中山大學 SUN YAT-SEN UNIVERSITY

Lecture 2
Dynamic Programming

Algorithm

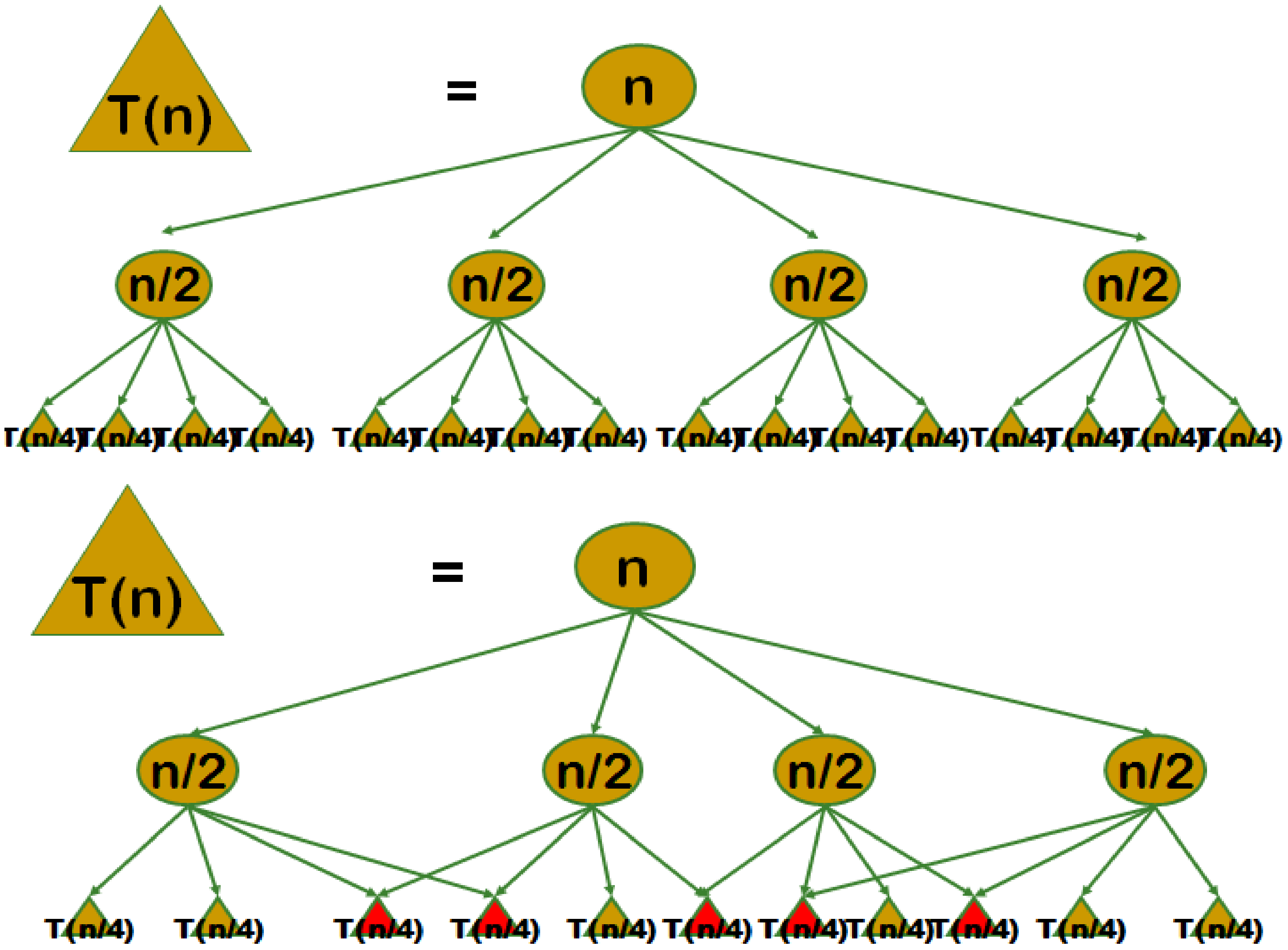
张子臻，中山大学计算机学院

zhangzizhen@gmail.com

Algorithmic Paradigms

- **Greedy**: Build up a solution incrementally, myopically optimizing some local criterion.
- **Divide-and-conquer**: Break up a problem into independent subproblems, solve each subproblem, and combine solution to subproblems to form solution to original problem.
- **Dynamic programming**: Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.

