Lecture 7-Dynamic Programming

AIAA 5037 Advanced Algorithms and Data Structures

Ying Sun, AI Thrust

Outline

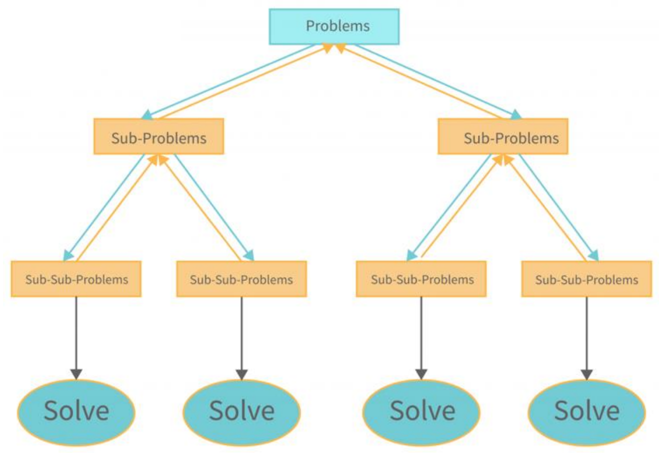
- Introduction to Dynamic Programming
- Rod Cut
- Matrix Chain Multiplication
- Longest Common Subsequence
- Longest Increasing Subsequence
- DP on Tree

From Divide-and-Conquer to Dynamic Programming

Recall Divide-and-Conquer

A strategy to recursively solve the problem

- **Divide** the problem into subproblems
- **Conquer** the subproblems
- **Combine** the subproblems' solutions into original problem's solution



Recall Divide-and-Conquer

Recall Fibonacci sequence: given n, output the n-th Fibonacci number

$$F_n = \begin{cases} 0 & if\ n = 0; \\ 1 & if\ n = 1; \\ F_{n-1} + F_{n-2} & if\ n \geq 2. \end{cases}$$

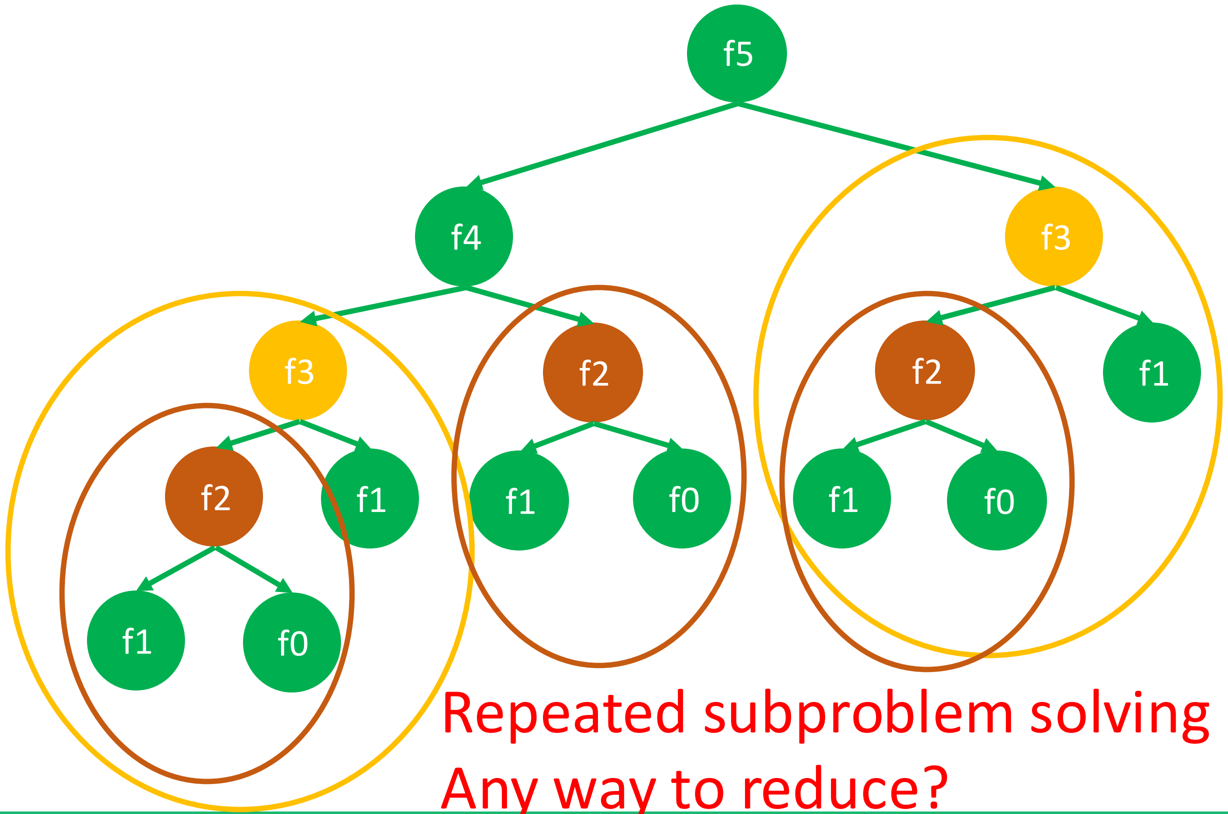
1 1 2 3 5 8 13 21 34 ...

- Divide-and-Conquer solution

```
fib(n):
    if n<=1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

- Complexity: $O(\phi^n)$
- Why is it so high?

Observe the recursion tree for the subproblems



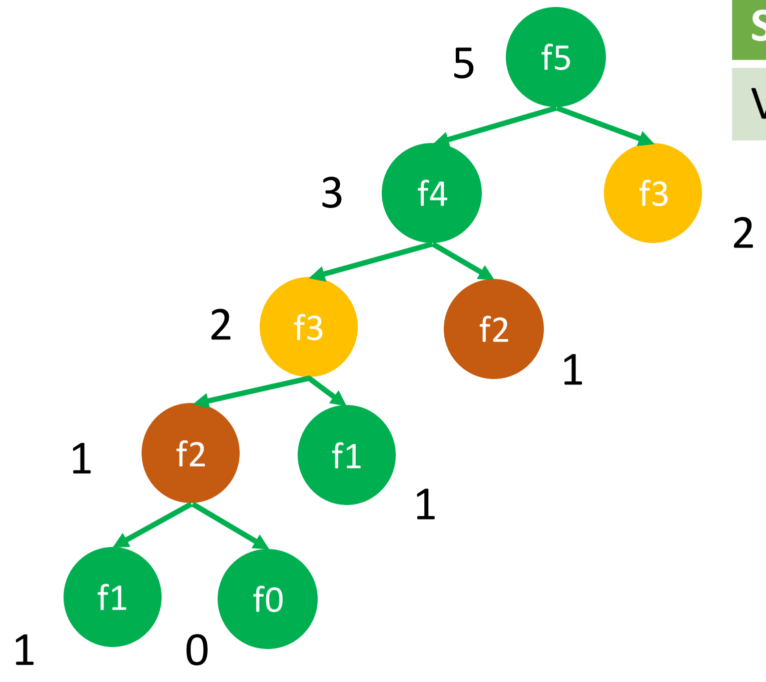
Repeated subproblem solving

Any way to reduce?

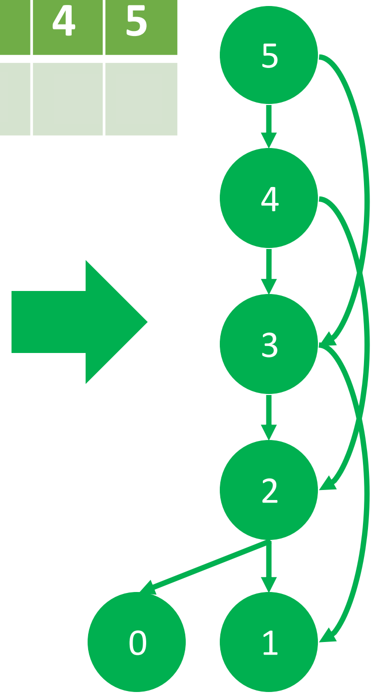
Dynamic Programming

Another paradigm that solves problems by combining the solutions to subproblems

- Handle overlapping subproblems with a table (difference from divide-and-conquer)
- (Programming here refers to the tabular method)



State	0	1	2	3	4	5
Value						

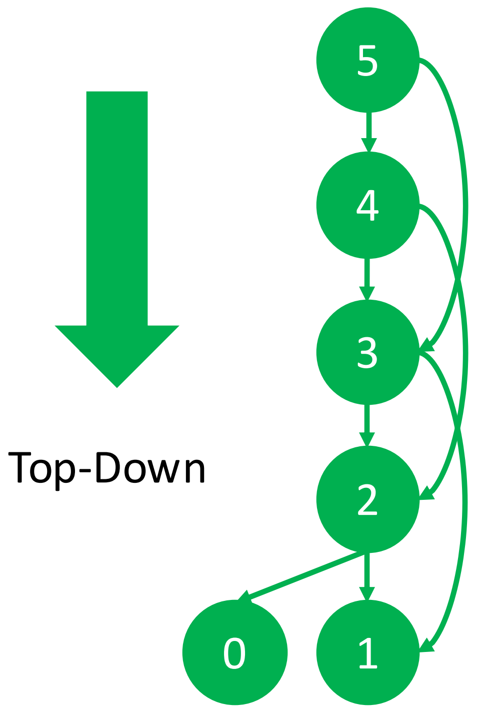


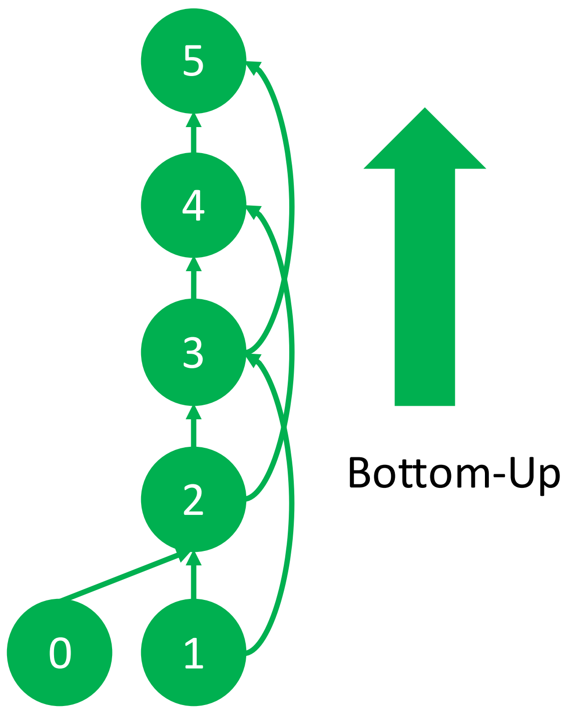
Subproblem graph

- Each node is a subproblem
- The dependency between subproblems
- A reduced version of recursion tree

Dynamic Programming

Two implementations of dynamic programming

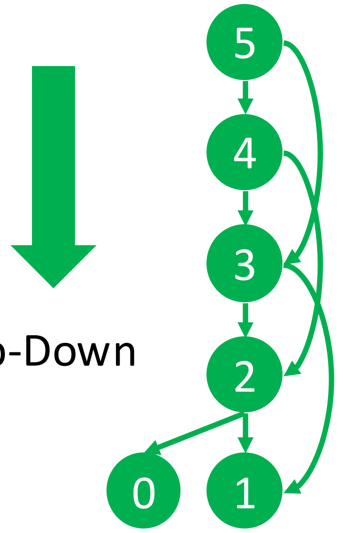




Top-Down with Memorization

Recursive implementation

- Start from original problem, recursively solves small subproblems
- Once called, subproblem solution stored in a table or array (memorization table)
- Checks if the solution to that subproblem is in the memorization
    - Exist: retrieve and use; Not exist: (recursively) solve the subproblem and store it

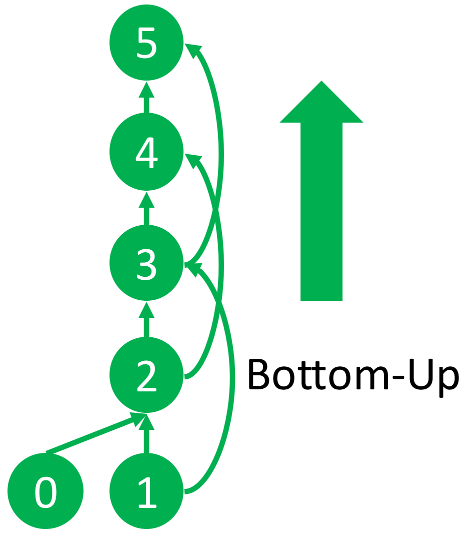


