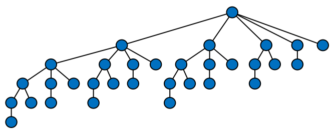
# **Lecture 18.** Dynamic Programming

**CpSc 8400: Algorithms and Data Structures**
**Brian C. Dean**



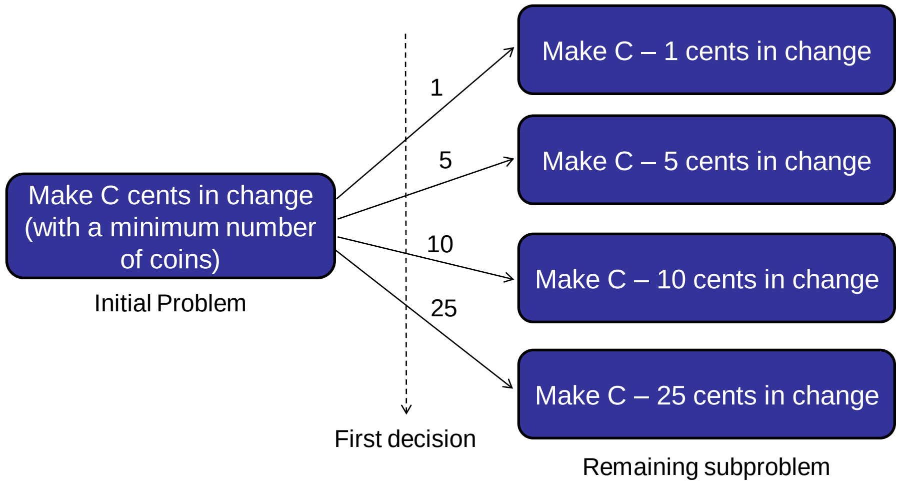
**School of Computing**
**Clemson University**
**Spring 2016**

## Making Change

- We have N different denominations of coins (e.g., 1 cent, 5 cent, 10 cent, 25 cent).
- We can use as many coins of each denomination as we wish.
- What is the minimum number of coins we need in order to construct exactly C cents worth of change?

