Data Structures and Algorithms

Lecture 05 – Maximum Subarray Sum Problem

(Dynamic Programming & Divide and Conquer)

Pengju Ren
Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University

http://gr.xjtu.edu.cn/web/pengjuren

Problem Solving

Given the daily stock *price changes*, find the maximum subarray sum, and the corresponding buy and sell dates.

Solution1：Brute Force

```
def max_subarray_brute_force(nums):
    n = len(nums)
    max_sum = float('-inf')

    # 枚举起点i和终点j的所有可能子数组：[0,-],[1-],[2-] ……，然后求解其最大值;
    for i in range(n):
        current_sum = 0
        for j in range(i, n):
            current_sum += nums[j]
            if current_sum > max_sum:
                max_sum = current_sum
    return max_sum
```

Solution2：Brute Force(optimized)

```
def max_subarray_brute_force_optimized(nums):
    n = len(nums)
    max_sum = float('-inf')
    # 计算前缀和
    prefix_sum = [0] * (n+1)
    for i in range(1, n+1):
        prefix_sum[i] = prefix_sum[i-1] + nums[i-1]

    # 枚举所有子数组，通过前缀和快速计算和
    for i in range(n):
        for j in range(i, n):
            # 子数组nums[i..j]的和 = prefix_sum[j+1] - prefix_sum[i]
            current_sum = prefix_sum[j+1] - prefix_sum[i]
            if current_sum > max_sum:
                max_sum = current_sum

    return max_sum
```

Solution3：Dynamic Programming (kadane)

```
def max_subarray_kadane(nums):
    if not nums:
        return 0
    # 初始化
    max_ending_here = nums[0]   # 以当前元素结尾的最大子数组
    max_so_far = nums[0]        # 全局最大子数组和

    # 遍历数组
    for i in range(1, len(nums)):
        # 关键状态转移：以nums[i]结尾的最大子数组和
        # 要么是nums[i]自己，要么是前面的最大子数组和加上nums[i]
        max_ending_here = max(nums[i], max_ending_here + nums[i])
        # 更新全局最大值
        max_so_far = max(max_so_far, max_ending_here)
    return max_so_far
```

Solution4：Divide and Conquer

```
def divide_and_conquer(left, right):
        # 基准情况：只有一个元素
        if left == right:
            return nums[left]
        # 计算中间位置
        mid = (left + right) // 2
        # 递归求解左右两部分
        left_max = divide_and_conquer(left, mid)
        right_max = divide_and_conquer(mid + 1, right)
        # 计算跨越中间的最大子数组和
        # 从中间向左扩展
        left_cross_max = float('-inf')
        current_sum = 0
        for i in range(mid, left - 1, -1):
            current_sum += nums[i]
            left_cross_max = max(left_cross_max, current_sum)
        # 从中间向右扩展
        right_cross_max = float('-inf')
        current_sum = 0
        for i in range(mid + 1, right + 1):
            current_sum += nums[i]
            right_cross_max = max(right_cross_max, current_sum)
        # 跨越中间的最大和
        cross_max = left_cross_max + right_cross_max
        # 返回三者中的最大值
        return max(left_max, right_max, cross_max)
```

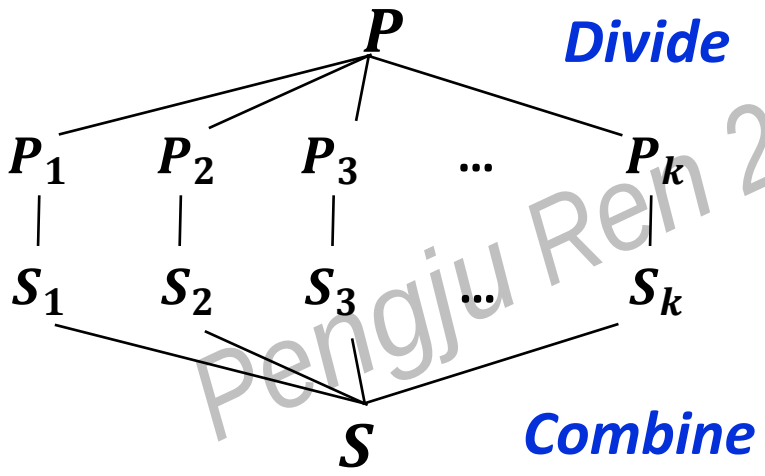
Extension

- **Question 1: Is it possible to use a parallel algorithm to accelerate the problem?**
- **Question 2: How to solve the problem if the array is circular (i.e., the beginning and end are connected)?**
- **Question 3: How to extend the maximum subarray sum problem to two dimensions (matrix)?**

