中山大學
SUN YAT-SEN UNIVERSITY

Lecture 6
Dynamic Programming
Part II

Algorithm Design

zhangzizhen@gmail.com

QQ group: 117282780

Longest Non-Decreasing Subsequence

- The longest non-decreasing subsequence (**LNDS**) problem is to find a subsequence of a given sequence in which the subsequence's elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible. This subsequence is not necessarily contiguous, or unique.
- Example: Consider the following sequence

  [1, 2, 5, 2, 8, 6, 3, 6, 9, 7]

  [1, 5, 8, 9] forms a non-decreasing subsequence, so does

  [1, 2, 2, 6, 6, 7] but it is longer.

Longest Non-Decreasing Subsequence

- Solve subproblem on $s_1,...,s_{n-1}$ and then try to extend using s_n.
- Two cases:
  - s_n is not used, answer is the same answer as on $s_1,...,s_{n-1}$.
  - s_n is used, answer is s_n preceded by the longest increasing subsequence in $s_1,...,s_{n-1}$ that ends in a number smaller than s_n.
- Recurrence:
  - Let L[i] be the length of longest non-decreasing subsequence in $s_1,...,s_n$ that ends in s_i.
  - $L[j]=1+\max\{L[i]: i<j \text{ and } s_i<=s_j\}$
  - L[0]=0
  - Length of longest increasing subsequence:

    $\max\{L[i]: 1\leq i \leq n\}$

Longest Non-Decreasing Subsequence

- We also maintain P[j] to be the value of i that achieved the max L[j].
  - This will be the index of the predecessor of s_j in a longest increasing subsequence that ends in s_j.
  - By following the P[j] values we can reconstruct the whole sequence in linear time.
- Implementation: $O(n^2)$

```
for (j = 1; j <= n; j++) {
        L[j] = 1;
        P[j] = 0;
        for (i = 1; i < j; i++)
        if (s[i]<=s[j] && L[i]+1>L[j]) {
                P[j] = i;
                L[j] = L[i]+1;
        }
}
```

Longest Non-Decreasing Subsequence

- Exercise

index	1	2	3	4	5	6	7	8	9	10
sequence	1	2	5	2	8	6	3	6	9	7
L[i]										
P[i]										

index	1	2	3	4	5	6	7	8	9	10
sequence	1	2	5	2	8	6	3	6	9	7
L[i]	1	2	3	3	4	4	4	5	6	6
P[i]	0	1	2	2	3	3	4	6	8	8

Longest Non-Decreasing Subsequence

- Improvement

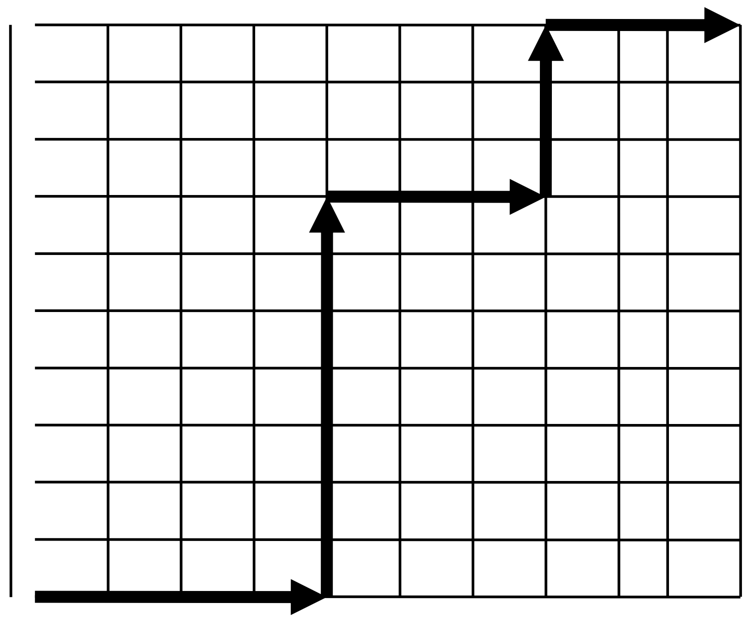
L[i]	index set	element set	min. element
0	$\emptyset$	$\emptyset$	-
1	$\{1\}$	$\{1\}$	1
2	$\{2\}$	$\{2\}$	2
3	$\{3, 4\}$	$\{2, 5\}$	2
4	$\{5, 6, 7\}$	$\{3, 6, 8\}$	3
5	$\{8\}$	$\{6\}$	6
6	$\{9, 10\}$	$\{7, 9\}$	7

Non-decreasing Order

- Maintain the min.element array, which is a sorted array
- Use binary search to update the table
- O(n logn)

Street walking

- 一个城市的街道布局如下，从最左下方走到最右上方，每次只能往上或往右走，一共有多少种走法?