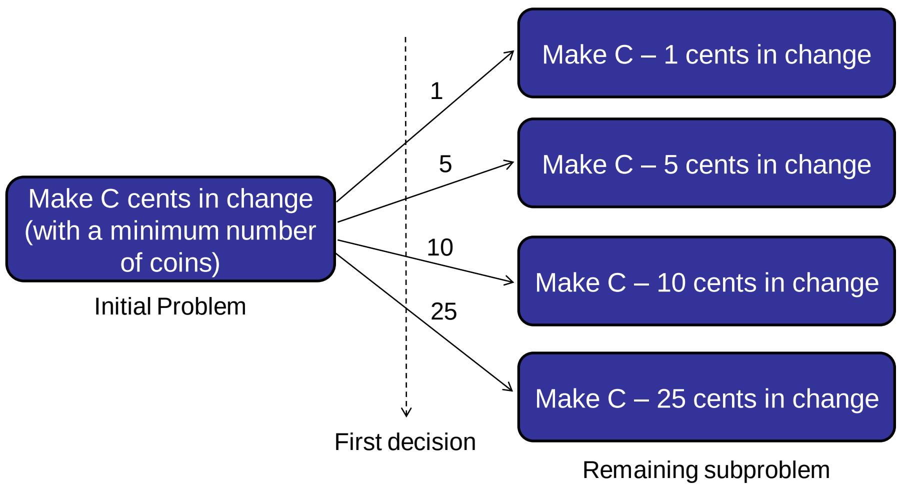
2

## Sequential Decisions…




3

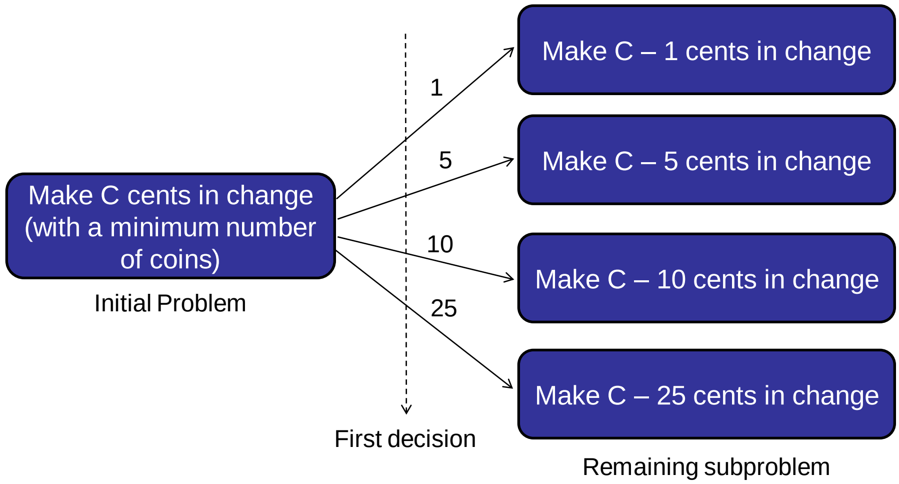
## Sequential Decisions…




**Greedy:** Initial decision can be made safely and irrevocably made according simple rule (e.g., use largest available coin).

4

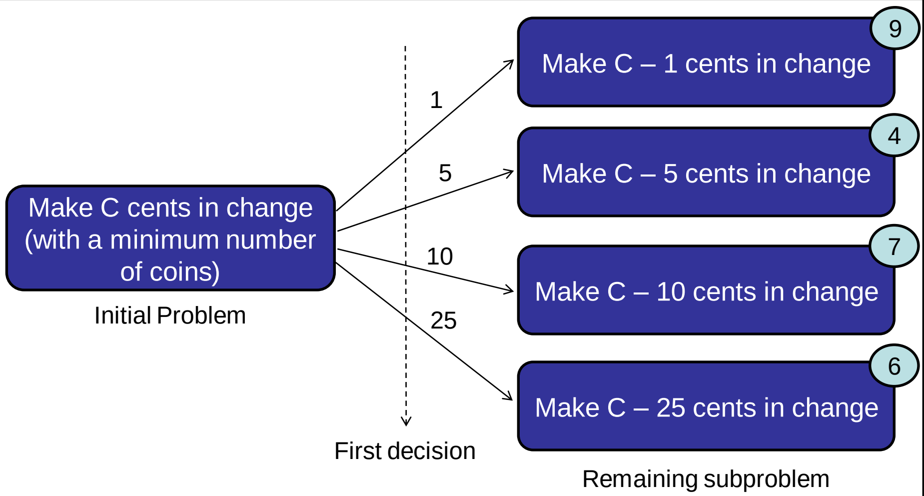
## Sequential Decisions…




**Exhaustive Search:** Recursively try all possible first decisions, usually in some sort of “greedy” order, usually with some sort of pruning heuristics to speed things up.

5

## Sequential Decisions…




**Dynamic Programming:** Solve all possible subproblems from smallest to largest, making decisions easy!

6

## Example: Activity Selection

- Given intervals $[a_i, b_i]$, select a disjoint subset of as many intervals as possible.

7

## Example: Activity Selection

- Given intervals $[a_i, b_i]$, select a disjoint subset of intervals of maximum total length.

8

## Example: Activity Selection

- Given intervals $[a_i, b_i]$, select a disjoint subset of intervals of maximum total length.
- Initially sort intervals so $a_1 \leq a_2 \leq \ldots \leq a_n$.
- Let $L[i]$ denote the value of an optimal solution for just the intervals $i \ldots n$.
- $L[i] = \max(L[i+1], (b_i - a_i) + \max \{L[j] : a_j \geq b_i\})$

9

## Example: Activity Selection

- Given intervals $[a_i, b_i]$, select a disjoint subset of intervals of maximum total length.
- Initially sort intervals so $a_1 \leq a_2 \leq \ldots \leq a_n$.
- Let $L[i]$ denote the value of an optimal solution for just the intervals $i \ldots n$.
- $L[i] = \max(\boxed{L[i+1]}, \boxed{(b_i - a_i) + \max \{L[j] : a_j \geq b_i\}})$

Optimal solution value if we decide not to include the $i^{th}$ interval.