Divide and Conquer

What is Divide and Conquer Algorithm

A design paradigm which works by recursively breaking down a problem into *subproblems of similar type* until these become simple enough to be solved directly, then solutions to the subproblems are *combined* to give a solution to the original one




```
Div_and_Con(P)
    if (small(p)):
        return Solution(P)
    else:
        divide P into P_1, P_2,... P_k
        Div_and_Con(P_i)
        Combine(S_1, S_2,... S_k)
```


Example:* *Binary Search, Finding Maximum/Minimum, Merge Sort, Quick Sort, Closest Pair of Points, Karatsuba &Toom-Cook, Strassen's Matrix Multiplication

Time Complexity of Divide and Conquer

```
fib1(int n)
{
  if n<=2;
     return 1, 1;
  else
     a, b = fib1(n-1);
     return b, a+b;
}
```

$T(n) = T(n-1) + 1$

$T(n) = T(n-k) + C$

$T(n) = T(n-k) + \log n$

$T(n) = T(n-k) + n$

```
fib2(int n)
{
  if n<=2;
     return 1;
  else
     return fib(n-1)+fib(n-2);
}
```

$T(n) = 2T(n-1) + 1$

***Decreasing* Function**

```
def rec_bsearch(arr,tar,lt=None,rt=None):
if lt is None: //lt is short for left
        lt = 0
if rt is None: //rt is short for right
        rt = len(arr) - 1
if lt > rt:
        return -1
mid = (lt + rt) // 2
if arr[mid] == tar:
   return mid
elif arr[mid] < tar:
   return rec_bsearch(arr,tar,mid + 1,rt)
else:
   return rec_bsearch(arr,tar,lt,mid - 1)
```

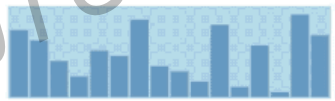
$T(n) = T(n/2) + 1$

***Dividing* Function**

Example of D&C Algorithms

Recap *Merge Sorting*: divide the array in *two halves* and keep halving recursively until they become one element, merge halves by sorting them