Street walking

- **Suppose that there are n horizontal streets and m vertical streets. We set a coordinate for each cross point. The left-bottom point is set to (0,0) and the right-upper point is set to (10,10).**
- 不难看出子问题就是：从（**0**，**0**）走到（**x**，**y**），每次只能往上或往右走，一共有多少种走法，将这个走法数记为**f(x, y)**，原问题就是求**f(10, 10)**。
- 走到（**x**，**y**）有两个方法**,** 一个是从（**x**—**1**，**y**）往右走**1**步**,** 另一个是从（**x**，**y**—**1**）往上走**1**步**,** 前者有**f(x-1, y)**种方法**,** 后者有**f(x, y-1)**种方法**,** 所以**:**

  f(x, y)=f(x-1, y)+f(x, y-1), 另外当**x**或**y**为**0**的时候，

  明显**f(x, y)=1**，即：
- 当 **x=0** 或 **y =0** 时， **f(x, y)=1**
- 当 **x>0** 且 **y>0** 时， **f(x, y) = f(x-1,y) + f(x, y-1)**

Street walking

```
// 先置初始解
for (i = 0; i <= 10; i++) {
    f[i][0] = 1;
    f[0][i] = 1;
}
// 递推的求解各个子问题
for (i = 1; i <= 10; i++) {
    for (j = 1; j <= 10; j++) {
        f[i][j] = f[i-1][j] + f[i][j-1];
    }
}
// 输出解
cout<<f[10][10]<<endl;
```

Matrix-Chain Multiplication

- Let A be an n by m matrix, let B be an m by p matrix, then $C = AB$ is an n by p matrix.
- $C = AB$ can be computed in O(nmp) time, using traditional matrix multiplication.
- Suppose we want to compute $A_1A_2A_3A_4$.
- Matrix Multiplication is associative, so we can do the multiplication in several different orders.
- Given n matrices, the size of the matrix A_i is $p_{i-1}*p_i$, find the minimum multiplication operations.

Matrix-Chain Multiplication

- Example:

  A_1 is 10 by 100 matrix

  A_2 is 100 by 5 matrix

  A_3 is 5 by 50 matrix

  A_4 is 50 by 1 matrix

  A_1A_2A3A4 is a 10 by 1 matrix

- 5 different orderings = 5 different parenthesizations

  $(A_1(A_2(A_3A_4)))$

  $((A_1A_2)(A_3A_4))$

  $(((A_1A_2)A_3)A_4)$

  $((A_1(A_2A_3))A_4)$

  $(A_1((A_2A_3)A_4))$

Matrix-Chain Multiplication

- $(A_1(A_2 (A_3A_4)))$
  - $A_{34} = A_3A_4$, 250 mults, result is 5 by 1
  - $A_{24} = A_2A_{34}$, 500 mults, result is 100 by 1
  - $A_{14} = A_1A_{24}$, 1000 mults, result is 10 by 1
  - Total is 1750
- $((A_1A_2)(A_3A_4))$
  - $A_{12} = A_1A_2$, 5000 mults, result is 10 by 5
  - $A_{34} = A_3A_4$, 250 mults, result is 5 by 1
  - $A_{14} = A_{12}A_{34}$, 50 mults, result is 10 by 1
  - Total is 5300
- $(((A_1A_2)A_3)A_4)$
  - $A_{12} = A_1A_2$, 5000 mults, result is 10 by 5
  - $A_{13} = A_{12}A_3$, 2500 mults, result is 10 by 50
  - $A_{14} = A_{13}A_4$, 500 mults, results is 10 by 1
  - Total is 8000

Matrix-Chain Multiplication

- $((A_1(A_2A_3))A_4)$
  - $A_{23} = A_2A_3$, 25000 mults, result is 100 by 50
  - $A_{13} = A_1A_{23}$, 50000 mults, result is 10 by 50
  - $A_{14} = A_{13}A_4$, 500 mults, results is 10 by 1
  - Total is 75500
- $(A_1 ((A_2A_3)A_4))$
  - $A_{23} = A_2A_3$, 25000 mults, result is 100 by 50
  - $A_{24} = A_{23}A_4$, 5000 mults, result is 100 by 1
  - $A_{14} = A_1A_{24}$, 1000 mults, result is 10 by 1
  - Total is 31000
- Conclusion: Order of operations makes a huge difference. How do we compute the minimum?