```
Fib = [-1] # Memorization table
def Cal_Fib(n):
    if n <= 1:
        return n
    if Fib[n] != -1:
        return Fib[n]
    Fib[n] = Cal_Fib(n-1) + Cal_Fib(n-2)
    return Fib[n]
```

Bottom-Up

Apply on a subproblem graph where subproblem depends only on “smaller” subproblems

- Iteratively solve subproblems in the order of smaller problems to larger problems
- Larger subproblems solved based on solutions of previously solved small subproblems



```
def Cal_Fib(n):
  fib_table = [0] * (n+1)
  fib_table[0] = 0
  fib_table[1] = 1

  for i in range(2, n+1):
    fib_table[i] = fib_table[i-1] + fib_table[i-2]

  return fib_table[n]
```

Notice: There will always be subproblems that are bottom in the topological order, but not always easy to define the order based on the physical meaning of the problem.

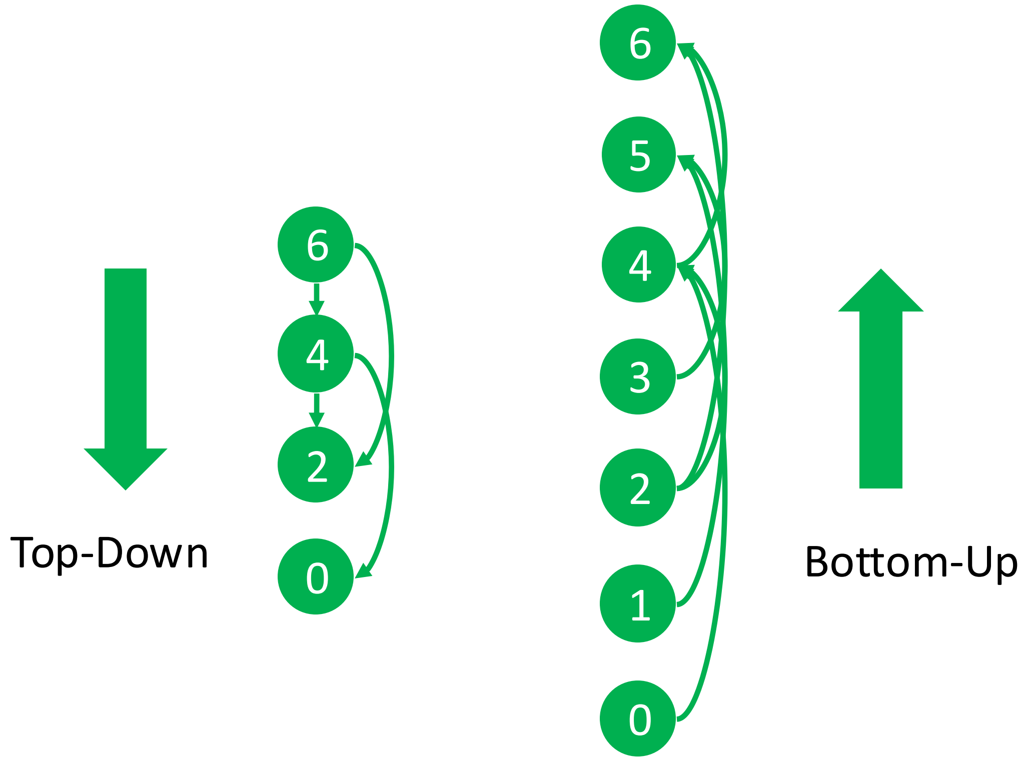
Top-Down vs. Bottom-Up

Bottom-Up: easy implementation, no cost for recursive function calling, smaller coefficient for time complexity

Top-Down: Sometimes no need to solve all the subproblems

Example

- Base: $f(0) = f(1) = f(2) = f(3) = 1$
- Recursive: $f(n) = f(n-2) + f(n-4)$



Complexity Analysis

Analyze time complexity with subproblem graph

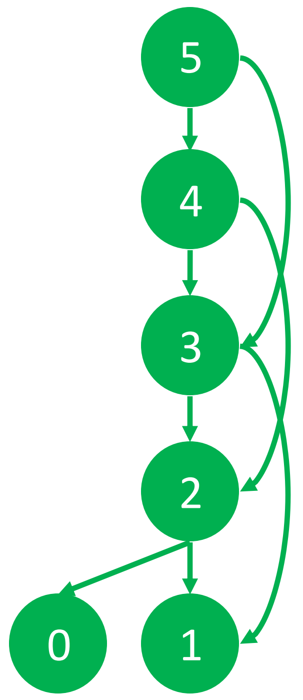
Time complexity

- n subproblem (nodes), each is solved at most once
- m dependences (edges), each is used once to solve the subproblem
- In total: $O(n+m)$

Space complexity

- n subproblem (nodes), each recorded in the table: $O(n)$

When influenced by multiple terms containing multiple parameters without a definite order of dominance, we use the sum to depict complexity. E.g., $O(n+m)$, $O(n^2+m\log n)$, etc

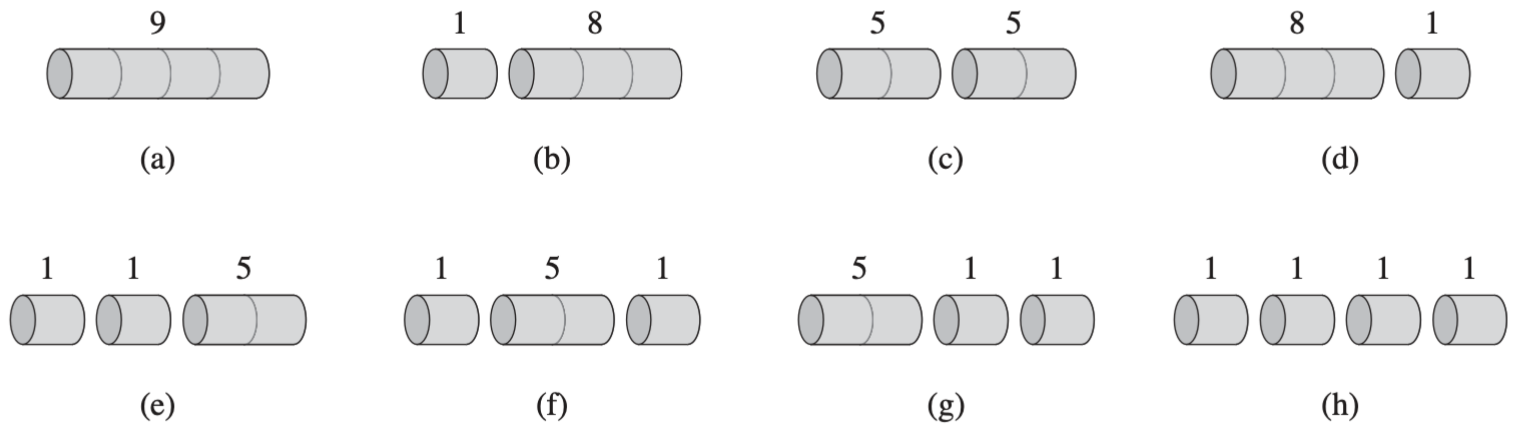


Optimization with DP: Rod Cutting

Rod Cutting

Problem: given a rod of length n, the price for selling a rod with length i is p_i. You can cut up the rod and sell the pieces, determine the maximum revenue r_n

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



Example: 8 ways to cut a rod of length 4, (c) is the best

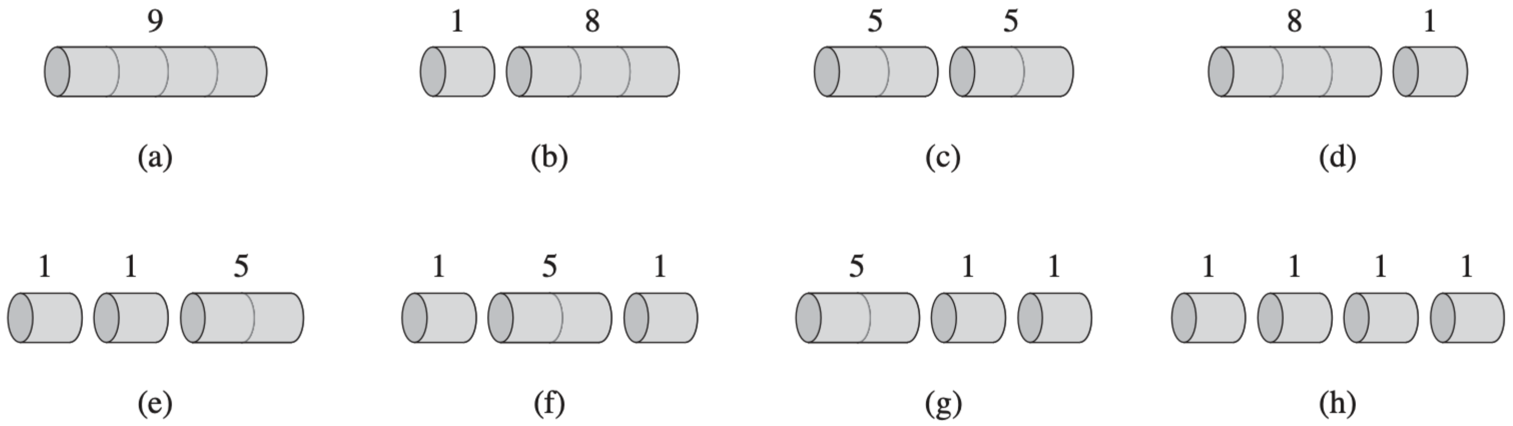
Rod Cutting

Problem: given a rod of length n, the price for selling a rod with length i is p_i. You can cut up the rod and sell the pieces, determine the maximum revenue r_n

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Naïve solution: Brute Force

- 2^{n-1} plans, at every inch, we choose cut/not cut, so exponential complexity



Rod Cutting

Problem: given a rod of length n, the price for selling a rod with length i is p_i. You can cut up the rod and sell the pieces, determine the maximum revenue r_n

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Divide-and-Conquer

- If the first cut divides the rod into length i and $n - i$, the maximum revenue is $r_i + r_{n-i}$
  - Try the first cuts and pick the best one: $f(n) = \max(p_n,\ \max_{i=1}^{n-1}(f(i) + f(n-i)))$