DC v.s. DP


T(n)
=
n
n/2
n/2
n/2
n/2
T(n/4)T(n/4)T(n/4)T(n/4) T(n/4)T(n/4)T(n/4)T(n/4) T(n/4)T(n/4)T(n/4)T(n/4)T(n/4)T(n/4)T(n/4)T(n/4)
T(n)
=
n
n/2
n/2
n/2
n/2
T(n/4) T(n/4) T(n/4) T(n/4) T(n/4) T(n/4) T(n/4)T(n/4)T(n/4) T(n/4) T(n/4)

Definition

- **Dynamic Programming (DP)** is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable to problems exhibiting the properties of **overlapping subproblems** and **optimal substructure**.
- A problem is said to have **overlapping subproblems** if the problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems.
- A problem is said to have **optimal substructure** if an optimal solution can be constructed efficiently from optimal solutions of its subproblems.

Dynamic Programming

- History
  - Richard Bellman pioneered the systematic study of dynamic programming in 1950s.
- Applications
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, compilers, systems…

Dynamic Programming Approaches

- **Top-down approach**: This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its subproblems, and if its subproblems are overlapping, then one can easily **memoize** or store the solutions to the subproblems in a table. Whenever we attempt to solve a new subproblem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it directly, otherwise we solve the subproblem and add its solution to the table.
- **Bottom-up approach**: Once we formulate the solution to a problem recursively as in terms of its subproblems, we can try reformulating the problem in a bottom-up fashion: try solving the subproblems first and use their solutions to build-on and arrive at solutions to bigger subproblems.

Development of DP algorithms

1. Characterize the structure of an optimal solution.
2. Define subproblems (states).
3. Write down the recurrence that relates subproblems.
4. Compute the value of an optimal solution.
5. Construct an optimal solution from computed information.

Fibonacci and Jump Steps Problem

- Naive Recursive Function:

```
int Fib(int n) {
        if (n==1 || n==2) return 1;
        return Fib(n-1)+Fib(n-2);
}
```

- Top-down Approach:

```
int Fib(int n) {
        if (n==1 || n==2) return 1;
        if (F[n] is defined) return F[n];
        F[n] = Fib(n-1)+Fib(n-2);
        return F[n];
}
```

- Bottom-up Approach

```
F[1] = F[2] = 1;
for (int i = 3; i < N; i++) F[i] = F[i-1]+F[i-2];
```

Fibonacci and Jump Steps Problem

- **Problem**: A frog can jump up 1, 3, or 4 steps in each move, calculate the number of different ways for the frog to achieve the *n-th* steps.
- Example:

  for $n = 5$, the answer is 6,

  5 = 1+1+1+1+1

  = 1+1+3

  = 1+3+1

  = 3+1+1

  = 1+4

  = 4+1

Fibonacci and Jump Steps Problem

- Let D_n be the number of ways to write n as the sum of 1, 3, 4.
- Find the recurrence