Optimal solution value if we decide to include the $i^{th}$ interval.

10

## Example: Activity Selection

- Given intervals $[a_i, b_i]$, select a disjoint subset of intervals of maximum total length.
- Initially sort intervals so $a_1 \le a_2 \le \ldots \le a_n$.
- Let L[i] denote the value of an optimal solution for just the intervals i … n.
- $L[i] = \max(L[i+1], (b_i - a_i) + \max \{L[j] : a_j \ge b_i\})$
- We now have a simple $O(n^2)$ algorithm:
  - Compute L[n], L[n-1], …, L[1] in sequence according to the formula above.
  - L[1] tells us the **value** of an optimal solution.
  - What about the **intervals** in an optimal solution?

11

## The Dynamic Programming Technique

- Decompose problem into successively larger subproblems all of same form:

  "Let L[i] denote the value of an optimal solution for just the intervals i … n."
- Recursively express optimal solution to a large problem in terms of optimal solutions of smaller subproblems:

  "$L[i] = \max(L[i+1], (b_i - a_i) + \max \{L[j] : a_j \ge b_i\})$"
- Then just solve the problems in sequence from smallest to largest, building up a table of optimal solutions.

12

## Bottum-Up Versus Top-Down

- Initially: L[i] = undefined for all i = 1 .. n.
- Compute_L(i):

  If L[i] not undefined,
  Return L[i].
  Else,
  L[i] = max(Compute_L[i+1], $(b_i - a_i)$ + max {Compute_L[j] : $a_j \geq b_i$})
  Return L[i].

13

## Bottum-Up Versus Top-Down

- Initially: L[i] = undefined for all i = 1 .. n.
- Compute-L(i):

  If L[i] not undefined,
  Return L[i].
  Else,
  L[i] = max(Compute-L[i+1], $(b_i - a_i)$ + max {Compute-L[j] : $a_j \geq b_i$})
  Return L[i].
- Here we solve our subproblems in a top-down, recursive manner. If we aren't careful, this type of approach will take exponential time.
- However, since we store solutions to subproblems in a table once they're computed, we never solve the same subproblem more than once.
- Running time $O(n^2)$, just like the bottom-up variant.
- DP traditionally done bottom-up, but it's equivalent to do it top-down with "memoization" of solutions as we go.

14

## Example: Maximum-Value Subarray

- Given an array A[1..n], find a contiguous subarray A[i..j] that has maximum sum.
- Rather boring if all A[i]'s nonnegative, so assume some values are negative.

15

## Example: Maximum-Value Subarray

- Given an array A[1..n], find a contiguous subarray A[i..j] that has maximum sum.
- V[j] : sum of best subarray ending at index j.
- V[j] = max(A[j], V[j-1] + A[j])
- Simple O(n) algorithm: compute V[1], …, V[n] in sequence. Then take find $\max_j$ V[j].
- Note that we could have formulated it "backwards" (V[j] = opt subarray starting at j) like with the activity selection problem (or we could have formulated the activity selection algorithm "forwards"…)

16

## Example: Longest Increasing Subsequence

- Given an array A[1..n], what is the length of its longest increasing subsequence.
- E.g., 14 8 12 9 7 4 11 15 6 20 -3

17

## Example: Longest Increasing Subsequence

- Given an array A[1..n], what is the length of its longest increasing subsequence.
- E.g., 14 8 12 9 7 4 11 15 6 20 -3
- Let L[j] denote the length of the longest increasing subsequence ending at index j.
- $L[j] = \max_{\{i < j \,:\, A[i] \leq A[j]\}} \{L[i] + 1\}$
- To find the best subsequence overall, look for the maximum L[j] over all j = 1 .. n.
- The actual elements in this subsequence can be found by "tracing back" through the subproblem computation.

18

## Knapsack (Multiple Copies of Items Allowed)

- **Input**: n item types, with:
  - Sizes $s_1 \ldots s_n$ (all integer).
  - Values $v_1 \ldots v_n$.