```
f(p,n)
    q = p[n]
    for i = 1 to n-1:
        q = max(q,f(p,i)+ f(p,n-i))
    return q
```

- Complexity: $\mathrm{T}(n) = \Theta(1) + \sum_{i=1}^{n-1}\big(\mathrm{T}(i) + \mathrm{T}(n-i)\big) = \Theta(1) + 2\sum_{i=1}^{n-1}\mathrm{T}(i)$

Rod Cutting

Problem: given a rod of length n, the price for selling a rod with length i is p_i. You can cut up the rod and sell the pieces, determine the maximum revenue r_n

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Better solution?

- When the first piece has length i, then maximum revenue is $p_i + r_{n-i}$
  - Try the first piece and pick the best one: $f(n) = \max(p_n,\ \max_{i=1}^{n-1}(p_i + f(n-i)))$

```
f(p,n)
    q = p[n]
    for i = 1 to n-1:
        q = max(q, p[i] + f(p,n-i))
    return q
```

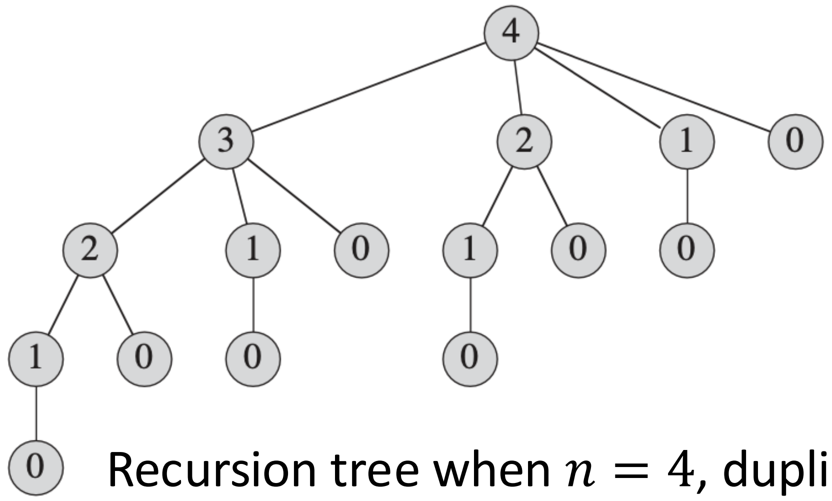
- Complexity: $\mathrm{T}(n) = \Theta(1) + \sum_{i=1}^{n-1} \mathrm{T}(i) = \Theta(2^n)$
- Why is it so slow?

Rod Cutting

Problem: given a rod of length n, the price for selling a rod with length i is p_i. You can cut up the rod and sell the pieces, determine the maximum revenue r_n

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Observe the subproblem recursion tree for $f(n) = \max(p_n,\ \max_{i=1}^{n-1}(p_i + f(n-i)))$



Recursion tree when $n = 4$, duplicated calculation

Practice

- Draw the subproblem graph
- Solve with dynamic programming

Rod Cutting

Problem: given a rod of length n, the price for selling a rod with length i is p_i. You can cut up the rod and sell the pieces, determine the maximum revenue r_n

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Dynamic programming for $f(n) = \max(p_n,\ \max_{i=1}^{n-1}(p_i + f(n-i)))$

- Top-Down

```
MEMOIZED-CUT-ROD$(p, n)$
let $r[0..n]$ be a new array
for $i = 0$ to $n$
    $r[i] = -\infty$
return MEMOIZED-CUT-ROD-AUX$(p, n, r)$
```

```
MEMOIZED-CUT-ROD-AUX$(p, n, r)$
if $r[n] \geq 0$
    return $r[n]$
if $n == 0$
    $q = 0$
else $q = -\infty$
    for $i = 1$ to $n$
        $q = \max(q, p[i] + $ MEMOIZED-CUT-ROD-AUX$(p, n - i, r))$
$r[n] = q$
return $q$
```

Rod Cutting

Problem: given a rod of length n, the price for selling a rod with length i is p_i. You can cut up the rod and sell the pieces, determine the maximum revenue r_n

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Dynamic programming for $f(n) = \max(p_n,\ \max_{i=1}^{n-1}(p_i + f(n-i)))$

- Bottom-Up
  - Solution for larger length based on on smaller ones
  - Problem size: length of rod

Time Complexity:

$$T(n) = \sum_{j=1}^{n} \sum_{i=1}^{j} \Theta(1) = \Theta(n^2)$$

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = −∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j − i])
7      r[j] = q
8  return r[n]
```

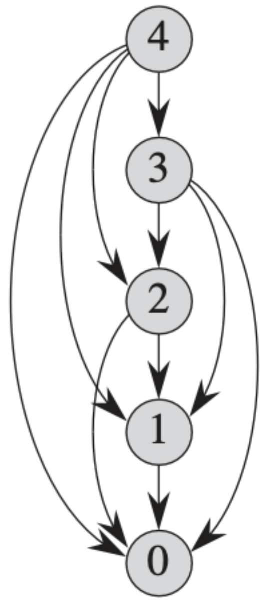
Rod Cutting

Complexity analysis with subproblem graph for $f(n) = \max(p_n,\ \max_{i=1}^{n-1}(p_i + f(n-i)))$

- Number of subproblems: n
- Number of edges for problem i: i
- Total number of edges: $\sum_{i=1}^{n} i = \Theta(n^2)$

Time complexity: $\Theta(n + n^2) = \Theta(n^2)$

Space complexity: $\Theta(n)$



Key Characteristics

Use dynamic programming when you construct a problem with characteristics:

- **Optimal Substructure:** An optimal solution to the problem can be constructed from optimal solutions to its subproblems
    - If the first cut divides the rod into i and $n-i$, the maximum revenue is $r_i + r_{n-i}$
        - $f(n) = \max(p_n,\ \max_{i=1}^{n-1}(f(i) + f(n-i)))$
    - When the first piece has length i, then maximum revenue is $p_i + r_{n-i}$
        - $f(n) = \max(p_n,\ \max_{i=1}^{n-1}(p_i + f(n-i)))$
- Overlapping Subproblems

Reconstruct the Optimal Rod Cut

- Store the transition in the subproblem graph (i.e., length of the first piece for the optimal solution of each subproblem)
- (Recursively) print the first piece for each subproblem and go to the subsequent subproblem

```
EXTENDED-BOTTOM-UP-CUT-ROD$(p, n)$
    let $r[0..n]$ and $s[0..n]$ be new arrays
    $r[0] = 0$
    for $j = 1$ to $n$
        $q = -\infty$
        for $i = 1$ to $j$
            if $q < p[i] + r[j-i]$
                $q = p[i] + r[j-i]$
                $s[j] = i$
        $r[j] = q$
    return $r$ and $s$
```

```
PRINT-CUT-ROD-SOLUTION$(p, n)$
    $(r, s) =$ EXTENDED-BOTTOM-UP-CUT-ROD$(p, n)$
    while $n > 0$
        print $s[n]$
        $n = n - s[n]$
```

Summary of Dynamic Programming

Solving problems with overlapping subproblems

- Usually applied to optimization problems

Steps of Dynamic Programming:

1. Characterize the structure of an optimal solution (define problem that has optimal substructure): most critical, usually not trivial
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution (top-down with memorization or bottom-up)
4. (optional) Construct an optimal solution from computed information

Important Intuition: The subproblems can be stored in a table for efficient reference

- Cannot have too many subproblems
- Subproblem is representable by a distinct ‘state identifier’

Matrix-Chain Multiplication

Matrix Chain Multiplication

Problem: Given a sequence (chain) $< A_1; A_2; \dots; A_n >$ of n matrices ($A_i \in R^{p_{i-1} \times p_i}$), compute the product $A_1 A_2 \dots A_n$, decide the computation order to minimize computation cost.

The order of multiplication has a dramatic impact on the cost of evaluating the product.

Example: $A_1 \in R^{10\times100}, A_2 \in R^{100\times5}, A_3 \in R^{5\times50}$

- $((A_1A_2)A_3)$: $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$ *multiplications*
- $(A_1(A_2A_3))$: $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75000$ *multiplications*

We refer to this form as a **fully parenthesized matrix multiplication:**

- A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses

Problem transformed: find the optimal FPMM

Matrix Chain Multiplication

Problem: Given a sequence (chain) $< A_1; A_2; \dots; A_n >$ of n matrices ($A_i \in R^{p_{i-1} \times p_i}$), compute the product $A_1 A_2 \dots A_n$, decide the computation order to minimize computation cost

- Complexity for brute force?
  - What is the total number of FPMM?
    - Hint: FPMM is product of two FPMM surrounded by a parenthesis
      - $((A_1A_2)(A_3A_4))$, $((A_1A_2A_3)A_4)$, $(A_1(A_2A_3A_4))$
    - Product of FPMM of two sub-chains

$$P(n) = \begin{cases} 1 & \text{if } n = 1\,, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2\,. \end{cases}$$

$$\begin{array}{l} (A_1 | (A_2(A_3A_4))) \\ (A_1 | ((A_2A_3)A_4)) \\ ((A_1A_2) | (A_3A_4)) \\ ((A_1(A_2A_3)) | A_4) \\ (((A_1A_2)A_3) | A_4) \end{array}$$

- $P(n)$ is Catalan number, which grows as $\Omega(4^n/n^{3/2})$

Matrix Chain Multiplication

Problem: Given a sequence (chain) $< A_1; A_2; \dots; A_n >$ of n matrices compute the product $A_1 A_2 \dots A_n$, with $A_i \in R^{p_{i-1} \times p_i}$ decide the computation order to minimize computation cost

$$\begin{array}{l} (A_1 | (A_2(A_3A_4))) \\ (A_1 | ((A_2A_3)A_4)) \\ ((A_1A_2) | (A_3A_4)) \\ ((A_1(A_2A_3)) | A_4) \\ (((A_1A_2)A_3) | A_4) \end{array}$$

Dynamic programming?