  $D_n = D_{n-1} + D_{n-3} + D_{n-4}$
- Solve the base cases

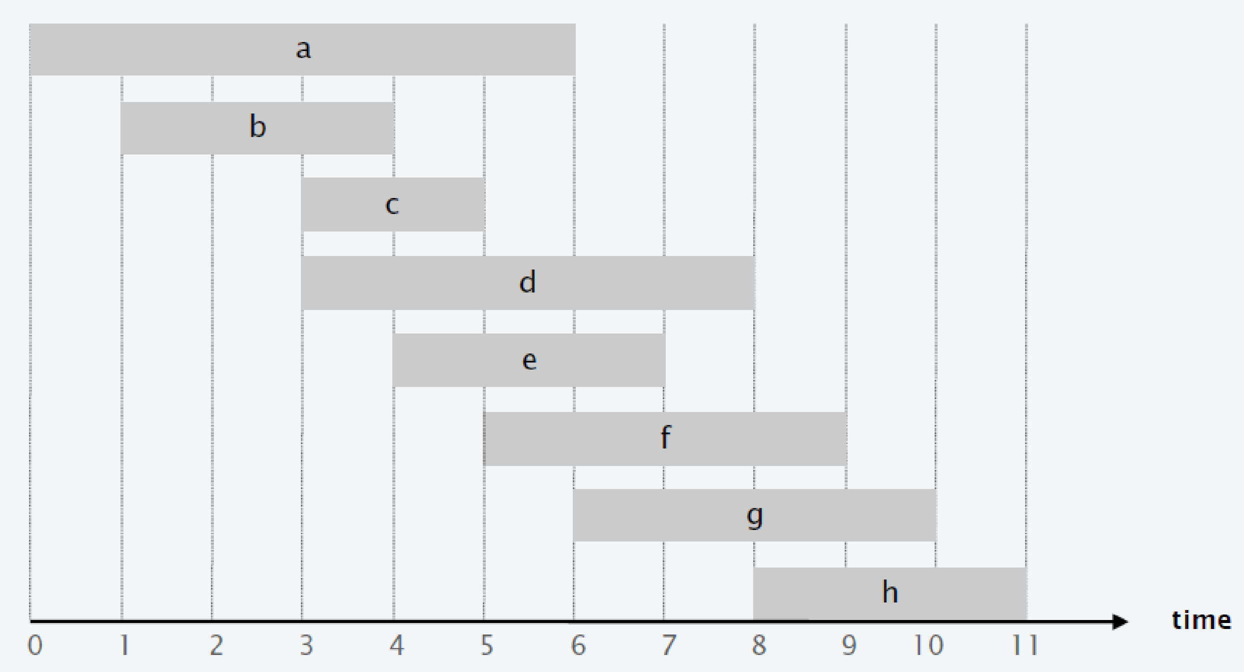
  $D_0 = 1$

  $D_n = 0$ for all negative n
- Implementation

```
D[0] = 1;
D[1] = D[2] = 1;
D[3] = 2;
for (int i = 4; i<= n; i++)
    D[i] = D[i-1] + D[i-3] + D[i-4];
```

Weighted interval scheduling (activity selection)

- Job *j* starts at s_j, finishes at f_j, and has weight or value v_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.



