item to include in a collection so that
    - The total cost is less than some given cost
    - And the total value is as large as possible

THE ORIGINAL KNAPSACK PROBLEM (2)

- Three Types
  - 0/1 Knapsack Problem
    - restricts the number of each kind of item to zero or one
  - Bounded Knapsack Problem
    - restricts the number of each item to a specific value
  - Unbounded Knapsack Problem
    - places no bounds on the number of each item

- Complexity Analysis
  - The general knapsack problem is known to be NP-hard
    - No polynomial-time algorithm is known for this problem
  - Here, we use greedy heuristics which cannot guarantee the optimal solution

0/1 KNAPSACK PROBLEM (1)

- Problem: John wishes to take n items on a trip
  - The weight of item *i* is w_i & items are all different (0/1 Knapsack Problem)
  - The items are to be carried in a knapsack whose weight capacity is *c*
    - When sum of item weights ≤ c, all n items can be carried in the knapsack
    - When sum of item weights > c, some items must be left behind
- Which items should be taken/left?

0/1 KNAPSACK PROBLEM (2)

- John assigns a profit p_i to item i
  - All weights and profits are positive numbers
- John wants to select a subset of the n items to take
  - The weight of the subset should not exceed the capacity of the knapsack (constraint)
  - Cannot select a fraction of an item (constraint)
  - The profit of the subset is the sum of the profits of the selected items (optimization function)
  - The profit of the selected subset should be maximum (optimization criterion)

- Let $x_i = 1$ when item i is selected and $x_i = 0$ when item i is not selected
  - Because this is a 0/1 Knapsack Problem, you can choose the item or not choose it.

GREEDY ATTEMPTS FOR 0/1 KNAPSACK

- Apply greedy method:
  - Greedy attempt on capacity utilization
    - Greedy criterion: select items in increasing order of weight
    - When n = 2, c = 7, w = [3, 6], p = [2, 10],
      if only item 1 is selected □ profit of selection is 2 □ not best selection!
  - Greedy attempt on profit earned
    - Greedy criterion: select items in decreasing order of profit
    - When n = 3, c = 7, w = [7, 3, 2], p = [10, 8, 6],
      if only item 1 is selected □ profit of selection is 10 □ not best selection!

FRACTIONAL KNAPSACK PROBLEM

- In fractional knapsack problem, where we are given a set S of n items, s.t., each item *I* has a *positive* benefit b_i and a *positive* weight w_i, and we wish to find the maximum-benefit subset that *doesn't exceed* a given weight *W*.

- We are also allowed to take *arbitrary fractions* of each item.

FRACTIONAL KNAPSACK PROBLEM

- We can take an amount x_i of each item i such that

$$0 \leq x_i \leq w_i \text{ for each } i \in S \quad \text{and} \quad \sum_{i \in S} x_i \leq W.$$

  - The *total benefit* of the items taken is determined by the *objective function*

$$\sum_{i \in S} b_i (x_i / w_i)$$

FRACTIONAL KNAPSACK PROBLEM

Algorithm FractionalKnapsack(S, W):

Input: Set S of items, such that each item $i \in S$ has a positive benefit b_i and a positive weight w_i; positive maximum total weight W

Output: Amount x_i of each item $i \in S$ that maximizes the total benefit while not exceeding the maximum total weight W

for each item $i \in S$ **do**
  $x_i \leftarrow 0$
  $v_i \leftarrow b_i / w_i$ {***value index*** of item i}
$w \leftarrow 0$ {total weight}
while $w < W$ **do**
  remove from S an item i with highest value index {greedy choice}
  $a \leftarrow \min\{w_i, W - w\}$ {more than $W - w$ causes a weight overflow}
  $x_i \leftarrow a$
  $w \leftarrow w + a$

FRACTIONAL KNAPSACK PROBLEM

- In the solution we use a heap-based *PQ* to store the items of S, where the *key* of each item is its *value index*
- With *PQ*, each greedy choice, which removes an item with the greatest value index, takes *O(log n)* time
- The *fractional knapsack algorithm* can be implemented in time *O(n log n)*.