$$T(n) = 2T(n/2) + O(n)$$



```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

return merge(left, right)
```

```
def merge(left, right):
    result = []
    i = j = 0
    while i<len(left) and j<len(right):
       if left[i] <= right[j]:
          result.append(left[i])
          i += 1
       else:
          result.append(right[j])
          j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Example of D&C Algorithms

Recap *Quick Sorting*: Selecting a *pivot element* from the array and partitioning the other elements into two sub-arrays, according to whether they are < or > than the pivot. Then sub-arrays are sorted recursively

$$T(n) = 2T(n/2) + O(n)$$

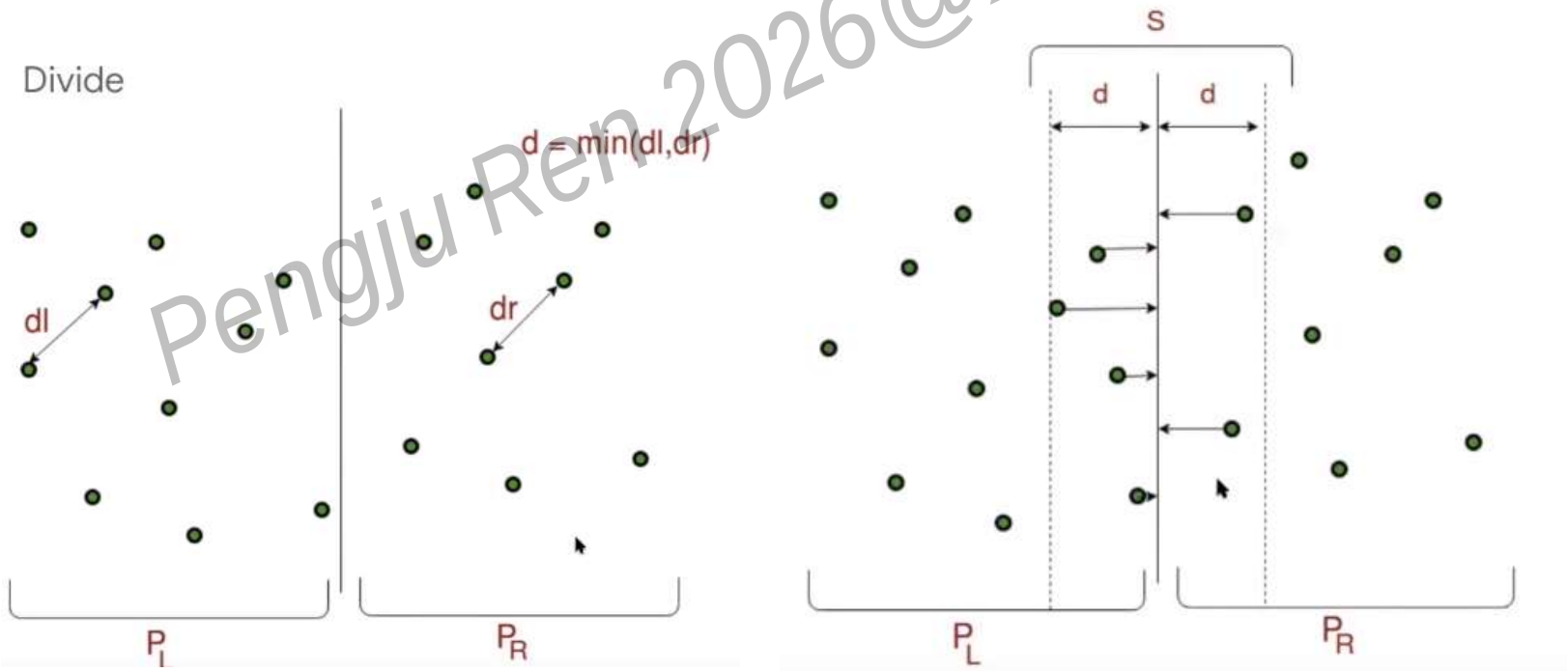
```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr)//2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

Example of D&C Algorithms

Closest Pair of Points : given an array of n points in the plane, and the problem is to find out the closest pair of points

$T(n) = 2T(n/2) + O(n\log n)$ and $T(n) = 2T(n/2) + O(n)$ (optimized)

Divide-and-Conquer Strategy:



Example of D&C Algorithms

Karatsuba Algorithm: **Given two large integers x and y with n digits, compute their product.** $\boldsymbol{T(n) = 3T(n/2) + O(n)}$

Divide-and-Conquer Strategy:

Split x and y into higher and lower parts:

$$\mathrm{x} = a \cdot 10^{\mathrm{m}} + \mathrm{b}, \mathrm{y} = \mathrm{c} \cdot 10^{\mathrm{m}} + \mathrm{d},$$

where m is approximately $\mathrm{n}/2$. Then

$$\mathrm{x} \cdot \mathrm{y} = ac \cdot 10^{2\mathrm{m}} + (a\mathrm{d} + \mathrm{bc}) \cdot 10^{\mathrm{m}} + \mathrm{bd}.$$

Direct computation requires *four* multiplications ($ac, a\mathrm{d}, \mathrm{bc}, \mathrm{bd}$).
Karatsuba only needs *three*（ac, bd, (a+b)(c+d)):

$$ad + bc = (a + b)(c + d) - ac - bd.$$

Example of D&C Algorithms

Toom-Cook Algorithm: **generalizes Karatsuba by splitting each one into $\mathbf{k}$ parts, reducing the number of multiplications to $\mathbf{2k-1}$**

$$\boldsymbol{T(n) = 5T(n/3) + O(n)}$$

Divide-and-Conquer Strategy:

$$A(x) = a_{k-1}x^{k-1} + \cdots + a_0 \quad B(x) = b_{k-1}x^{k-1} + \cdots + b_0$$