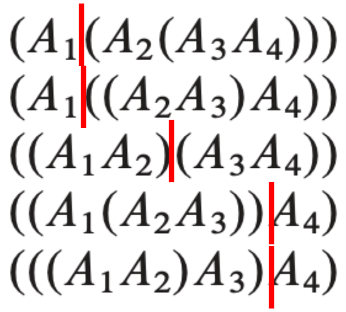
Step 1: Characterize optimal substructure

- Given optimal FPMM $P_{1:n}$ of $A_{1:n}$, if $P_{1:n}$ separates the two chains at A_k, it contains FPMM $P_{1:k}$ and $P_{k+1:n}$ for $A_{1:k}$ and $A_{k+1:n}$, which are optimal FPMM for $A_{1:k}$ and $A_{k+1:n}$
- Proof by contradiction – obvious
  - If there is $P'_{1:k}$ so that $cost(P'_{1:k}) < cost(P_{1:k}) \Rightarrow \text{cost}\big((P'_{1:k}, P_{k+1:n})\big) < cost(P_{1:n})$
  - $P_{1:n}$ is not optimal

Matrix Chain Multiplication

Problem: Given a sequence (chain) $< A_1; A_2; \dots; A_n >$ of n matrices compute the product $A_1 A_2 \dots A_n$, with $A_i \in R^{p_{i-1} \times p_i}$ decide the computation order to minimize computation cost

Dynamic programming?



$$(A_1 | (A_2(A_3A_4)))$$
$$(A_1 | ((A_2A_3)A_4))$$
$$((A_1A_2) | (A_3A_4))$$
$$((A_1(A_2A_3)) | A_4)$$
$$(((A_1A_2)A_3) | A_4)$$

Step 2: Recursively define the value of an optimal solution

- Use $f(A_1, \dots, A_n)$ to denote the minimal computation cost
- Supposing the cutting point is between A_i and A_{i+1}, the minimum cost is

$$f(A_1, \dots, A_i) + f(A_{i+1}, \dots, A_n) + p_0 \cdot p_i \cdot p_n$$

- $f(A_1, \dots, A_n) = \min_i (f(A_1, \dots, A_i) + f(A_{i+1}, \dots, A_n) + p_0 \cdot p_i \cdot p_n)$

Matrix Chain Multiplication

Problem: Given a sequence (chain) $< A_1; A_2; \dots; A_n >$ of n matrices compute the product $A_1 A_2 \dots A_n$, with $A_i \in R^{p_{i-1} \times p_i}$ decide the computation order to minimize computation cost

$$f(A_1, \dots, A_n) = \min_i (f(A_1, \dots, A_i) + f(A_{i+1}, \dots, A_n) + p_0 \cdot p_i \cdot p_n)$$

Step 3: Compute the value of an optimal solution

- Top-Down with memorization

```
Init m[i,j]=inf   # Memorization table
def f(l, r):
    if m[l,r] != inf:
        return m[l,r]
    if l == r:
        m[l,r] = 0
        return 0
    for k = l to r-1:
        q = f(l, k) + f(k+1, r)+p[l-1]*p[k]*p[r]
        if q < m[l,r]:
            s[l,r] = k  # the best cut
            m[l,r] = q
    return m[l,r]
```

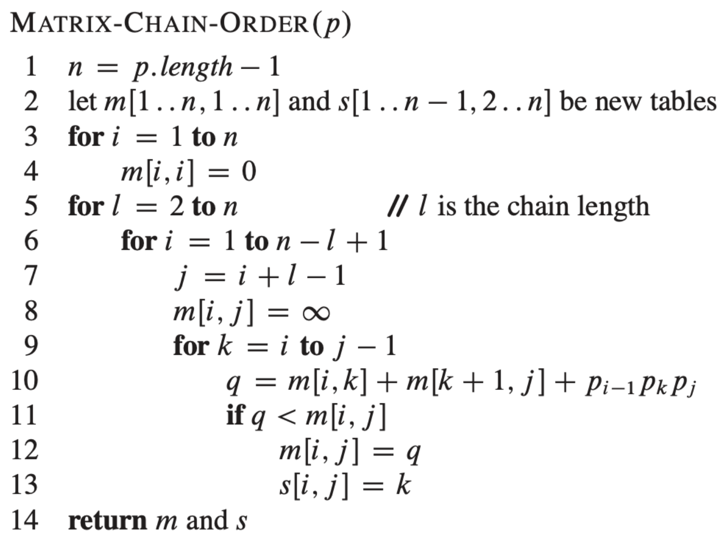
Matrix Chain Multiplication

Problem: Given a sequence (chain) $< A_1; A_2; \dots; A_n >$ of n matrices compute the product $A_1 A_2 \dots A_n$, with $A_i \in R^{p_{i-1} \times p_i}$ decide the computation order to minimize computation cost

$$f(A_1, \dots, A_n) = \min_i (f(A_1, \dots, A_i) + f(A_{i+1}, \dots, A_n) + p_0 \cdot p_i \cdot p_n)$$

Step 3: Compute the value of an optimal solution

- Bottom-Up
  - Define problem size: the length of the sequence
  - Get optimal solution from smaller (shorter) problems
- Complexity:
- $\sum_{l=2}^{n}(l-1)(n-l+1) = \sum_{l=2}^{n}(l-1)n - \sum_{l=2}^{n}(l-1)^2$
- $= \sum_{l=1}^{n-1} ln - \sum_{l=1}^{n-1} l^2 = \theta(n^3)$



```
MATRIX-CHAIN-ORDER($p$)
$n = p.length - 1$
let $m[1..n, 1..n]$ and $s[1..n-1, 2..n]$ be new tables
for $i = 1$ to $n$
    $m[i,i] = 0$
for $l = 2$ to $n$                // $l$ is the chain length
    for $i = 1$ to $n - l + 1$
        $j = i + l - 1$
        $m[i,j] = \infty$
        for $k = i$ to $j - 1$
            $q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$
            if $q < m[i,j]$
                $m[i,j] = q$
                $s[i,j] = k$
return $m$ and $s$
```

Matrix Chain Multiplication

Problem: Given a sequence (chain) $< A_1; A_2; \dots; A_n >$ of n matrices compute the product $A_1A_2 \dots A_n$, with $A_i \in R^{p_{i-1} \times p_i}$ decide the computation order to minimize computation cost

Step 4: Reconstruct the optimal solution (Recursively)

```
PRINT-OPTIMAL-PARENS($s, i, j$)
if $i == j$
    print "$A$"$_i$
else print "("
    PRINT-OPTIMAL-PARENS($s, i, s[i, j]$)
    PRINT-OPTIMAL-PARENS($s, s[i, j] + 1, j$)
    print ")"
```

Matrix Chain Multiplication

Problem: Given a sequence (chain) $< A_1; A_2; \ldots; A_n >$ of n matrices compute the product $A_1 A_2 \ldots A_n$, with $A_i \in R^{p_{i-1} \times p_i}$ decide the computation order to minimize computation cost

Complexity Analysis?

- Number of subproblems (sub-chains): $\sum_{i=1}^{n}(n-i+1) = \Theta(n^2)$
- Number of dependences (edges) for subproblem $< A_i; \ \ldots; A_j >$: $2(j-i)$
- Total edges
  - $\sum_{i=1}^{n}\sum_{j=i}^{n} 2(j-i) = \sum_{i=1}^{n}\sum_{l=1}^{n-i+1} 2(l \ -1) = 2\sum_{i=1}^{n}\sum_{l=2}^{n-i} l = \Theta(n^3)$
- Time complexity: $\Theta(n^3)$
- Space complexity: $\Theta(n^2)$

Longest Common Subsequence

Problem

Given two sequences A = <a_1, a_2, a_3 , … , a_n> and B = <b_1, b_2, b_3 , … , b_m>, find their Longest Common Subsequence (not necessarily consecutive)

- *Subsequence: a sequence derived from another sequence by deleting some elements without changing the order of the remaining elements*
- **Example**: A="acbdef", B="abcdef", then LCS(A,B)="acdef" or "abdef" with the length 5

Brute Force 1: enumerate all subsequences of A and B, check whether they are equal

- Enumeration: $2^n/2^m$ subsequences for A and B, corresponding to 0-1 strings denoting whether to select each character
- Check: suppose n < m, $O(n)$ in the worst case but on average low
- Time complexity: $O(2^{n+m})$

Problem

Given two sequences A = <a_1, a_2, a_3 , … , a_n> and B = <b_1, b_2, b_3 , … , b_m>, find their Longest Common Subsequence (not necessarily consecutive)

- **Example**: A="acbdef", B="abcdef", then LCS(A,B)="acdef" or "abdef" with the length = 5

Brute Force 2: (assume n < m) enumerate all subsequences of A, check whether it is a subsequence of B

- Enumeration: 2^n subsequences, corresponding to 0-1 strings of character selection
- Check: traverse B, compare with the leftmost unmatched character of the subsequence
- Time complexity: $\Theta(2^n m)$

```
    ret = 0
    for S in subsequences(A):
        c = 0
        for i in 0 to B.length:
            if c<S.length and B[i] == S[c]:
                c += 1
        if c == S.length:
            ret = max(ret, S.length)
```

Problem

Given two sequences A = <a_1, a_2, a_3 , … , a_n> and B = <b_1, b_2, b_3 , … , b_m>, find their Longest Common Subsequence (not necessarily consecutive)

- **Example**: A=“acbdef”, B=“abcdef”, then LCS(A,B)=“acdef” or “abdef” with the length = 5

Dynamic programming?

1. Characterize optimal substructure
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution
4. Construct optimal solution from computed information

Step 1: Optimal Substructure

Can we construct an optimal solution based on the optimal solution of subproblems?

Guess 1: Construct the LCS of A and B based on the LCS of their subsequences?

- Each subproblem can be represented as <$\{0,1\}^n, \{0,1\}^m$>
- Desired solution: LCS($\{1\}^n, \{1\}^m$)
- 2^{n+m} subproblems in total, too many values in the table & too many problems to solve

Intuition on reducing the subproblem space:

- *Usually* only consider prefix/suffix/continuous subsequences as subproblems

Step 1: Optimal Substructure

Can we construct an optimal solution based on the optimal solution of subproblems?

