Lecture 16
Dynamic Programming

CS 161 Design and Analysis of Algorithms

Ioannis Panageas

Dynamic Programming

Dynamic programming

- Avoids brute force search
- Somewhat Similar to D&C, but there are major differences
- Takes some practice to get used to

Note: This is difficult material. Readings:

- [GT]: Chapter 12
- [CLRS] Chapter 15
- [Kleinberg and Tardos], Chapter 6

Dynamic Programming vs. Recursion

- Dynamic programming be thought of as being the reverse of recursion
- Similar to D&C:
  - Is based on a recurrence
  - Obtains problem solution by using subproblem solutions
- Opposite of D&C:
  - Works from small problems to large problems
  - Motivated by recursion but does not actually use recursion
- Avoids redundantly solving the same subproblem multiple times by storing subproblem solutions
  - This requires careful indexing of subproblems

Dynamic Programming vs. Recursion

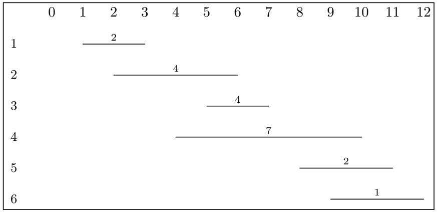
- Recursion is top-down
  Dynamic Programming (DP) is bottom-up
- Recursion solves all relevant subproblems
  DP may also solve some irrelevant subproblems
- Recursion may solve some subproblems many times
  DP solves each subproblem only once

	D&C / Recursion	Memoized Recursion	Dynamic Programming
Basic approach	recursion	recursion	iteration
Use of recurrence	top-down	top-down	bottom-up
Store subproblem solutions	No	Yes	Yes
Space needed for stack	Yes	Yes	No

Problem: Weighted interval scheduling

- Input: Collection of n Intervals represented by Start Time, Finish Time, and Value: $(s(j), f(j), v(j))$.
- Problem: Find a non-overlapping set of intervals that maximizes the total value.
- Example:

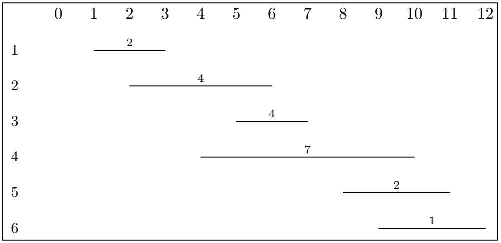
j	$s(j)$	$f(j)$	$v(j)$
1	1	3	2
2	2	6	4
3	5	7	4
4	4	10	7
5	8	11	2
6	9	12	1



Weighted interval scheduling problem: Preprocessing

1. Sort the intervals by finishing time. (Here they are already sorted).
2. For each interval j, define $p(j)$ to be:
   - The highest-numbered interval $i < j$ that does not overlap interval j (if such an interval exists)
   - 0 (if no such an interval exists)

j	$s(j)$	$f(j)$	$v(j)$	$p(j)$
1	1	3	2	0
2	2	6	4	0
3	5	7	4	1
4	4	10	7	1
5	8	11	2	3
6	9	12	1	3



Simple recursive algorithm

For the problem on intervals 1 through j:

- Either the optimal solution contains the last interval or it doesn't
- If it does:
  - The optimal value is $v(j)$ plus the value of the optimal collection from $1, \ldots, p(j)$
- If it does not:
  - optimal value is the value of the optimal collection from $1, \ldots, j-1$
- So the optimal value is the maximum of these two possible values:

```
def OPT(j):
    if j = 0:  return 0
    else:  return max(v(j)+OPT(p(j)), OPT(j-1))
```

Correct, but very inefficient because ...

- the same value of OPT() is recomputed multiple times.