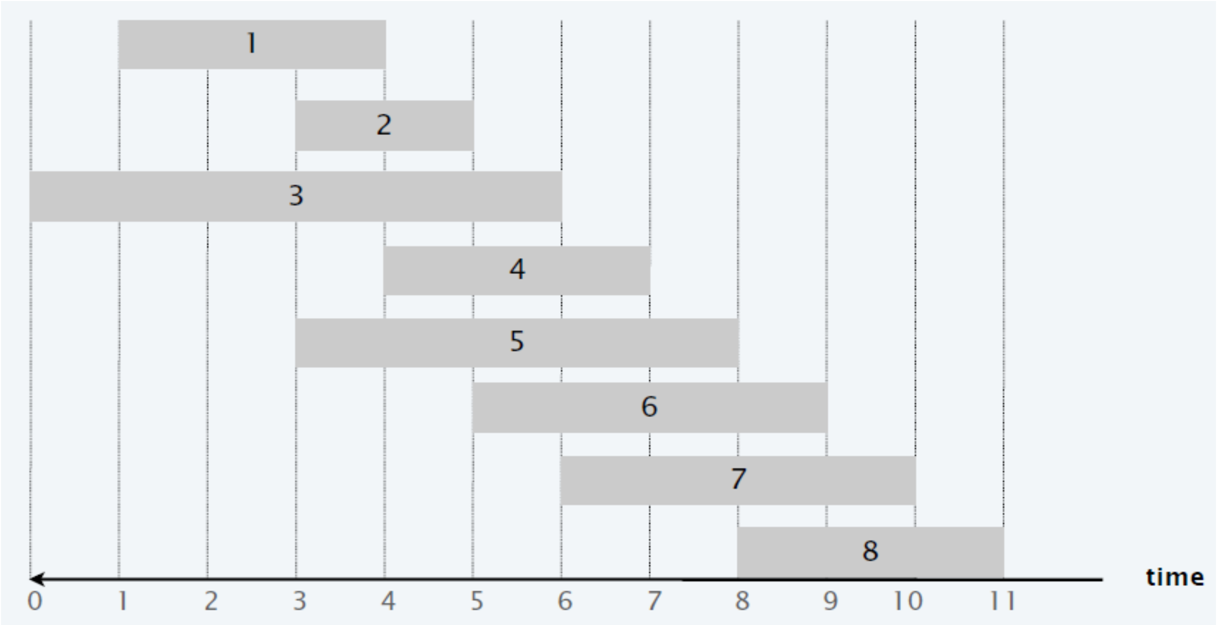
Weighted interval scheduling

- Consider jobs in ascending order of finish time: $f_1 \le f_2 \le . . . \le f_n$.
- Greedy algorithm is correct if all weights are 1.
- Greedy algorithm fails for weighted version.
- Let $p\ (j)$ = largest index $i < j$ such that job i is compatible with j.
- Example:

  p(8)=5,

  p(7)=3,

  p(2)=0.



Weighted interval scheduling

- Let *OPT(j)* = value of optimal solution to the problem consisting of job requests 1, 2, ..., *j*.
- Case 1. *OPT* selects job *j.*
  - Collect profit v_j.
  - Can't use incompatible jobs { *p(j) + 1, p(j) + 2, ..., j – 1 }.*
  - Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., *p(j).*
- Case 2. *OPT* does not select job *j.*
  - Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., *j – 1.*

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Weighted interval scheduling

- **Memoization**: Cache results of each subproblem; lookup as needed.

```
Input:  n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].
Compute p[1], p[2], …, p[n].

for j = 1 to  n
   M[j] ← empty.
M[0] ← 0.

M-Compute-Opt(j)
if M[j] is empty
   M[j] ← max(v[j] + M-Compute-Opt(p[j]), M-Compute-Opt(j – 1)).
return M[j].
```

Weighted interval scheduling

- Memoized version of algorithm takes *O(n log n)* time.
  - Sort by finish time: *O(n log n).*
  - Computing $p(\cdot)$: *O(n log n)* via binary search.
- Overall running time of M-COMPUTE-OPT(*n*) is *O(n).*
- Question. DP algorithm computes optimal value. How to find solution itself?
- Answer: Traceback.

Longest Common Subsequence

- The longest common subsequence (**LCS**) problem is to find the longest subsequence common to all sequences in a set of sequences.
- Given two strings *x* and *y*, find the LCS and print its length.
- Example:

  x: ABCBDAB

  y: BDCABC
- "BCAB" is the longest subsequence found in both sequences, so the answer is 4.

Longest Common Subsequence

- There are 2^m subsequences of X.
- Testing a subsequence (length k) takes time O(k).
- So brute force algorithm is O(n*2^m).
- Divide-and-conquer or Greedy algorithm?

Longest Common Subsequence

- Let two sequences be defined as follows: $X = (x_1, x_2...x_m)$ and $Y = (y_1, y_2...y_n)$. The prefixes of X are $X_{1,\ 2,...m}$; the prefixes of Y are $Y_{1,\ 2,...n}$.
- Let $LCS(X_i,Y_j)$ represent the set of longest common subsequence of prefixes X_i and Y_j.
- Find the recurrence
  - If $x_i = y_j$, they both contribute to the LCS. Then,
    $LCS(X_i,Y_j) = LCS(X_{i-1},Y_{j-1}) + 1$
  - Either x_i or y_j does not contribute to the LCS, so one can be dropped.
    $LCS(X_i,Y_j) = \max\{ LCS(X_{i-1},Y_j), LCS(X_i,Y_{j-1}) \}$
- Find and solve the base cases:

$$LCS(X_i,Y_0) = LCS(X_0,Y_j) = 0$$

Longest Common Subsequence

- Implementation: O(nm)

```
for (i=0;i<=n;i++) LCS[i][0]=0;
for (j=0;j<=m;j++) LCS[0][j]=0;
for (i=1;i<=n;i++) {
        for (j=1;j<=m;j++) {
                if (x[i]==y[j]) LCS[i][j]=LCS[i-1][j-1]+1;
                else LCS[i][j]=max(LCS[i-1][j],LCS[i][j-1]);
        }
}
```