Guess 2: Construct the LCS of A and B based on the LCS of prefix sequences of A and B

- Each subproblem represented as $< i, j >$, where i, j represent the length of prefix
- Desired solution: $LCS(n, m)$
- $n * m$ problems in total

Can $LCS(n, m)$ be constructed with the optimal solution of $LCS(i, j)$, where $i \leq n, j \leq m$?

Step 1: Optimal Substructure

Theorem (Optimal substructure of an LCS): Let $A = <a_1, a_2, a_3, \ldots, a_n>$ and $B = <b_1, b_2, b_3, \ldots, b_m>$ be sequences, and let $C = <c_1, c_2, c_3, \ldots, c_k>$ be any LCS of A and B.

1. If $a_n = b_m$, then $c_k = a_n = b_m$ and C_{k-1} is an LCS of A_{n-1} and B_{m-1}
2. If $a_n \neq b_m$, then $c_k \neq a_n$ implies that C is an LCS of A_{n-1} and B
3. If $a_n \neq b_m$, then $c_k \neq b_m$ implies that C is an LCS of A and B_{m-1}

At least 1 case happen, so must can be constructed from optimal solution of at most two subproblems

Step 1: Optimal Substructure - Proof

1. *If a_n = b_m, then c_k = a_n = b_m and C_{k-1} is an LCS of A_{n-1} and B_{m-1}*

- ($c_k = a_n = b_m$): If $c_k \neq a_n$, appending $a_n = b_m$ to C produces a common subsequence of length $k + 1$, contradicting the fact that C is LCS of A and B
- (*C_{k-1} is an LCS of A_{n-1} and B_{m-1}*)
  - C_{k-1} is common subsequence of A_{n-1} and B_{m-1} with length $k - 1$.
  - If C_{k-1} is not LCS of A_{n-1} and B_{m-1}, A_{n-1} and B_{m-1} has a common subsequence W of length $k' > k - 1$, appending $a_n = b_m$ to W produces a common subsequence of A and B whose length is $k' + 1 > k$. Contradiction with C (length k) is LCS of A and B

Step 1: Optimal Substructure - Proof

1. *If $a_n \neq b_m$, then $c_k \neq a_n$ implies that C is an LCS of A_{n-1} and B*

- Obviously, C must be common subsequence of A_{n-1} *and* B
- If C is not LCS of A_{n-1} and B, A_{n-1} and B has a common subsequence W of length $k' > k$, W is a common subsequence of A and B with length $k' > k$, contradicting the fact that C is the LCS of A and B.

2. *If $a_n \neq b_m$, then $c_k \neq b_m$ implies that C is an LCS of A and B_{m-1}*

- The proof is symmetric to $c_k \neq a_n$

Step 2: Recurrence of Solution

Theorem (Optimal substructure of an LCS): Let $A = <a_1, a_2, a_3, \ldots, a_n>$ and $B = <b_1, b_2, b_3, \ldots, b_m>$ be sequences, and let $C = <c_1, c_2, c_3, \ldots, c_k>$ be any LCS of A and B.

1. If $a_n = b_m$, then $c_k = a_n = b_m$ and C_{k-1} is an LCS of A_{n-1} and B_{m-1}
2. If $a_n \neq b_m$, then $c_k \neq a_n$ implies that C is an LCS of A_{n-1} and B
3. If $a_n \neq b_m$, then $c_k \neq b_m$ implies that C is an LCS of A and B_{m-1}

Use f(n, m) to represent the length of LCS of prefix A_n and B_m

- Base case: $f(i, 0) = f(0, j) = 0$
- $a_n = b_m$: $f(n, m) = f(n-1, m-1) + 1$
- $a_n \neq b_m$: at least one of $LCS(n-1, m)$ and $LCS(n, m-1)$ generate $LCS(n, m)$, the longer one is the solution: $f(n, m) = \max(f(n-1, m), f(n, m-1))$

Step 3: Top-down with Memorization

- Base case: $f(i,0) = f(0,j) = 0$
- $a_n = b_m$: $f(n,m) \ = \ f(n-1, m-1) + 1$
- $a_n \neq b_m$: $f(n,m) \ = \ \max(f(n-1,m), f(n,m-1))$

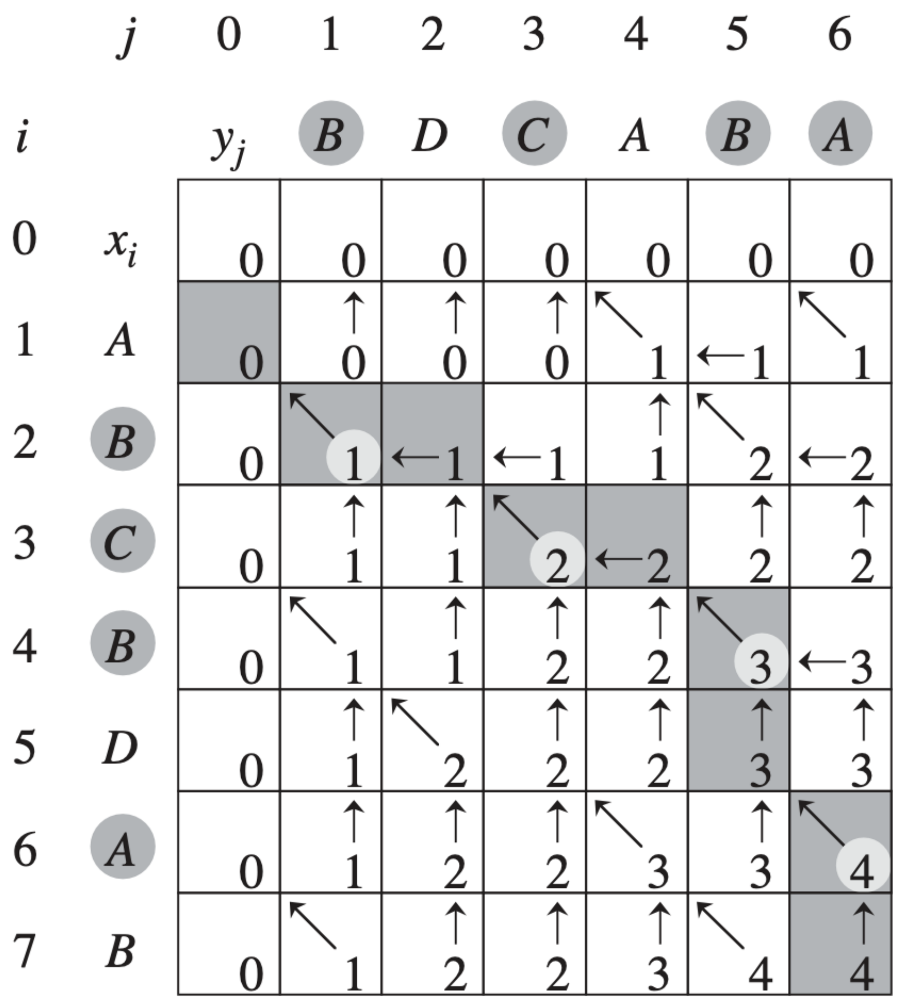
```
LCS[0…N][0…M] = -inf
LCS[0][…] = LCS[…][0] = 0
def f(i,j):
    if LCS[i][j] != -inf:
        return LCS[i][j]
    if A[i-1] == B[j-1]:
        LCS[i][j] = f(i - 1,j - 1) + 1
    else:
        LCS[i][j] = max(f(i-1, j), f(i, j-1))
    return LCS[i][j]
```

Step 3: Bottom-Up

- Base case: $f(i,0) = f(0,j) = 0$
- $a_n = b_m$: $f(n,m) \ = \ f(n-1, m-1) + 1$
- $a_n \neq b_m$: $f(n,m) \ = \ \max(f(n-1,m), f(n,m-1))$

```
1. LCS[1…N][1…M] = -inf
2. LCS[0][…] = LCS[…][0] = 0
3. for i from 1 to n:
4.     for j from 1 to m:
5.         if A[i-1] == B[j-1]:
6.             LCS[i][j] = LCS[i-1][j-1] + 1
7.         else:
8.             LCS[i][j] = max(LCS[i - 1][j], LCS[i][j – 1])
```

Step 4: Reconstruct an LCS



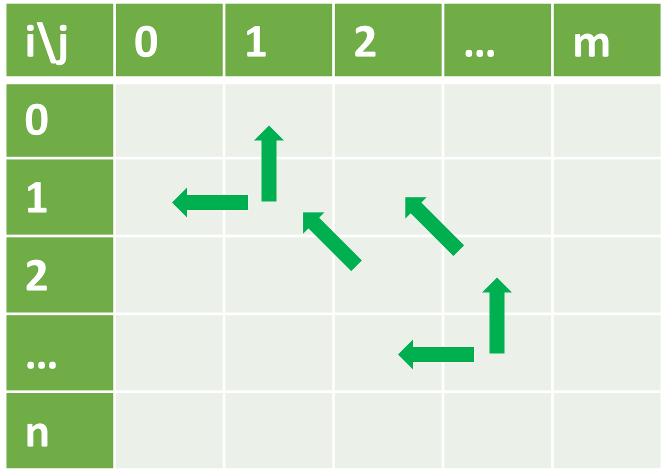
i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

Record the state that leads to the current optimal

```
1.  LCS[N][M] = -inf
2.  LCS[0][…] = LCS[…][0] = 0
3.  for i from 1 to n:
4.      for j from 1 to m:
5.          if A[i-1] == B[j-1]:
6.              LCS[i][j] = LCS[i-1][j-1] + 1
7.              s[i][j] = (i-1, j-1)
8.          else:
9.              if LCS[i - 1][j] > LCS[i][j – 1]:
10.                 LCS[i][j] = LCS[i - 1][j], s[i][j] = (i-1, j)
11.             else:
12.                 LCS[i][j] = LCS[i][j – 1], s[i][j] = (i, j-1)
```

Trace back through s from the final problem $< n, m >$

Complexity



i\j	0	1	2	…	m
0					
1					
2					
…					
n					

- At worst solve all the $n * m$ subproblems
- Each have at most 2 dependent subproblems (edges)
- Time Complexity: $\Theta(n * m)$
- Space Complexity: $\Theta(n * m)$

Longest Increasing Subsequence

Longest Increasing Subsequence (LIS)

Problem: given a sequence S = <s_1, s_2, s_3 , … , s_n>, find the longest subsequence where the elements are in ascending order

- **Example**: For 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15, a longest subsequence is 0, 2, 6, 9, 11, 15

Brute Force: enumerate all subsequences of A

- Enumeration: 2^n 0-1 strings denoting whether to select each character in A
- Check the order: at most $O(n)$
- $\mathrm{O}(2^n n)$

Longest Increasing Subsequence (LIS)

Problem: given a sequence S = <s_1, s_2, s_3 , … , s_n>, find the longest subsequence where the elements are sorted in ascending order

- **Example**: For 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15, a longest subsequence is 0, 2, 6, 9, 11, 15

Dynamic programming?