FRACTIONAL KNAPSACK PROBLEM

- Board work (Example)

DYNAMIC PROGRAMMING

OPTIMAL SUBSTRUCTURE PROPERTY

- If S is an optimal solution to a problem, then the components of S are optimal solutions to subproblems
- Examples:
  - True for knapsack
  - True for coin-changing
  - True for single-source shortest path
  - Not true for longest-simple-path

DYNAMIC PROGRAMMING

- Works "bottom-up"
  - Finds solutions to small sub-problems first
  - Stores them
  - Combines them somehow to find a solution to a slightly larger subproblem

- Compare to greedy approach
  - Also requires optimal substructure
  - But greedy makes choice first, then solves

PROBLEMS SOLVED WITH DYN. PROG.

- Coin changing
- Multiplying a sequence of matrices
  - Can do in various orders: (AB)C vs. A(BC)
  - Pick order that does fewest number of scalar multiplications
- Longest common subsequence
- All-pairs shortest paths (Floyd's algorithm)
- Constructing optimal binary search trees
- Knapsack problems

REMEMBER FIBONACCI NUMBERS?

- Recursive code:

```
long fib(int n) {
    assert(n >= 0);
    if ( n == 0 ) return 0;
    if ( n == 1 ) return 1;
    return fib(n-1) + fib(n-2);
}
```

- What's the problem?
  - Repeatedly solves the same subproblems

MEMOIZATION

- Before talking about dynamic programming, another general technique: **Memoization**
  - AKA using a *memory function*
- Simple idea:
  - Calculate and store solutions to subproblems
  - Before solving it (again), look to see if you've remembered it

MEMOIZATION

- Use a Table abstract data type
  - Lookup key: whatever identifies a subproblem
  - Value stored: the solution
- Could be an array/vector
  - E.g. for Fibonacci, store **fib(n)** using index **n**
  - Need to initialize the array
- Could use a map / hash-table

MEMOIZATION AND FIBONACCI

- Before recursive code below called, must initialize results[] so all values are -1

```
long fib_mem(int n, long results[]) {
    if ( results[n] != -1 )
        return results[n];  // return stored value
    long val;
    if ( n == 0 || n ==1 ) val = n; // odd but right
    else
        val = fib_mem(n-1, results)
              + fib_mem(n-2, results);
    results[n] = val; // store calculated value
    return val;
}
```

OBSERVATIONS ON FIB_MEM()

- Same elegant top-down, recursive approach based on definition
  - Without repeated subproblems
- Memory function: a function that remembers
  - Save time by using extra space
- Can show this runs in $\Theta(n)$

GENERAL STRATEGY OF DYN. PROG.

1. Structure: What's the structure of an optimal solution in terms of solutions to its subproblems?
2. Give a recursive definition of an optimal solution in terms of optimal solutions to smaller problems
   - Usually using min or max
3. Use a data structure (often a table) to store smaller solutions in a bottom-up fashion
   - Optimal value found in the table
4. (If needed) Reconstruct the optimal solution
   - I.e. what produced the optimal value

DYN. PROG. VS. DIVIDE AND CONQUER

- Remember D & C?
  - Divide into subproblems. Solve each. Combine.
- Good when subproblems do not overlap, when they're independent
  - No need to repeat them

- Divide and conquer: top-down
- Dynamic programming: bottom-up

LONGEST COMMON SUBSEQUENCE (LCS)

Problem: Given 2 sequences, $X = \langle x_1,...,x_m \rangle$ and $Y = \langle y_1,...,y_n \rangle$, find a common subsequence whose length is maximum.

Subsequence need not be consecutive, but must be in order.