Memoizing the recursion

- We can avoid recomputing OPT() values by storing them
  - So we just look up a previously computed value rather than recomputing it
- Declare an array $M[1..n]$, where each entry can contain an integer or "undefined"
- Initialize all entries to “undefined”

```
def Memoized_OPT(j):
   if j = 0:  return(0);
   else:
      if M[j] = "undefined" :
         M[j] = max(v(j)+Memoized_OPT(p(j)), Memoized_OPT(j-1))
      return (M[j])
```

Analysis of Memoized Algorithm

```
def Memoized_OPT(j):
   if j = 0:  return(0);
   else:
      if M[j] = "undefined" :
         M[j] = max(v(j)+Memoized_OPT(p(j)), Memoized_OPT(j-1))
      return (M[j])

Memoized_OPT(n)
```

Run `Memoized_OPT` on a collection of n intervals:

- For every pair of recursive calls, an entry of M gets filled in.
- Hence, $O(n)$ calls.

Dynamic Programming Solution

- In memoized recursion, we entered a value in the M array based on values that appear earlier in that array
- Instead of computing the entries in array M recursively, we can:
  - Get rid of the recursion entirely
  - Compute the array entries iteratively
- This is the dynamic programming solution.

```
def Iterative_OPT:
   M[0] = 0
   for j = 1 to n:
      M[j] = max(v(j)+M[p(j)],M[j-1])
```

- Simple, efficient code.
- Runs in $O(n)$ time.

Computing the Optimal Set of Intervals

```
def Iterative_OPT:
   M[0] = 0
   for j = 1 to n:
      M[j] = max(v(j)+M[p(j)],M[j-1])
```

- There is one issue here:
  - The algorithm given above computes the value of an optimal interval set, but not the intervals themselves.
- This is a standard with dynamic programming problems. We usually proceed in two steps.
  1. We first consider how to compute the optimum cost or value
  2. Once we know how to compute the optimum cost or value, we then consider how to compute a configuration that has the optimum cost or value.
- Compute additional information (usually an additional array) as we compute the optimum cost or value.
- Run a post-processing step that uses this additional information

Computing the Optimal Set of Intervals

- For each *j*, we remember whether the optimal set for the first *j* intervals contains interval *j*.
- We compute two arrays:
  - `M[j]` stores the best value we can get from the first *j* intervals (as before)
  - `keep[j]` stores whether the best choice for the first *j* intervals includes interval *j*

```
def Iterative_OPT:
   M[0] = 0
   for j = 1 to n:
      if v(j)+M[p(j)] > M[j-1]:
         M[j] = v(j)+M[p(j)]
         keep[j] = True
      else:
         M[j] = M[j-1]
         keep[j] = False
```

Computing the Optimal Set of Intervals, continued

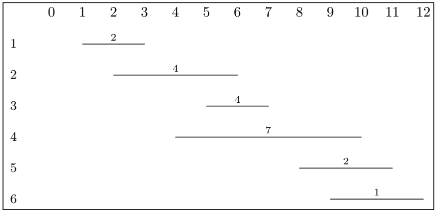
Once we have computed the two arrays `M[ ]` and `keep[ ]`:

```
def PrintSolution(j):
   if j = 0:  return;
   if keep[j]:
      PrintSolution(p(j))
      print(j)
   else:
      PrintSolution(j-1)

PrintSolution(n)
```

Our example

j	$s(j)$	$f(j)$	$v(j)$	$p(j)$
1	1	3	2	0
2	2	6	4	0
3	5	7	4	1
4	4	10	7	1
5	8	11	2	3
6	9	12	1	3



	0	1	2	3	4	5	6
`M:`	0	2	4	6	9	9	9
`keep:`		T	T	T	T	F	F

Selected intervals: $\{1, 4\}$.

The array M contains the solutions of the subproblems. We will refer to this as the memoization table

Principles of Dynamic Programming

Dynamic programming can be applied when there is a set of subproblems derived from the original subproblem such that:

- There are only a polynomial number of subproblems
- The solution to the original problem can be easily computed from the solution to the subproblems.
  - For example, when the original problem *is* one of the subproblems
- There is
  - An ordering on the subproblems, together with
  - A recurrence on subproblem solution that enable the solution to any subproblem P to be computed from the solutions to some of the subproblems that precede P in the ordering

We saw this in the case of the weighted interval scheduling problem.