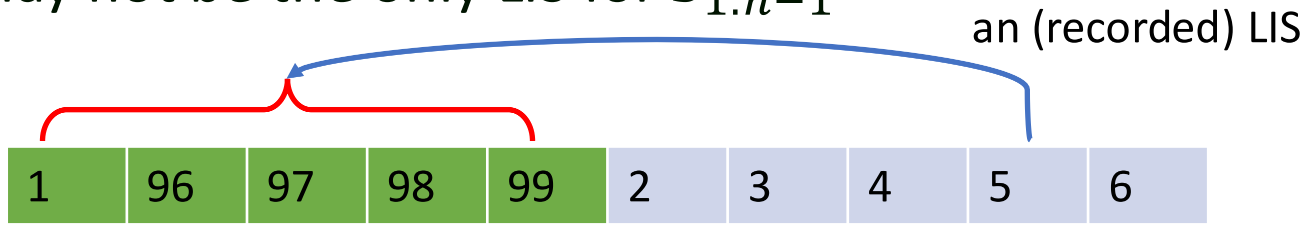
1. Characterize optimal substructure
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution
4. Construct optimal solution from computed information

Step 1: Optimal Substructure

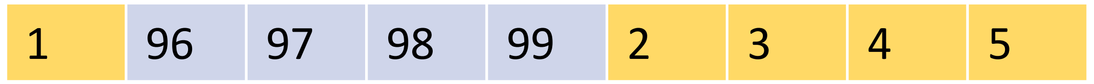
Can we construct an optimal solution based on the optimal solution of subproblems?

Guess: Construct LIS of S $=< s_1, s_2, \dots, s_n >$ based on an LIS $C =< c_1, c_2, \dots\ c_k >$ of $S_{1:n-1}$?

- If $c_k \leq s_n$: appending s_n to C makes an LIS
- If $c_k > s_n$, there is still possibility that S has an LIS of length $k+1$
  - C may not be the only LIS for $S_{1:n-1}$



- We need to know all the LIS of $S_{1:n-1}$ to construct an LIS at the next position
- Construct all the LIS for all the prefix sequences?
  - Problem: given an LIS C' of S, $C'_{1:k-1}$ may not be an LIS of any prefix sequence



Step 1: Optimal Substructure

A relevant problem that solves the original problem: given a sequence $S = < s_1, s_2, s_3, \ldots, s_n >$, find the LIS that ends at position k

- LIS of S can be found among LISs ending at each position: find $\max_{k=1}^{n} LIS_k(S).length$

Optimal substructure: If $C_i = < s_{c_1}, s_{c_2}, \ldots, s_{c_{k-1}}, s_i >$ is an LIS ending at i for $S =< s_1, s_2, \ldots, s_n >$, then $< s_{c_1}, s_{c_2}, \ldots, s_{c_{k-1}} >$ is an LIS ending at c_{k-1} for S

Proof:

- $< s_{c_1}, s_{c_2}, \ldots, s_{c_{k-1}} >$ must be an increasing subsequence that ends at position c_{k-1}
- If a longer increasing subsequence ending at c_{k-1}exists: $< s_{c'_1}, s_{c'_2}, \ldots, s_{c_{k-1}} >$, $< s_{c'_1}, s_{c'_2}, \ldots, s_{c_{k-1}}, s_i >$ is a longer subsequence ending at i, contradicting the fact that C_i is LIS

Step 1: Optimal Substructure

Optimal substructure: If $C_i = < s_{c_1}, s_{c_2}, \dots, s_{c_{k-1}}, s_i >$ is an LIS ending at i for $S =< s_1, s_2, \dots, s_n >$, then $< s_{c_1}, s_{c_2}, \dots, s_{c_{k-1}} >$ is an LIS ending at c_{k-1} for S

- Traversing all the LIS ending before i and try appending s_i can produce the LIS ending at i
- All the LIS ending at c_{k-1} are the same for i, they form an IS ending at i of fixed length as long as $c_{k-1} < s_i$ $\Rightarrow$ only need 1 LIS for each position

Step 2: Recurrence of Solution

Optimal substructure: If $C_i = < s_{c_1}, s_{c_2}, \dots, s_{c_{k-1}}, s_i >$ is an LIS ending at i for $S =< s_1, s_2, \dots, s_n >$, then $< s_{c_1}, s_{c_2}, \dots, s_{c_{k-1}} >$ is an LIS ending at c_{k-1} for S

- Use $f(i)$ to represent the length of the LIS ending at s_i

$$f(i) = \max(1, \max_{j<i, s_j \le s_i} (f(j) + 1))$$

Step 3: Implementation

Use $f(i)$ to represent the length of the LIS ending at s_i

$$f(i) = \max(1, \max_{j<i, a_j \le a_i} (f(j) + 1))$$

Top-down with Memorization

```
1.  LIS[1…N] = -inf
2.  f(i):
3.      if(LIS[i] != -inf):
4.          return LIS[i]
5.      LIS[i] = 1
6.      for j from 1 to i-1:
7.          if A[j] < A[i]:
8.              LIS[i] = max(LIS[i], f(j) +1)
9.  return LIS[i]
10. ret = 1
11. for i from 1 to N:
12.   ret = max(ret, f(i))
```

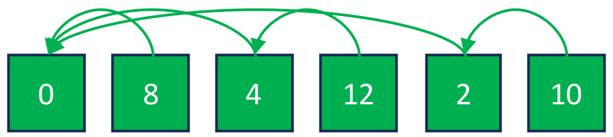
Bottom-Up

```
1.  LIS[1…N] = 1
2.  for i from 1 to N:
3.      for j from 1 to i-1:
4.          if A[j] < A[i]:
5.              LIS[i] = max(LIS[i], LIS[j] + 1)
6.      ret = max(ret, LIS[i])
```

Step 4: Reconstruct an LIS

Record the state that leads to the current optimal

```
1.  LIS[1…N] = 1
2.  s[1..N] = -1
3.  ret_p = 0
4.  for i from 1 to N:
5.      s[i] = i
6.      for j from 1 to i-1:
7.          if A[j] < A[i]:
8.              if LIS[i] < LIS[j] + 1:
9.                  s[i] = j
10.                 LIS[i] = LIS[j] + 1
11.     if LIS[i] > ret:
12.         ret = LIS[i]
13.         ret_p = i
```



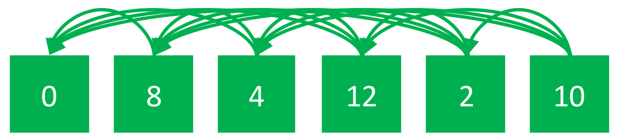
Reconstruction: Trace back through s from the final problem ret_p

Complexity

- In the worst case solve all the n subproblems
- The i-th subproblem depends on $i-1$ subproblems
- In total: $\sum_{i=1}^{n-1} i = n^2$ edges

Time complexity: $\Theta(n^2)$

Space complexity: $\Theta(n)$



Faster Solution

Use $f(i)$ to represent the length of the LIS ending at s_i

$$f(i) = \max(1, \max_{j<i, s_j \le s_i} (f(j) + 1))$$

Observation

$$f(i) = \max(1, \max_{j<i, s_j \le s_i} f(j) + 1)$$

The LIS ending with a_i is constructed based on $\max_{j<i, s_j \le s_i} f(j)$

- Maximum length of the increasing subsequence found *so far* that ends with a value *less than the current s_i*

Faster Solution

$$f(i) = \max(1, g(i) + 1)$$

- $g(i)$: Maximum length of the increasing subsequence found *so far* that ends with a value *less than the current s_i*

Efficiently obtaining $g(i)$: store the minimal ending value for each length of increasing subsequence so far (in an array $B[1 \dots N]$), then $g(i) = \max_{B_l \le s_i} l$.

- Characteristic: B is increasing
  - If there exists $i < j$ that $B_i > B_j$, there is an increasing sequence J with length j and $J_j = B_j$. $J_{1:i}$ is an increasing sequence and $J_i \le B_j < B_i$, contradicting the fact that B_i is the minimal ending value of an increasing subsequence with length i
- Find $\max_{B_l \le s_i} l$?

Binary search! $O(\log n)$

Faster Solution

$$f(i) = \max(1, g(i) + 1)$$

- $g(i)$: Maximum length of the increasing subsequence found *so far* that ends with a value *less than the current s_i*

Efficiently obtaining $g(i)$: store the minimal ending value for each length of increasing subsequence so far (in an array $B[1 \dots N]$), then $g(i) \; = \max_{B_l \leq s_i} l$.

- Update B after traversing s_i?
  - We have new increasing subsequences of length $1, 2, \dots, g(i) + 1$ that ends with s_i
  - What values in B can we update?
    - Observation: $B_1 \leq B_2 \leq \cdots \leq B_{g(i)} \leq s_i < B_{g(i)+1} < \cdots$

Only $B_{g(i)+1}$ needs to be updated! $\Theta(1)$

Faster Solution

$$f(i) = \max(1, g(i) + 1)$$

- $g(i)$: Maximal length of increasing subsequence so far that ends with a value smaller than the current s_i

Efficiently obtaining $g(i)$: store the minimal ending value for each length of increasing subsequence so far (in an array $B[1 \ldots N]$), then $g(i) = \max_{B_l \le s_i} l$.

- Find $\max_{B_l \le s_i} l$: Binary search $\Theta(\log n)$
- Update B after traversing s_i: Update $B_{g(i)+1}$: $\Theta(1)$
- In total: $\Theta(n\log n)$

DP on Trees

Banquet Without Bosses

Problem: A company holds a banquet and will let a subset out of n employees to attend. The i-th employee attending the banquet has happiness score r_i. Given the organization tree of these employees. The principal is the root, each employee's supervisor is the parent. **An employee and their supervisor cannot simultaneously attend the banquet**. Set the plan to invite employees to maximize the total happiness score of the banquet.

Use dynamic programming to solve problems on tree structures

- Define optimal substructure in terms of subtrees

Banquet Without Bosses

Step 1: Optimal substructure

Can we construct an optimal solution based on the optimal solution of subproblems?

Intuition: treat each subtree as a subproblem

- If the root attends, we optimize the solution by getting the optimal solution for all its subtrees, constraining that each of them cannot have the root attends, plus the happiness of the root
- If does not attend, we optimize the solution by getting the optimal solution for all its subtrees

One of the two cases happens

Define subproblem: get maximal happiness score of the tree when the root attend/not attend the banquet.