Taking Toom-3 for an example:

$$A(x) = a_2x^2 + a_1x^1 + a_0 \quad B(x) = b_2x^2 + b_1x^1 + b_0$$

And the result is : $C(x) = C_4x^4 + C_3x^3 + C_2x^2 + C_1x^1 + C_0$

This reduces the number of sub-multiplications required from the ***nine*** needed in a direct block decomposition to only ***five*** sub-multiplications

Example of D&C Algorithms

Strassen's Matrix Multiplication: **Given two $\mathbf{n \times n}$ matrices $\mathbf{A}$ and $\mathbf{B}$, compute their product $\mathbf{C = A \times B}$. The standard algorithm requires $\mathbf{O(n^3)}$ time.**

$$\boldsymbol{T(n) = 7T(n/2) + O(n^2)}$$

Divide-and-Conquer Strategy:

Partition the matrices into 2 × 2 blocks:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$M_3 = A_{11}(B_{12} - B_{22})$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12})B_{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Requires 7 multiplications and $O(n^2)$ additions/subtractions.

Master Theorem

The *Master Theorem* for solving decreasing functions applies to recurrences of the form

$$T(n) = aT(n-b) + f(n) \qquad \text{Let } f(n) = n^c$$

where $a > 0$, $b > 0$, and f is asymptotically positive.

IDEA: Compare a with 1 .

- **CASE 1: $a > 1 \Rightarrow T(n) = O(a^{n/b} * f(n))$.**
- **CASE 2: $a = 1 \Rightarrow T(n) = O(n * f(n))$.**
- **CASE 3: $a < 1 \Rightarrow T(n) = O(f(n))$**

Master Theorem

The *Master Theorem* for solving dividing functions applies to recurrences of the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \qquad \text{Let } f(n) = n^c$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

IDEA: Compare $n^{\log_b a}$ with n^c .

- **CASE 1:** $\log_b a > c$

  $f(n) = O(n^{\log_b a - \varepsilon})$, constant $\varepsilon > 0$ $\Rightarrow T(n) = \Theta(n^{\log_b a})$.

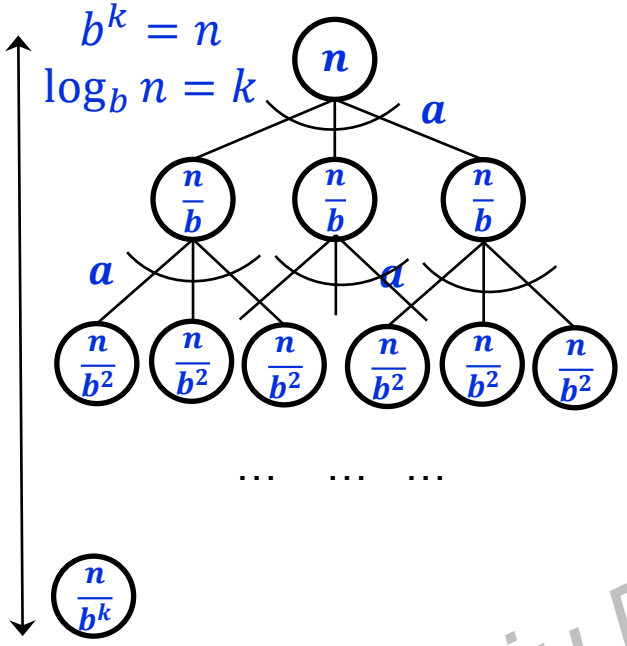
- **CASE 2:** $\log_b a \approx c$

  $f(n) = \Theta(n^{\log_b a} \lg^k n)$, constant $k \geq 0$ $\Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$.