Specifying a Dynamic Programming Solution

The solution to a Dynamic Programming Solution is specified by writing:

1. The subproblem domain: the set of indices of the subproblems.
2. A precise definition of of what the function mapping each subproblem to its solution represents. (Equivalently, a precise definition of what each entry in the memoization table represents.)
3. The goal: the solution to the original problem, expressed in terms of certain values of the function from item #2.
4. The initial value(s) / condition(s): values of the function from item #2 for small subproblems that do not need to be decomposed further.
5. The recurrence: a formula describing how to compute the solution of a subproblem from the solutions to smaller subproblems.

Here, "smaller" means "earlier in the ordering"

Solution to Weighted-Interval Scheduling

1. Subproblem domain: $\{0, \ldots, n\}$
2. Function / Memoization table definition: $M(j)$ is the maximum value that can be obtained from a set of non-overlapping intervals with indices in the range $\{1, \ldots, j\}$
3. Goal: $M(n)$
4. Initial value: $M(0) = 0$
5. Recurrence: $M(j) = \max(v(j) + M(p(j)), M(j-1))$ for $j \geq 1$. Here, $p(j)$ is a precomputed function defined by

$$p(j) = \begin{cases} \text{The highest-numbered interval } i < j \text{ that does not} \\ \text{overlap interval } j \text{ if such an interval exists} \\ 0 \text{ otherwise} \end{cases}$$

Truck loading problem

(Note: This problem is usually called the subset-sum problem, but sometimes that name is used for a different problem. Here we call it the truck-loading problem.)

Problem definition:

- Truck has weight limit of W.
- n boxes. Box i has weight w_i.
- We want to load the truck to carry the maximum weight possible, subject to the weight restriction.

Greedy heuristics don't work

1. Heaviest boxes first:
$$W = 100,\ w_1 = 51,\ w_2 = 50,\ w_3 = 50$$
2. Lightest boxes first:
$$W = 100,\ w_1 = 1,\ w_2 = 50,\ w_3 = 50$$

Dynamic Programming Solution: Basic Idea

Suppose we have i boxes and a truck with weight capacity j.

- Either the optimum solution contains the last box or it doesn't.
- If the optimum solution contains the last box:
  - The optimum value is w_i plus the optimum value we can get by fitting the first $i-1$ boxes on the truck, after accounting for the weight taken up by box i.
- If the optimum solution does not contains the last box:
  - The optimum value is the optimum value we can get from the first $i-1$ boxes.

We will express this more formally on the next slide.

Solution: Expressed as recurrence equation

- Let OPT(i, j) be the maximum weight we can get by loading from boxes 1 through i, up to the weight limit j.
- Applying what we said on the previous slide:

$$\text{OPT}(i, j) = \max\left(w_i + \text{OPT}(i-1, j-w_i), \text{OPT}(i-1, j)\right)$$

- Note that if $w_i > j$, we can't use box i, so only the second choice is available.
- This recurrence equation gives us the dynamic programming solution (specified on next slide)

Specifying the Solution

1. Subproblem domain: $\{0,\ldots,n\} \times \{0,\ldots,W\}$
2. Function / Memoization table definition: $\text{OPT}(i,j)$ is the value of the best way of loading a subset of the first i boxes into a truck with maximum capacity j.
3. Goal: $\text{OPT}(n,W)$
4. Initial values:

$$\begin{array}{rcll} \text{OPT}(i,0) & = & 0 & \text{for all } i \geq 0 \\ \text{OPT}(0,j) & = & 0 & \text{for all } j \geq 0 \end{array}$$

5. Recurrence:

$$\text{OPT}(i,j) = \begin{cases} \max\left(w_i + \text{OPT}(i-1, j-w_i), \text{OPT}(i-1,j)\right) & \text{if } w_i \leq j \\ \text{OPT}(i-1,j) & \text{if } w_i > j \end{cases}$$