- <node, {0/1}>: where 0 represents the root does not attend, 1 represents attend
- Maximal between optimal solution of <node, 0> and <node, 1> is the optimal

Optimal substructure: Given any child of a node,

- The optimal solution for <node, 0> contain either optimal solution of <child, 1> or <child, 0>.
- The optimal solution for <node, 1> contain optimal solution of <child, 0>

Banquet Without Bosses

Step 2: Recurrence of value

Let $f(u, s)$ be the maximum happiness for subtree rooted on u when u's state is $s \in \{0,1\}$

$$f(u, 0) = \sum_{x \in u.children} \max\left(f(x, 0), f(x, 1)\right)$$

$$f(u, 1) = \sum_{x \in u.children} f(x, 0) + r_u$$

Banquet Without Bosses

Step 3: Top-Down with Memorization (Trees are naturally recursive)

```
dp[0…n][2] = 0
Def f(u, s):
    if dp[u][s] != 0: return dp[u][s]
    dp[u][s] = s * r[u]
    for v in get_children(u):
        if s == 0:
            dp[u][s] += max(f(v, 0), f(v, 1))
        else:
            dp[u][s] += f(v, 0)
print(max(f(root, 0), f(root, 1))
```

Reconstruction?

Bottom-Up?

- Sort the nodes in terms of the depth in decreasing order

Complexity Analysis

- Nodes: $2 * n$ subproblems
- Edges: each rely on at most 2#children, there are n children in total in a tree

Time Complexity: $\Theta(n)$

Space Complexity

- Store N edges on the organization tree: $\Theta(n)$
- Auxiliary space complexity for dynamic programming $\Theta(n)$

Knapsack Problem

Knapsack Problem

Problem: Given a Knapsack with an integer capacity W. Given N items, each with an integer volume w_i and a value v_i. Select a subset of these items to place into the knapsack such that the total value is maximized without exceeding the knapsack's capacity.

$$\max \sum_{i=1}^{n} x_i v_i \qquad s.t. \sum_{i=1}^{n} x_i w_i \leq W$$

Table-1 item's weight-value table

index	1	2	3	4	5
Weight	3	4	7	2	5
value	6	10	4	3	12

Knapsack Problem

Problem: Given a Knapsack with an integer capacity W. Given N items, each with an integer volume w_i and a value v_i. Select a subset of these items to place into the knapsack such that the total value is maximized without exceeding the knapsack's capacity.

$$\max \sum_{i=1}^{n} x_i v_i \qquad s.t. \sum_{i=1}^{n} x_i w_i \leq W$$

Knapsack problem is NP-hard, but can have a Pseudo-polynomial time solution

- Polynomial time: polynomial in the length of the input, irrelevant to the scale of values
- Pseudo-polynomial time: polynomial in the value of the input (the largest integer present in the input), e.g., W in knapsack

Can be solved when W is small

Knapsack Problem

Problem: Given a Knapsack with an integer capacity W. Given N items, each with an integer volume w_i and a value v_i. Select a subset of these items to place into the knapsack such that the total value is maximized without exceeding the knapsack's capacity.

$$\max \sum_{i=1}^{n} x_i v_i \qquad s.t. \sum_{i=1}^{n} x_i w_i \leq W$$

Different types consider the scope of x

- 0-1 knapsack problem
- Bounded knapsack problem
- Unbounded knapsack problem
- …

0-1 Knapsack Problem

Problem: Given a Knapsack with an integer capacity W. Given N items, each with an integer volume w_i and a value v_i. Select a subset of these items to place into the knapsack such that the total value is maximized without exceeding the knapsack‘s capacity.

$$\max \sum_{i=1}^{n} x_i v_i \qquad s.t. \sum_{i=1}^{n} x_i w_i \le W\,, \qquad x_i \in \{0,1\}$$

0-1 Knapsack Problem: every item can be picked at most once

Brute Force: enumerate all possible item combination and select the one with highest value

- Complexity: 2^n combinations for n items in the worst case, $O(2^n)$

Take advantage of the fact that W is not large and use dynamic programing?

0-1 Knapsack Problem

Step 1: Optimal substructure

Can we construct an optimal solution based on the optimal solution of subproblems?

Let $X = \arg\max \ \sum_{i=1}^{n} x_i v_i \quad s.t. \sum_{i=1}^{n} x_i w_i \leq W;\ x_i \in \{0,1\}$, we can prove that

- if $x_n = 1$: $X_{1:n-1} = \arg\max \sum_{i=1}^{n-1} x_i v_i \qquad s.t. \sum_{i=1}^{n-1} x_i w_i \leq W - w_n\ ,\ x_i \in \{0,1\}$
- if $x_n = 0$: $X_{1:n-1} = \arg\max \sum_{i=1}^{n-1} x_i v_i \qquad s.t. \sum_{i=1}^{n-1} x_i w_i \leq W\ ,\ x_i \in \{0,1\}$

Intuition: For the last item,

- If I pick it, the maximum value I can get is its value + the maximum value I can get from picking the previous items with the rest of space in the knapsack
- If I do not pick it, the maximum value I can get is the maximum value I can get from picking the previous items in the knapsack

0-1 Knapsack Problem

Step 1: Optimal substructure

Can we construct an optimal solution based on the optimal solution of subproblems?

Let $X = \arg\max \sum_{i=1}^{n} x_i v_i \quad s.t. \sum_{i=1}^{n} x_i w_i \leq W;\ x_i \in \{0,1\}$, we can prove that

- if $x_n = 1$: $X_{1:n-1} = \arg\max \sum_{i=1}^{n-1} x_i v_i \quad s.t. \sum_{i=1}^{n-1} x_i w_i \leq W - w_n,\ x_i \in \{0,1\}$
- if $x_n = 0$: $X_{1:n-1} = \arg\max \sum_{i=1}^{n-1} x_i v_i \quad s.t. \sum_{i=1}^{n-1} x_i w_i \leq W,\ x_i \in \{0,1\}$

Proof by contradiction:

- if $x_n = 1$: if there exists $\mathrm{X'}_{1:n-1}$ with $\sum_{i=1}^{n-1} x_i' v_i > \sum_{i=1}^{n-1} x_i v_i$ and $\sum_{i=1}^{n-1} x_i' w_i \leq W - w_n$, then $\mathrm{X'} =< x_1', x_2', \ldots, x_{n-1}', 1 >$ becomes a better solution with $\sum_{i=1}^{n} x_i' v_i > \sum_{i=1}^{n} x_i v_i$ and $\sum_{i=1}^{n} x_i' w_i \leq W$
- if $x_n = 0$: if there exists $\mathrm{X'}_{1:n-1}$ with $\sum_{i=1}^{n-1} x_i' v_i > \sum_{i=1}^{n-1} x_i v_i = \sum_{i=1}^{n} x_i v_i$ and $\sum_{i=1}^{n-1} x_i' w_i \leq W$, then $\mathrm{X'} =< x_1', x_2', \ldots, x_{n-1}', 0 >$ makes a better solution with $\sum_{i=1}^{n} x_i' v_i > \sum_{i=1}^{n} x_i v_i$ and $\sum_{i=1}^{n} x_i' w_i \leq W$

0-1 Knapsack Problem

Step 1: Optimal substructure

Can we construct an optimal solution based on the optimal solution of subproblems?

Let $X = \arg\max \sum_{i=1}^{n} x_i v_i \quad s.t. \sum_{i=1}^{n} x_i w_i \le W;\ x_i \in \{0,1\}$, we can prove that

- if $x_n = 1$: $X_{1:n-1} = \arg\max \sum_{i=1}^{n-1} x_i v_i \quad s.t. \sum_{i=1}^{n-1} x_i w_i \le W - w_n,\ x_i \in \{0,1\}$
- if $x_n = 0$: $X_{1:n-1} = \arg\max \sum_{i=1}^{n-1} x_i v_i \quad s.t. \sum_{i=1}^{n-1} x_i w_i \le W,\ x_i \in \{0,1\}$

Optimal substructure: The optimal solution of subproblem (n, W) can be obtained from optimal solution of problem $(n-1, W - w_n)$ or $(n-1, W)$

0-1 Knapsack Problem

Step 2: Define the recurrence of value

Let $f(i,j)$ be the maximum value we can obtain with capacity j by selecting the first i items:

$$f(i,j) = \max(f(i-1,j), f(i-1,j-w_i) + v_i)$$

Consider base case and when $j < w_i$

$$f(i,j) = \begin{cases} 0 & , \quad if\ i = 0\ or\ j = 0 \\ f(i-1,j) & , \quad if\ j < w_i \\ \max(f(i-1,j), f(i-1,j-w_i) + v_i) & , \quad if\ j \geq w_i \end{cases}$$

0-1 Knapsack Problem

Step 3: Top-down with Memorization

```
1.  dp[1…N][1…W] = -inf
2.  Def f(i, j):
3.      if dp[i][j] != -inf: return dp[i][j]
4.      if i ==0 or j == 0:
5.          dp[i][j] = 0
6.          return dp[i][j]
7.      else:
8.          if j < $w_i$:  dp[i][j] = f(i - 1, j)
9.          else:  dp[i][j] = max(f(i - 1, j), f(i - 1, j - $w_i$) + $v_i$)
10.     return dp[i][j]
11. print f(N, W)
```

0-1 Knapsack Problem

Step 3: Bottom-Up

```
1. dp[1…N][1…W] = -inf
2. for i in range(0, K+1):
3.     for j range(0, W+1):
4.         if i == 0 or j == 0: dp[i][j] = 0
5.         else if j < $w_i$ : dp[i][j] = dp[i - 1][j]
6.         else dp[i][j] = max(dp[i – 1][j], dp[i-1][j - $w_i$] + $v_i$)
7. print dp[N][W]
```

0-1 Knapsack Problem

Step 4: reconstruction

- Store whether you pick the current item i in subproblem (i, j)
- Backtracking the subproblems to optimize the current state

```
1. dp[1…N][1…W] = -inf
2. for i in range(0, K+1):
3.     for j in range(0, W+1):
4.         if i == 0 or j == 0: dp[i][j] = 0
5.         else if j < $w_i$: dp[i][j] = dp[i - 1][j], s[i][j] = 0
6.         else:
7.             if dp[i – 1][j] > dp[i-1][j - $w_i$] + $v_i$: dp[i][j] = dp[i - 1][j], s[i][j] = 0
8.             else: dp[i][j] = dp[i-1][j - $w_i$] + $v_i$, s[i][j] = 1
9. print dp[N][W]
```

```
1. dp[1…N][1…W] = -inf
2. x[1…N] = 0
3. i=N, j=W
4. while i > 0:
5.     x[i] = s[i][j]
6.     j = j - $w_i$*s[i][j]
7.     i -= 1
```

0-1 Knapsack Problem

Complexity

- Nodes: $n*W$ subproblems
- Edges: each rely on 2 subproblems: $2nW$

Time Complexity: $O(n*W)$

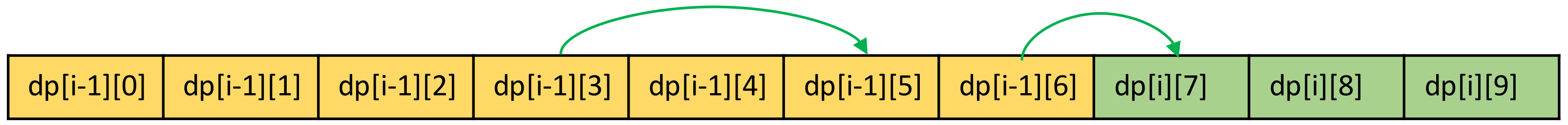
Space complexity: $O(n*W)$

Can we reduce space complexity?