Matrix-Chain Multiplication

- **Parenthesization**: A product of matrices is fully parenthesized if it is either
  - a single matrix, or
  - a product of two fully parenthesized matrices, surrounded by parentheses
- Each parenthesization defines a set of n-1 matrix multiplications. We just need to pick the parenthesization that corresponds to the best ordering.
- Question: How many parenthesizations are there?

Matrix-Chain Multiplication

- Let *P(n)* be the number of ways to parenthesize *n* matrices.

$$P(n) = \begin{cases} \Sigma_{k=1}^{n-1} P(k)P(n-k) & \textbf{if } n \geq 2 \\ 1 & \textbf{if } n = 1 \end{cases}$$

- This recurrence is related to the Catalan numbers

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

- Asymptotically, the Catalan numbers grow as

$$C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

Matrix-Chain Multiplication

- Structure of an optimal solution: If the outermost parenthesization is

  $((A_1A_2 \cdot \cdot \cdot A_i)(A_{i+1} \cdot \cdot \cdot A_n))$

  then the optimal solution consists of solving $A_{1,i}$ and $A_{i+1,n}$ optimally and then combining the solutions.

- Overlapping subproblems: In the enumeration of the $P(n) = \Omega(4^n/n^{3/2})$ subproblems, how many unique subproblems are there?

Recursive solution

- A subproblem is of the form A_{ij} with $1<=i<=j<=n$, so there are O(n^2) subproblems.
- Let A_i be p_{i-1} by p_i .
- Let $m[i, j]$ be the cost of computing A_{ij}
- If the final multiplication for A_{ij} is $A_{ij} = A_{ik}A_{k+1,j}$ then $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$.
- We don't know k a priori, so we take the minimum

$$m[i,j] = \begin{cases} 0 \quad \text{if } i = j \ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} \quad \text{if } i < j \end{cases}$$

- Direct recursion on this does not work! We must use the fact that there are at most O(n^2) different calls. What is the order?

0-1 Knapsack Problem

- Given *n* objects and a "knapsack."
- Item *i* weighs $w_i > 0$ and has value $v_i > 0$.
- Knapsack has a capacity of *W*.
- Goal: fill knapsack so as to maximize total value.
- Example:

  { 1, 2, 5 } has value 35.

  { 3, 4 } has value 40.

  { 3, 5 } has value 46,

  (but exceeds weight limit).

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance
(weight limit W = 11)

0-1 Knapsack Problem

- Let *OPT(i)* = max profit subset of items *1, …, i.*
- Case 1. *OPT* does not select item *i.*
  - *OPT* selects best of *{ 1, 2, …, i – 1 }.*
- Case 2. *OPT* selects item *i.*
  - Selecting item *i* does not immediately imply that we will have to reject other items.
  - Without knowing what other items were selected before *i,* we don't even know if we have enough room for *i.*
- Conclusion: Need more subproblems!

0-1 Knapsack Problem

- Let *OPT(i, w)* = max profit subset of items *1, …, i* with weight limit *w.*
- Case 1. OPT does not select item *i.*
  - *OPT* selects best of *{ 1, 2, …, i – 1 }* using weight limit *w.*
- Case 2. *OPT* selects item *i.*
  - New weight limit = $w - w_i$.
  - *OPT* selects best of *{ 1, 2, …, i – 1 }* using this new weight limit.

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1, w-w_i) \,\} & \text{otherwise} \end{cases}$$

0-1 Knapsack Problem

- Implementation: O(nW)

```
for (w = 0; w <= W; w++)
        M[0, w] = 0;
for (i = 1; i <= n; i++)
for (w = 1; w <= W; w++)
        if (wt[i] > w) M[i, w] = M[i – 1, w];
        else M[i, w] = max { M[i – 1, w], v[i] + M[i – 1, w – wt[i]] };
return M[n, W];
```

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance
(weight limit W = 11)

weight limit w

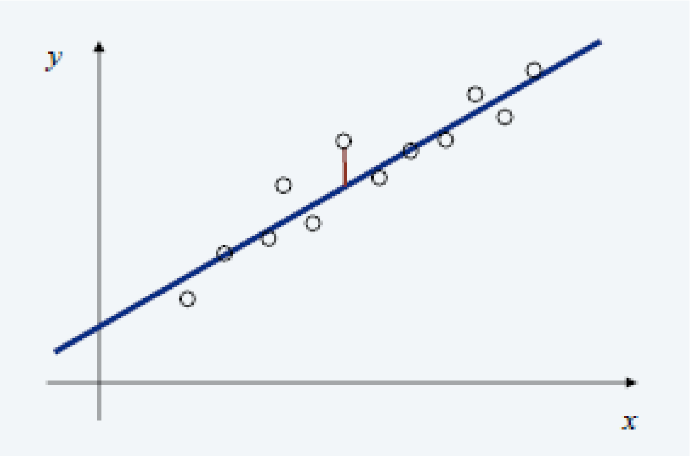
subset of items 1, …, i	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

