中山大學
SUN YAT-SEN UNIVERSITY

Lecture 6
Dynamic Programming
Part I

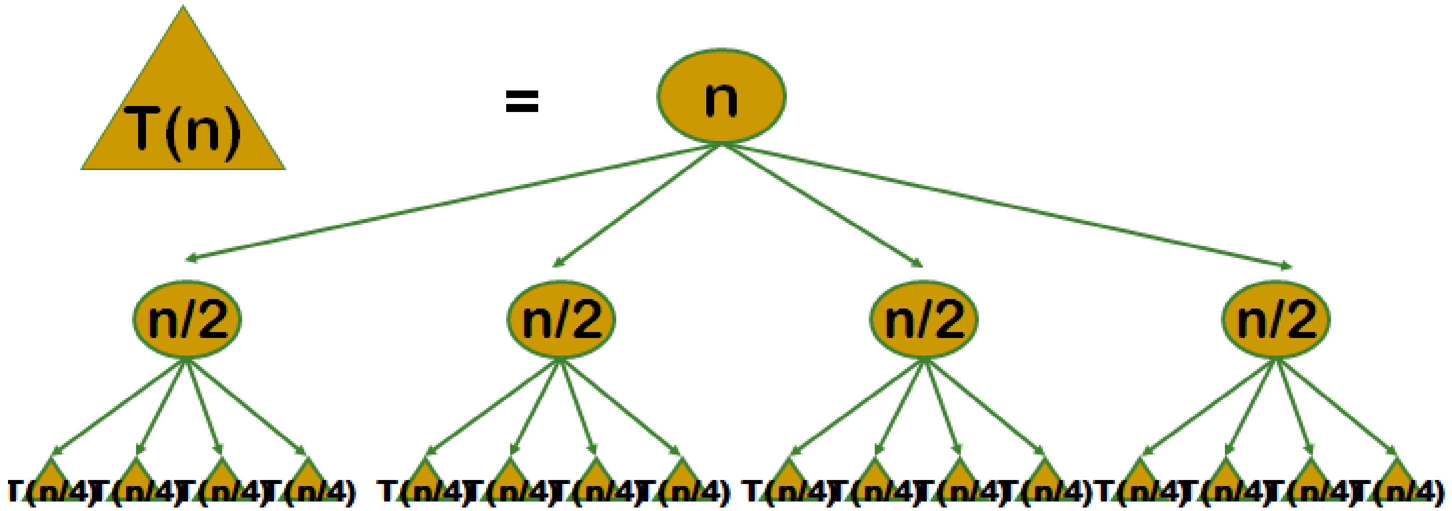
Algorithm Design and Analysis

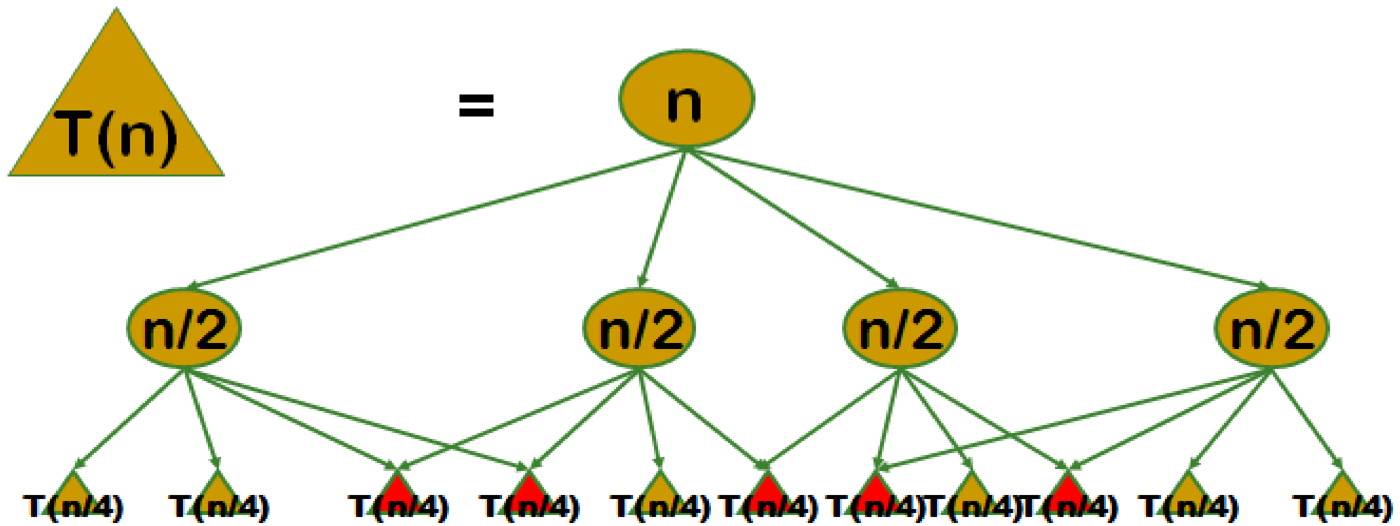
zhangzizhen@gmail.com

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, reinforcement learning…

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 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];
```

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

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

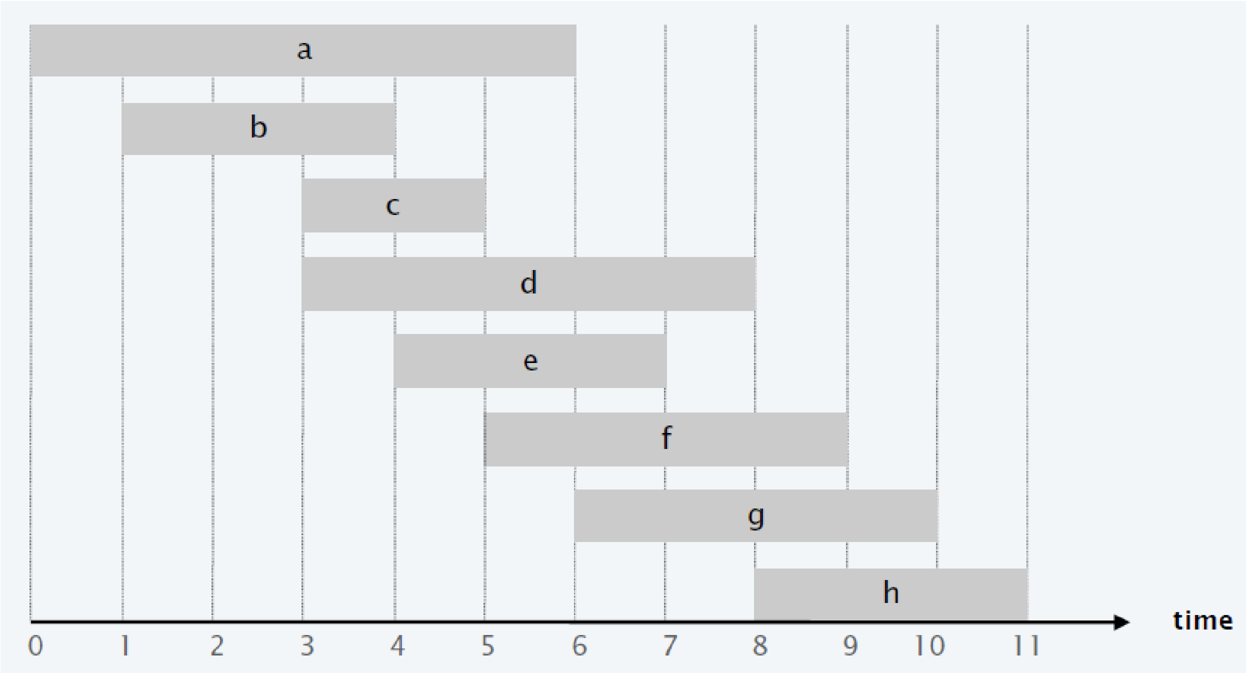
  $D_0 = 1$

  $D_n = 0$ 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.



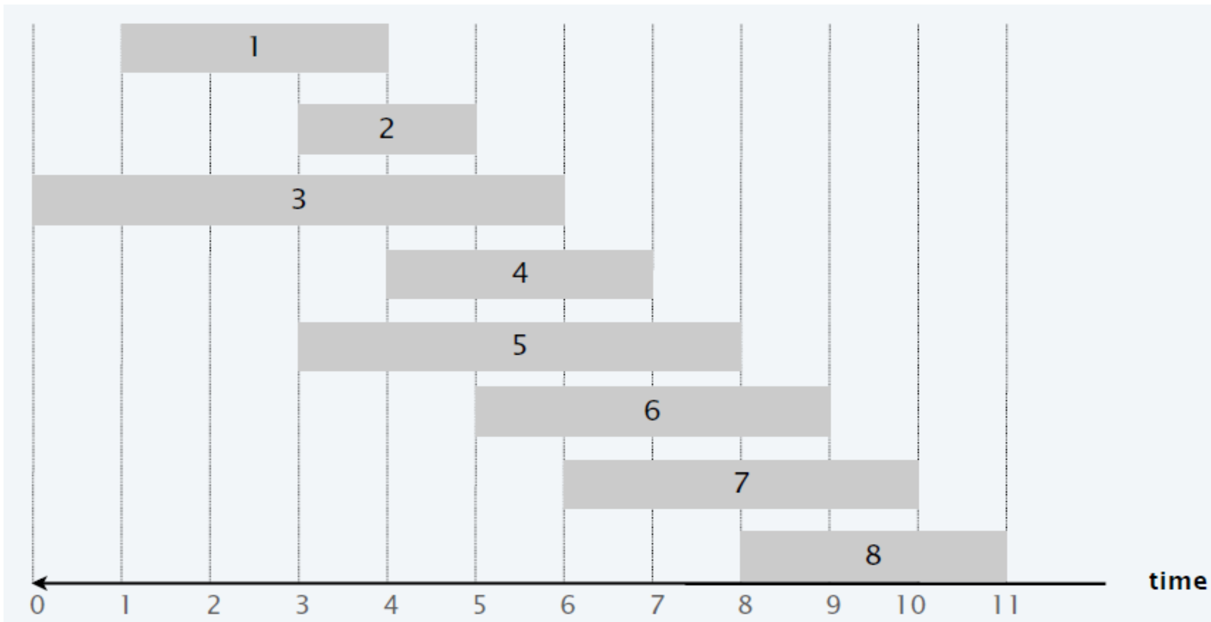
Weighted interval scheduling

- 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(·) : 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: $X = (x_1, x_2 \ldots x_m)$ and $Y = (y_1, y_2 \ldots y_n)$. The prefixes of X are $X_{1,\ 2,\ldots m}$; the prefixes of Y are $Y_{1,\ 2,\ldots n}$.
- 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]: $i<j$ 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;
        }
}
```

Exercise

- soj.acmm.club

  1176 1011 1121 1264 1828

  1527 1148 1163 1345 13062

- Choose at least 2 problems and write a report.
- Create a zip/rar file: ID_name_version.zip
- Submit it to 239o58336k.qicp.vip:55469
  - Account: login
  - Password: 123456

- Deadline: March 31.

Thank you!