Observation:

- dp[1..i-2][…] are not used when you calculating dp[i][…]
- Only the last row of the table is used!

Can we only keep the last row?



```
1. dp[1…N][1…W] = -inf
2. for i in range(0, K+1):
3.     for j in range(0, W+1):
4.         if i == 0 or j == 0: dp[i][j] = 0
5.         else if j < $w_i$ : dp[i][j] = dp[i - 1][j]
6.         else dp[i][j] = max(dp[i – 1][j], dp[i-1][j - $w_i$] + $v_i$)
7. print dp[N][W]
```

```
1. dp[1…W] = -inf
2. for i in range(0, K+1):
3.     for j in from W to 0:    # Update from the no longer used slots
4.         if i == 0 or j == 0: dp[j] = 0
5.         else if j < $w_i$ : dp[j] = dp[j]
6.         else dp[j] = max(dp[j], dp[j - $w_i$] + $v_i$)
7. print dp[W]
```

Unbounded Knapsack Problem

Problem: Given a Knapsack with an integer capacity W. Given N items, each with an integer volume w_i and a value v_i. Select a subset of these items to place into the knapsack such that the total value is maximized without exceeding the knapsack‘s capacity

$$\max \sum_{i=1}^{n} x_i v_i \qquad s.t. \sum_{i=1}^{n} x_i w_i \le W\,, \qquad x_i \in N$$

Brute Force: enumerate all possible item combination and select the one with highest value

- Even larger complexity than 0-1 knapsack problem

Supposing W is not large, dynamic programing?

Unbounded Knapsack Problem

Step 1: Optimal substructure

Can we construct an optimal solution based on the optimal solution of subproblems?

Let $X = \arg\max \sum_{i=1}^{n} x_i v_i \quad s.t. \sum_{i=1}^{n} x_i w_i \leq W$, we can prove that

- if $x_n = 0$: $X_{1:n-1} = \arg\max \sum_{i=1}^{n-1} x_i v_i \quad s.t. \sum_{i=1}^{n-1} x_i w_i \leq W$,
- if $x_n \neq 0$: $< X_{1:n-1}, x_n - 1 >= \arg\max \sum_{i=1}^{n} x_i v_i \quad s.t. \sum_{i=1}^{n} x_i w_i \leq W - w_n$,

Intuition: For the last item,

- If I pick 0, the maximum value I can get is the maximum value I can get from picking the previous items in the knapsack
- If I pick at least 1, the maximum value I can get is its value + the maximum value I can get from picking from this item and the previous items with the rest of space in the knapsack

Unbounded Knapsack Problem

Step 1: Optimal substructure

Can we construct an optimal solution based on the optimal solution of subproblems?

Let $X = \arg\max \sum_{i=1}^{n} x_i v_i \quad s.t. \sum_{i=1}^{n} x_i w_i \le W$, we can prove that

- if $x_n = 0$: $X_{1:n-1} = \arg\max \sum_{i=1}^{n-1} x_i v_i \quad s.t. \sum_{i=1}^{n-1} x_i w_i \le W$,
- if $x_n \neq 0$: $< X_{1:n-1}, x_n - 1 >= \arg\max \sum_{i=1}^{n} x_i v_i \quad s.t. \sum_{i=1}^{n} x_i w_i \le W - w_n$

Prove by contradiction:

- If $x_n = 0$: if there exists $\mathrm{X'}_{1:\mathrm{n}-1}$ with $\sum_{i=1}^{n-1} x_i' v_i > \sum_{i=1}^{n-1} x_i v_i = \sum_{i=1}^{n} x_i v_i$ and $\sum_{i=1}^{n-1} x_i' w_i \le W$, then $\mathrm{X'} =< x_1', x_2', \dots, x_{n-1}', 0 >$ makes a better solution with $\sum_{i=1}^{n} x_i' v_i > \sum_{i=1}^{n} x_i v_i$ and $\sum_{i=1}^{n} x_i' w_i \le W$
- If $x_n \neq 0$: if there exists X' with $\sum_{i=1}^{n} x_i' v_i > \sum_{i=1}^{n} x_i v_i - v_n$ and $\sum_{i=1}^{n} x_i' w_i \le W - w_n$, then $\mathrm{X''} =< x_1', x_2', \dots, x_{n-1}', x_n' + 1 >$ becomes a better solution with $\sum_{i=1}^{n} x_i'' v_i > \sum_{i=1}^{n} x_i v_i$ and $\sum_{i=1}^{n} x_i'' w_i \le W$

Unbounded Knapsack Problem

Step 1: Optimal substructure

Can we construct an optimal solution based on the optimal solution of subproblems?

Let $X = \arg\max \ \sum_{i=1}^{n} x_i v_i \qquad s.t. \sum_{i=1}^{n} x_i w_i \leq W$, we can prove that

- if $x_n \neq 0$: $< X_{1:n-1}, x_n - 1 >= \arg\max \sum_{i=1}^{n} x_i v_i \qquad s.t.\ \sum_{i=1}^{n} x_i w_i \leq W - w_n$,
- if $x_n = 0$: $X_{1:n-1} = \arg\max \sum_{i=1}^{n-1} x_i v_i \qquad s.t.\ \sum_{i=1}^{n-1} x_i w_i \leq W$,

There must be one case that happens. The optimal solution of problem $(n,\ W)$ must contain the optimal solution of problem $(n-1, W)$, $(n,\ W - w_n)$

Unbounded Knapsack Problem

Step 2: Recurrence of value

Let $f(i,j)$ be the maximum value we can obtain with capacity j by selecting the first i items:

$$f(i,j) = \max(f(i-1,j), f(i,j-w_i)+v_i)$$

$$dp[i][j] = \begin{cases} 0 & , \quad if\ i = 0\ or\ j = 0 \\ dp[i-1][j] & , \quad if\ j < w_i \\ \max(dp[i-1][j], dp[i][j-w_i]+v_i) & , \quad if\ j \geq w_i \end{cases}$$

Unbounded Knapsack Problem

Step 3: Top-Down with Memorization

```
1.  dp[1…N][1…W] = -inf
2.  Def f(i, j):
3.      if dp[i][j] != -inf: return dp[i][j]
4.      if i ==0 or j == 0:
5.          dp[i][j] = 0
6.          return dp[i][j]
7.      else:
8.          if j < $w_i$:  dp[i][j] = f(i - 1, j)
9.          else:  dp[i][j] = max(f(i - 1, j), f(i, j - $w_i$) + $v_i$)
10.     return dp[i][j]
11. print f(N, W)
```

Unbounded Knapsack Problem

Step 3: Bottom-Up

```
1. dp[1…N][1…W] = -inf
2. for i in range(0, K+1):
3.     for j range(0, W+1):
4.         if i == 0 or j == 0: dp[i][j] = 0
5.         else if j < $w_i$ : dp[i][j] = dp[i - 1][j]
6.         else dp[i][j] = max(dp[i – 1][j], dp[i][j - $w_i$] + $v_i$)
7. print dp[N][W]
```

Unbounded Knapsack Problem

Complexity

- Nodes: n * W subproblems
- Edges: each rely on 2 subproblems: 2nW

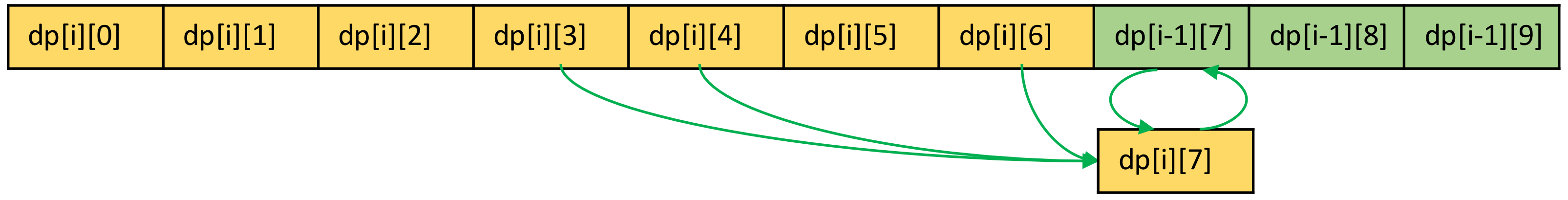
Time Complexity: $O(nW)$

Space complexity: $O(nW)$

Can we reduce space complexity?

Observation:

- dp[i][j] uses dp[i][1 to j-1] and dp[i-1][j]
- After getting dp[i][j], dp[i-1][j] is no longer used
- Can we only keep dp[i][1 to j-1] and dp[i-1][j to W]?



```
1. dp[1…N][1…W] = -inf
2. for i in range(0, K+1):
3.     for j range(0, W+1):
4.         if i == 0 or j == 0: dp[i][j] = 0
5.         else if j < $w_i$ : dp[i][j] = dp[i - 1][j]
6.         else dp[i][j] = max(dp[i – 1][j], dp[i][j - $w_i$] + $v_i$)
7. print dp[N][W]
```

```
1. dp[1…W] = -inf
2. for i in range(0, K+1):
3.     for j range(0, W+1):
4.         if i == 0 or j == 0: dp[j] = 0
5.         if j > $w_i$:
6.             dp[j] = max(dp[j], dp[j - $w_i$] + $v_i$)
7. print dp[W]
```

# Update from problems that future problems depends on

Assignment

- Given a set of distinct positive integers, find the largest subset such that every pair (S_i, S_j) of elements in this subset satisfies: S_i % S_j = 0 or S_j % S_i = 0. Please return the largest size of the subset.
  - Hint: S_i % S_j = 0 means that S_i is divisible by S_j .
- Given n, how many structurally unique BST's (binary search trees) that store values 1...n?
  - Explanation: Given n = 3, there are a total of 5 unique BST's:

```
   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

Thank you!

AIAA 5037 Advanced Algorithms and Data Structures