Segmented Least Squares

- Least squares. Foundational problem in statistics.
- Given *n* points in the plane: *(x1, y1), (x2, y2) , …, (xn, yn).*
- Find a line *y* = *ax* + *b* that minimizes the sum of the squared error:

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

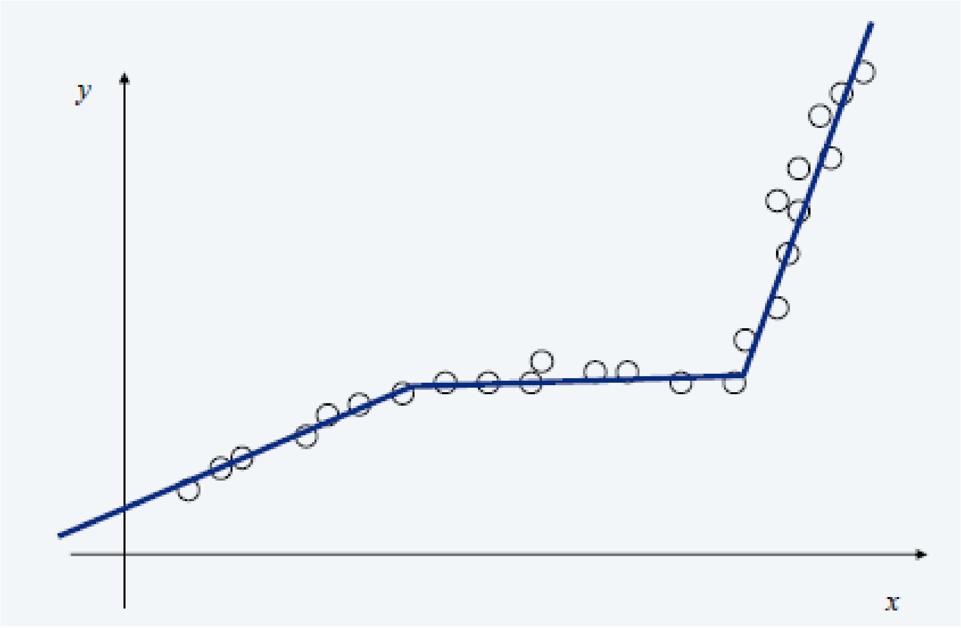


- Solution. Calculus ⇒ min error is achieved when

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented Least Squares

- Segmented least squares: Points lie roughly on a sequence of several line segments.
- Given *n* points in the plane: $(x_1, y_1), (x_2, y_2)$, …, (x_n, y_n) *with* $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes $f(x)$.



Segmented Least Squares

- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ *with* $x_1 < x_2 < \ldots < x_n$, *and a* constant $c > 0$, find a sequence of lines that minimizes $f(x) = E + c\,L$:
- *E = the sum of the sums of the squared errors in each segment.*
- *L = the number of lines.*

Segmented Least Squares

- Notation:
  - $OPT(j)$ = minimum cost for points $p_1, p_2, \ldots, p_j$.
  - $e(i, j)$ = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.
- To compute $OPT(j)$:
  - Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ *for some i.*
  - Cost = $e(i, j) + c + OPT(i - 1)$.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i \le j} \{\; e(i,j) + c + OPT(i-1) \} & \text{otherwise} \end{cases}$$

Segmented Least Squares

SEGMENTED-LEAST-SQUARES (n, p_1, …, p_n , c)

FOR j = 1 TO n

FOR i = 1 TO j

Compute the least squares $e(i, j)$ for the segment p_i, p_{i+1}, …, p_j.

$M[0] \leftarrow 0$.

FOR j = 1 TO n

$M[j] \leftarrow \min_{1 \le i \le j} \{ e_{ij} + c + M[i-1] \}$.

RETURN $M[n]$.

Segmented Least Squares

- The dynamic programming algorithm solves the segmented least squares problem in $O(n^3)$ *time and* $O(n^2)$ space.
- Bottleneck: computing $e(i, j)$ for $O(n^2)$ pairs.
- $O(n)$ per pair using formula.

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

- Can be improved to $O(n^2)$ time and $O(n)$ space by precomputing various statistics.

Thank you!