  And also a capacity C knapsack.
- **Goal**: Find a maximum-value collection of items that fits in the knapsack. Multiple copies of items of the same type are allowed.
- We can solve this problem in O(nC) time using dynamic programming.



## Knapsack (Multiple Copies of Items Allowed)

- V[c] : optimal value we can pack into a knapsack of capacity c.
- $V[c] = \max(V[c-1], \max_{i=1..n}\{V[c-s_i] + v_i\})$.
  (as a base case, $V[c \leq 0] = 0$)
- Algorithm: compute V[1], V[2], …, V[C].
- At termination, V[C] contains the value of an optimal solution.
- How do we extract the set of items in an optimal solution?

## 0/1 Knapsack

- **Input**: n items, with:
  - Sizes $s_1 \ldots s_n$ (all integer).
  - Values $v_1 \ldots v_n$.

  And also a capacity C knapsack.
- **Goal**: Find a maximum-value collection of items that fits in the knapsack. Only one copy of each item allowed.
- We can also solve this problem in O(nC) time using dynamic programming, but we need to use a slightly different formulation…

21

## 0/1 Knapsack

- Subproblems of the form V[c] no longer work, since we can't keep track of which items we've already used.
- We need to use a "two-dimensional" state space of subproblems.
- V[j, c] : optimal value we can achieve by packing a subset of just items 1 … j into a capacity-c knapsack.
- $V[j, c] = \max(v_j + V[j-1, c - s_j], V[j-1, c])$

22

## Longest Common Subsequence

- Given two strings A[1..m] and B[1..n], what is their longest common subsequence?
- Example:

  A: **X**G**YZ**CDE**ZY**QW

  B: WQ**X**H**Y**BK**ZZ**L**Y**

23

## Longest Common Subsequence

- Given two strings A[1..m] and B[1..n], what is their longest common subsequence?
- Example:

  A: **X**G**YZ**CDE**ZY**QW

  B: WQ**X**H**Y**BK**ZZ**L**Y**

- We can find this in O(mn) time with a simple dynamic programming algorithm.
- L[i, j] : length of longest common subsequence of A[1..i] and B[1..j].
- If A[i] = B[j]: L[i, j] = 1 + L[i – 1, j – 1]
- If A[i] ≠ B[j]: L[i, j] = max(L[i – 1, j], L[i, j – 1])

24

## LCS Relatives

- The structure of the DP algorithm for longest common subsequences is the same as for many other useful problems:
  - Minimum edit distance : given a cost for deleting, inserting, and modifying a character, what is the minimum-cost transformation that takes string A to string B?
  - Optimal string alignment : given a similarity function between characters, what is the optimal way to "align" two strings A and B?

    Example: **`-SI-MILA-R`**

    **`ALIGN-MENT`**

25

## Matrix Chain Multiplication

- Problem: Compute the product of a sequence of rectangular matrices $M_1M_2\ldots M_n$, where $M_j$ has dimensions $a_j$ x $b_j$.
- Example: $M_1$ (1 x 100), $M_2$ (100 x 100), $M_3$ (100 x 100), $M_4$ (100 x 1).

  If we compute $M_1(M_2M_3)M_4$, this takes more than 1 million individual multiplications, but if we instead compute $(M_1M_2)(M_3M_4)$, this only requires 20100 individual multiplications!

26

## Matrix Chain Multiplication

- Problem: Compute the product of a sequence of rectangular matrices $M_1 M_2 \ldots M_n$, where $M_j$ has dimensions $a_j$ x $b_j$.
- $A[i, j]$ : minimum number of individual multiplications required to compute $M_i \ldots M_j$.
- $A[i, j] = \min_{k: i \le k < j} \{A[i, k] + a_k b_k b_{k+1} + A[k+1, j]\}$.

$$\underbrace{\left(M_i\ M_{i+1} \ldots M_{k-1}\ M_k\right)}_{A[i,k]} \bullet \underbrace{\left(M_{k+1}\ M_{k+2} \ldots M_{j-1}\ M_j\right)}_{A[k+1,j]}$$

27