Longest Common Subsequence

- Example:

  X=(AGCAT)

  Y=(GAC)

- LCS matrix:

	Ø	A	G	C	A	T
Ø	0	0	0	0	0	0
G	0	←↑0	↖1	←1	←1	←1
A	0	↖1	←↑1	←↑1	↖2	←2
C	0	↑1	←↑1	↖2	←↑2	←↑2

Traceback example:

	Ø	A	G	C	A	T
Ø	0	0	0	0	0	0
G	0	←↑0	↖1	←1	←1	←1
A	0	↖1	←↑1	←↑1	↖2	←2
C	0	↑1	←↑1	↖2	←↑2	←↑2

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	∅	∅	-
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)

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

  $A_1A_2A_3A_4$ 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

$$m[i,j] = \begin{cases} 0 & \textbf{if } i = j \ , \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \textbf{if } i < j \end{cases}$$

- 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.

Traveling Salesman Problem

- Given n cities and the distances d_{ij} between any two of them, we wish to find the shortest tour going through all cities and back to the starting city. Generally, the TSP is given as a graph $G=(V,D)$ where $V=\{1,2, . . ., n\}$ is the set of cities, and D is the adjacency distance matrix, with $\forall i,j \in V$, $i \neq j$, $d_{ij}>0$, the problem is to find the tour with minimal distance weight, that starting at city 1 goes through all n cities and returns to city 1.

Traveling Salesman Problem

- The TSP is a well known NP-hard problem.
- There are n! feasible solutions. Enumerate them may take O(n!) time.
- What is the appropriate subproblem for the TSP?
  - Suppose we have started at city 1 as required, have visited a few cities, and are now in city j. What information do we need in order to extend this partial tour?
  - We need to know j, since this will determine which cities are most convenient to visit next.
  - We also need to know all the cities visited so far, so that we don't repeat any of them.

Traveling Salesman Problem

- For a subset of cities $S \subseteq \{1,2,\ldots,n\}$ that includes 1, and $j \in S$, let $C(S,j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.
- How to express $C(S,j)$ in terms of smaller sub-problems.
- We need to start at 1 and end at j; what should we pick as the second-to-last city? It has to be some $i \in S$, so the overall path length is the distance from 1 to i, namely, $C(S-\{j\}, i)$, plus the length of the final edge, d_{ij}.
- We pick the best i, then

$$C(S,j) = \min_{i \in S : i \neq j} C(S - \{j\}, i) + d_{ij}$$

Traveling Salesman Problem

- There are at most $2^n * n$ subproblems.
- Each one takes linear time to solve.
- The total time complexity is $O(2^n * n^2)$.
- The sub-problems are ordered by |S|. (Use a queue to extend *S*))

```
C({1},1) = 0
for s = 2 to n:
    for all subsets S ⊆ {1,2,...,n} of size s and containing 1:
        C(S,1) = ∞
        for all j ∈ S, j ≠ 1:
            C(S, j) = min{C(S − {j}, i) + d_ij : i ∈ S, i ≠ j}
return min_j C({1, ..., n}, j) + d_j1
```

Traveling Salesman Problem

- How to represent the set *S*?
- Usually, we can Represent a set of *n* elements as an *n* bit numbers.
- Example: (n=5)

S=Φ → $(00000)_2$ → 0

S={0} → $(00001)_2$ → 1

S={1,3} → $(01010)_2$ → 10

S={0,1,2,3,4}→ $(11111)_2$ → 15

Traveling Salesman Problem

- Check if element *i* is present in set *S*
- Find the resulting set when we add *i* to set *S*
- Iterating through all the subsets of size <= n

Homework

- http://soj.acmm.club/contest_detail.php?cid=2959

ID	Title
1000	Coins
1001	Traveling Salesman Problem

Thank you!