Truck Loading Problem DP Pseudocode: compute OPT Matrix

```
def compute_opt_matrix(w):
  for i = 0 to n:  OPT[i,0] = 0
  for j = 0 to W: OPT[0,j] = 0
  for i = 1 to n:
     for j = 1 to W:
        if w[i] > j:
           OPT[i,j] = OPT[i-1,j]
        else:
           OPT[i,j] = max(w[i] + OPT[i-1,j-w[i]], OPT[i-1,j])
  return OPT
```

This tells us the maximum possible weight, but we need to also compute which boxes to load to achieve this maximum weight . . .

Truck Loading Problem DP Pseudocode: compute choice of boxes

Introduce an new array `keep[i,j]`, which tells us whether we keep box i when we solve the subproblem with i boxes and capacity j.

```
def compute_opt_strategy(w):
  for i = 0 to n:  OPT[i,0] = 0
  for j = 0 to W: OPT[0,j] = 0
  for i = 1 to n:
     for j = 1 to W:
        if (w[i] > j) or (w[i] + OPT[i-1,j-w[i]] <= OPT[i-1,j])
           OPT[i,j] = OPT[i-1,j]
           keep[i,j] = False
        else:
           OPT[i,j] = w[i] + OPT[i-1,j-w[i]]
           keep[i,j] = True
     return (OPT,keep)
```

Running time: $O(n \cdot W)$

Truck Loading Problem DP Pseudocode: compute choice of boxes [continued]

```
def print_solution(OPT,keep,i,j):
   if i == 0:  return
   if keep[i,j]:
      print_solution(OPT,keep,i-1,j-w[i])
      print (i)
   else:
      print_solution(OPT,keep,i-1,j)

// Main program starts here
(OPT,keep) = compute_opt_strategy(w)
print_solution(OPT,keep,n,W)
```

0/1 Knapsack Problem

- We have a knapsack with limited capacity. We need to decide which items to put in the knapsack.
- There are n items: item i has weight w_i, value v_i.
- Knapsack can handle a total weight of at most W.
- We want to put in items with maximum total value, subject to the weight restriction.
- We can put all of an item in the knapsack, or none of it (fractional items have no value.)
- Recall: If fractional items can be taken, greedy heuristic works:
  - Order items according to value per unit weight.
  - This does not work if we can only take whole items.
  - Example:
    - $W = 100$
    - Item 1: $w_1 = 20$, $v_1 = 80$
    - Item 2: $w_2 = 90$, $v_2 = 90$.

Dynamic Programming Solution

- Very similar to truck loading problem.
- Let OPT(i, j) be the value of the best way to load the first i items, using a knapsack with maximum capacity j.
- If we optimally load i items using maximum capacity j either we include item i or we don't. So:

$$\mathrm{OPT}(i,j) = \max\left(v_i + \mathrm{OPT}(i-1, j-w_i), \mathrm{OPT}(i-1,j)\right);$$

- If $w_i > j$, we can't use item i, so only the second choice is available.

Specifying the Solution

1. Subprogram domain $\{0, \ldots, n\} \times \{0, \ldots, W\}$
2. Function /Memoization table definition: OPT(i, j) is the value of the best way of loading a subset of the first i items into a knapsack with maximum capacity j.
3. Goal: OPT(n, W)
4. Initial values:

$$\begin{array}{rcll} \mathrm{OPT}(i,0) & = & 0 & \text{for all } i \geq 0 \\ \mathrm{OPT}(0,j) & = & 0 & \text{for all } j \geq 0 \end{array}$$

5. Recurrence:

$$\mathrm{OPT}(i,j) = \begin{cases} \max\left(v_i + \mathrm{OPT}(i-1, j-w_i), \mathrm{OPT}(i-1,j)\right) & \text{if } w_i \leq j \\ \mathrm{OPT}(i-1,j) & \text{if } w_i > j \end{cases}$$