OTHER SEQUENCE QUESTIONS

- ***Edit distance:*** Given 2 sequences, $X = \langle x_1,...,x_m \rangle$ and $Y = \langle y_1,...,y_n \rangle$, what is the minimum number of deletions, insertions, and changes that you must do to change one to another?
- ***Protein sequence alignment:*** Given a score matrix on amino acid pairs, $s(a,b)$ for $a,b \in \{\Lambda\} \cup A$, and 2 amino acid sequences, $X = \langle x_1,...,x_m \rangle \in A^m$ and $Y = \langle y_1,...,y_n \rangle \in A^n$, find the alignment with lowest score…

MORE PROBLEMS

Optimal BST: Given sequence $K = k_1 < k_2 < \cdots < k_n$ of n sorted keys, with a search probability p_i for each key k_i, build a binary search tree (BST) with minimum expected search cost.

Matrix chain multiplication: Given a sequence of matrices $A_1 A_2 \ldots A_n$, with A_i of dimension $m_i \times n_i$, insert parenthesis to minimize the total number of scalar multiplications.

Minimum convex decomposition of a polygon,

Hydrogen placement in protein structures, …

DYNAMIC PROGRAMMING

- Dynamic Programming is an algorithm design technique for optimization problems: often minimizing or maximizing.
- Like divide and conquer, DP solves problems by combining solutions to subproblems.
- Unlike divide and conquer, subproblems are not independent.
  - Subproblems may share subsubproblems,
  - However, solution to one subproblem may not affect the solutions to other subproblems of the same problem. (More on this later.)
- DP reduces computation by
  - Solving subproblems in a bottom-up fashion.
  - Storing solution to a subproblem the first time it is solved.
  - Looking up the solution when subproblem is encountered again.
- Key: determine structure of optimal solutions

STEPS IN DYNAMIC PROGRAMMING

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching or bottom-up in a table.
4. Construct an optimal solution from computed values.

LONGEST COMMON SUBSEQUENCE (LCS)

Problem: Given 2 sequences, $X = \langle x_1,...,x_m \rangle$ and $Y = \langle y_1,...,y_n \rangle$, find a common subsequence whose length is maximum.

Subsequence need not be consecutive, but must be in order.

NAÏVE ALGORITHM

- For every subsequence of X, check whether it's a subsequence of Y .
- Time: $\Theta(n2^m)$.
  - 2^m subsequences of X to check.
  - Each subsequence takes $\Theta(n)$ time to check: scan Y for first letter, for second, and so on.

OPTIMAL SUBSTRUCTURE

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y.
3. or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1}.

Notation:

prefix $X_i = \langle x_1, \ldots, x_i \rangle$ is the first i letters of X.

This says what any longest common subsequence must look like; do you believe it?

OPTIMAL SUBSTRUCTURE

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y.
3. or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1}.

Proof: (case 1: $x_m = y_n$)

Any sequence Z' that does not end in $x_m = y_n$ can be made longer by adding $x_m = y_n$ to the end. Therefore,

(1) longest common subsequence (LCS) Z must end in $x_m = y_n$.

(2) Z_{k-1} is a common subsequence of X_{m-1} and Y_{n-1}, and

(3) there is no longer CS of X_{m-1} and Y_{n-1}, or Z would not be an LCS.

OPTIMAL SUBSTRUCTURE

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y.
3. or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1}.

Proof: (case 2: $x_m \neq y_n$, and $z_k \neq x_m$)