Pseudocode for DP Solution to 0/1 Knapsack Problem

```
def compute_opt_strategy(w,v):
  for i = 0 to n:  OPT[i,0] = 0
  for j = 0 to W: OPT[0,j] = 0
  for i = 1 to n:
     for j = 1 to W:
        if (w[i] > j) or (v[i] + OPT[i-1,j-w[i]] <= OPT[i-1,j])
           OPT[i,j] = OPT[i-1,j]
           keep[i,j] = False
        else:
           OPT[i,j] = v[i] + OPT[i-1,j-w[i]]
           keep[i,j] = True
     return (OPT,keep)
```

Pseudocode for DP Solution to 0/1 Knapsack Problem [continued]

```
def print_solution(OPT,keep,i,j):
   if i == 0:  return
   if keep[i,j]:
      print_solution(OPT,keep,i-1,j-w[i])
      print (i)
   else:
      print_solution(OPT,keep,i-1,j)

// Main program starts here
(OPT,keep) = compute_opt_strategy(w,v)
print_solution(OPT,keep,n,W)
```

Optimal Matrix Chain Multiplication

Some facts about matrix multiplication:

1. Multiplying a $p \times q$ matrix by a $q \times r$ matrix requires $p \cdot q \cdot r$ multiplications. (Because the product will be $p \times r$, and the computation of each entry requires q scalar multiplications).
2. Matrix multiplication is associative:

$$(A \times B) \times C = A \times (B \times C)$$

3. The parenthesizing may effect the efficiency.

A:	$p \times q$	$A \times B$:	$p \times r$
B:	$q \times r$	$B \times C$:	$q \times s$
C:	$r \times s$		

$(A \times B) \times C$: Number of scalar multiplications is:

$$p \cdot q \cdot r + p \cdot r \cdot s$$

$A \times (B \times C)$: Number of scalar multiplications is:

$$q \cdot r \cdot s + p \cdot q \cdot s$$

Example

Suppose A is 40×2, B is 2×100, and C is 100×50.

- $(A \times B) \times C$: Cost is

$$40 \cdot 2 \cdot 100 + 40 \cdot 100 \cdot 50 = 8,000 + 200,000 = 208,000$$

- $A \times (B \times C)$: Cost is

$$2 \cdot 100 \cdot 50 + 40 \cdot 2 \cdot 50 = 10,000 + 4,000 = 14,000$$

$A \times (B \times C)$ is considerably more efficient

Parenthesization Matters

Optimal Matrix Chain Multiplication problem

- Given n matrices: $A_1, \ldots, A_n$.
- Matrix A_i is $d_{i-1} \times d_i$.
- What is the most efficient way of grouping (i.e.,parenthesizing) to compute $A_1 \times \cdots \times A_n$?
  - Most efficient means fewest scalar multiplications

Example:

A_1	:	10×15
A_2	:	15×5
A_3	:	5×60
A_4	:	60×100
A_5	:	100×20
A_6	:	20×40
A_7	:	40×47

d_0	$=10$
d_1	$=15$
d_2	$=5$
d_3	$=60$
d_4	$=100$
d_5	$=20$
d_6	$=40$
d_7	$=47$

- As we will see, the optimal cost is 56,500 scalar multiplications
- The optimal grouping is:

$$(A_1 \times A_2) \times ((((A_3 \times A_4) \times A_5) \times A_6) \times A_7)$$