Since Z does not end in x_m,

(1) Z is a common subsequence of X_{m-1} and Y, and

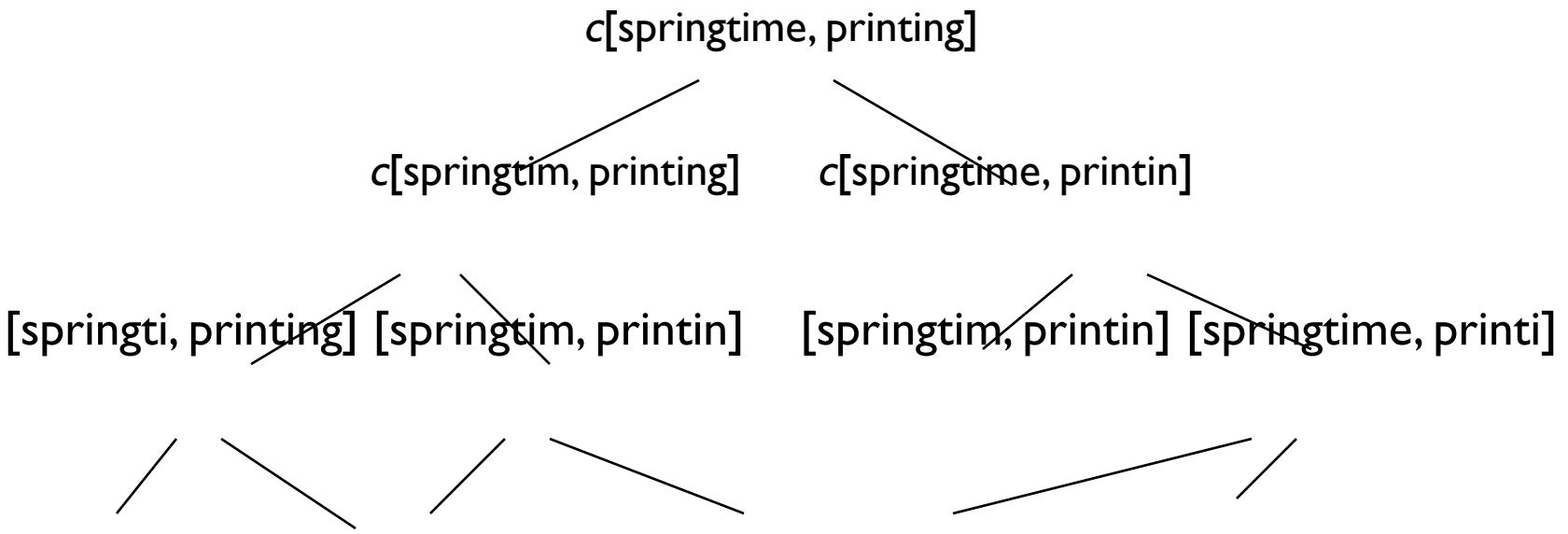
(2) there is no longer CS of X_{m-1} and Y, or Z would not be an LCS.

- Define $c[i, j]$ = length of LCS of X_i and Y_j.
- We want $c[m,n]$.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

This gives a recursive algorithm and solves the problem.
But does it solve it well?

$$c[\alpha,\beta] = \begin{cases} 0 & \text{if } \alpha \text{ empty or } \beta \text{ empty,} \\ c[prefix\alpha, prefix\beta]+1 & \text{if } \text{end}(\alpha) = \text{end}(\beta), \\ \max(c[prefix\alpha,\beta], c[\alpha, prefix\beta]) & \text{if } \text{end}(\alpha) \neq \text{end}(\beta). \end{cases}$$



LONGEST COMMON SUBSEQUENCE (LCS)

Application: comparison of two DNA strings

Ex: X= {A B C B D A B }, Y= {B D C A B A}

Longest Common Subsequence:

X = A **B** **C** **B** D **A** B

Y = **B** D **C** A **B** **A**

Brute force algorithm would compare each subsequence of X with the symbols in Y

LCS ALGORITHM

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i, Y_j to be the prefixes of X and Y of length i and j respectively
- Define $c[i,j]$ to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be $c[m,n]$

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

LCS RECURSIVE SOLUTION

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

- We start with $i = j = 0$ (empty substrings of x and y)
- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. $c[0,0] = 0$)
- LCS of empty string and any other string is empty, so for every i and j: $c[0, j] = c[i,0] = 0$

LCS RECURSIVE SOLUTION

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

- When we calculate *c[i,j],* we consider two cases:
- **First case:** *x[i]=y[j]*: one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{i-1} , plus 1