Dynamic Programming Solution

- Subproblems: optimally multiplying chains $A_i \times \cdots \times A_j$
- Define $M(i,j) =$ the number of multiplications required to compute the product $A_i \times \cdots \times A_j$ using the best possible grouping
- The final multiplication will consist of a left subchain and a right subchain.
- Suppose the left subchain stops at A_k: $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$
  - The cost of computing the left subchain is $M(i, k)$
  - The cost of computing the right subchain is $M(k + 1, j)$
  - Cost of final multiplication:
    - $(A_i \times \cdots \times A_k)$ is $d_{i-1} \times d_k$
    - $(A_{k+1} \times \cdots \times A_j)$ is $d_k \times d_j$
    - Cost of multiplication is $d_{i-1}d_kd_j$
  - Total cost is $M(i, k) + M(k + 1, j) + d_{i-1}d_kd_j$.
- Choose the best index k:

$$M(i,j) = \min_{i \le k \le j-1} (M(i, k) + M(k + 1, j) + d_{i-1}d_kd_j)$$

Specifying the Solution

1. Subproblem domain $\{(i,j) : 1 \le i \le j \le n\}$
2. Function / Memoization table definition: $M(i,j)$ is the minimum number of multiplications required to compute the product $A_i \times \cdots \times A_j$ (using the best possible grouping).
3. Goal: $M(1,n)$
4. Initial values: $M(i,i) = 0$
5. Recurrence:

$$M(i,j) = \min_{i \le k \le j-1} \left(M(i,k) + M(k+1,j) + d_{i-1} d_k d_j \right)$$

Note:

- The fact that $M(i,i+1) = d_{i-1} d_i d_{i+1}$ does not need to be stated as an initial condition.
- It follows from the recurrence equation that

$$\begin{aligned} M(i,i+1) &= M(i,i) + M(i+1,i+1) + d_{i-1} d_i d_{i+1} \\ &= 0 + 0 + d_{i-1} d_i d_{i+1} \\ &= d_{i-1} d_i d_{i+1}. \end{aligned}$$

Pseudocode

- The input is just the array of dimensions: $d_0, \ldots, d_n$.
- We need to compute the chain costs in increasing order of the chain lengths. (The length of the chain $A_i \times \cdots \times A_j$ is $j - i + 1$.)

```
def optMatrixChain(d):
    for i = 1 to n:
        M[i,i] = 0
    for len = 2 to n:
        for i = 1 to n - len + 1:
            j = i + len - 1
            M[i,j] = +∞
            for k = i to j-1:
                x = M[i,k] + M[k+1,j] + d[i-1]*d[k]*d[j]
                if x < M[i,j]:
                    M[i,j] = x
    return M
```

Computing the chains

- Augment the preceding pseudocode by storing the best split for each (i, j) in an array S.
- $S[i, j] = k$ when the best split for $A_i \times \cdots \times A_j$ is

$$(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$$

```
def optMatrixChain(d):
    for i = 1 to n:
        M[i,i] = 0
    for len = 2 to n:
        for i = 1 to n - len + 1:
            j = i + len - 1
            M[i,j] = +∞
            for k = i to j-1:
                x = M[i,k] + M[k+1,j] + d[i-1]*d[k]*d[j]
                if x < M[i,j]:
                    M[i,j] = x
                    S[i,j] = k
    return M,S
```

Solution to our example

$A_1 : 10 \times 15$
$A_2 : 15 \times 5$
$A_3 : 5 \times 60$
$A_4 : 60 \times 100$
$A_5 : 100 \times 20$
$A_6 : 20 \times 40$
$A_7 : 40 \times 47$

$d_0 = 10$
$d_1 = 15$
$d_2 = 5$
$d_3 = 60$
$d_4 = 100$
$d_5 = 20$
$d_6 = 40$
$d_7 = 47$

j

1	2	3	4	5	6	7		i
0 —	750 1	3750 2	35750 2	41750 2	46750 2	56500 2	1	
	0 —	4500 2	37500 2	41500 2	47000 2	56925 2	2	
		0 —	30000 3	40000 4	44000 5	53400 6	3	
			0 —	120000 4	168000 5	214000 5	4	
				0 —	80000 5	131600 5	5	
					0 —	37600 6	6	
						0 —	7	

Optimal value is 56500

Optimal grouping is:

$$(A_1 \times A_2) \times ((((A_3 \times A_4) \times A_5) \times A_6) \times A_7)$$