- **CASE 3:** $\log_b a < c$

  $f(n) = \Omega(n^{\log_b a + \varepsilon})$, constant $\varepsilon > 0$ $\Rightarrow T(n) = \Theta(f(n))$.

Proven of Master Theorem



$\rightarrow f(n)$

$\rightarrow af(\frac{n}{b})$

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Let $f(n) = n^c$

$$T(n) = f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \cdots a^k f(1)$$

$$= \sum_{i=0}^{k} a^i f\left(\frac{n}{b^i}\right) = \sum_{i=0}^{k} a^i \left(\frac{n}{b^i}\right)^c = n^c \sum_{i=0}^{k} \left(\frac{a}{b^c}\right)^i$$

Let $\frac{a}{b^c} = q$

$$T(n) = n^c \frac{1 - q^{k+1}}{1 - q}$$

1) $q < 1 \Rightarrow \frac{a}{b^c} < 1 \Rightarrow a < b^c \Rightarrow c > \log_b a$ $\quad T(n) = n^c \frac{1}{1-q} = \alpha\, n^c \Rightarrow \boldsymbol{T(n) = O(n^c)}$

2) $q = 1 \Rightarrow \frac{a}{b^c} = 1 \Rightarrow a = b^c \Rightarrow c = \log_b a$ $\quad T(n) = n^c k = n^c \log_b n = n^c \frac{\log_2 n}{\log_2 b} \Rightarrow \beta n^c \log_2 n$

$\Rightarrow \boldsymbol{T(n) = O(n^c \log_2 n) = O(n^{\log_b a} \log_2 n)}$

3) $q > 1 \Rightarrow \frac{a}{b^c} > 1 \Rightarrow a > b^c \Rightarrow c < \log_b a$ $\quad T(n) = n^c q^k = n^c \left(\frac{a}{b^c}\right)^{\log_b n} = n^c \frac{a^{\log_b n}}{b^{c \log_b n}}$

$= n^c \frac{a^{\log_b n}}{(b^{\log_b n})^c} = n^c \frac{a^{\log_b n}}{n^c} = a^{\frac{\log_a n}{\log_a b}} = a^{\log_a n \log_b n} = n^{\log_b a}$ $\quad \Rightarrow \boldsymbol{T(n) = O(n^{\log_b a})}$

Master Theorem

- **Binary Search:**

$$T(n) = T\left(\frac{n}{2}\right) + O(1) \quad \Rightarrow \quad O(\log_2 n)$$

- **Merge/Quick Sort/Closest Pair of Points:**

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \quad \Rightarrow \quad O(n \log_2 n)$$

- **Karatsuba Algorithm:** $T(n) = 3T(n/2) + O(n) \Rightarrow O(n^{\log_2 3}) = O(n^{1.585})$

- **Toom-Cook Algorithm:** $T(n) = 5T(n/3) + O(n) \Rightarrow O(n^{\log_3 5}) = O(n^{1.465})$

- **Recursive Matrix Multiplication**

$$T(n) = 8T\left(\frac{n}{2}\right) + n^2 \quad \Rightarrow \quad O(n^{\log_2 8}) = O(n^3)$$

- **Strassen Matrix Multiplication**

$$T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \quad \Rightarrow \quad O(n^{\log_2 7}) = O(n^{2.81})$$

Homework 4

Given three polynomials:

$$A(x) = \mathrm{a}_0 + \mathrm{a}_1 x + \cdots + \mathrm{a}_{\mathrm{n-1}} x^{n-1}$$
$$B(x) = b_0 + b_1 x + \cdots + b_{\mathrm{n}-1} x^{n-1}$$
$$C(x) = c_0 + c_1 x + \cdots + c_{\mathrm{n}-1} x^{n-1}$$

We want to compute their product

$$\mathrm{D}(x) = \mathrm{A}(x)\mathrm{B}(x)\mathrm{C}(x)$$

try to accelerate the process using divide and conquer algorithm

Submission requirements:

1: Description of the implementation approach
2: Performance test results and analysis
3: Difficulties encountered and solutions

Homework naming: HW4-Class X-stu ID-name, e.g. HW4-Class01-12345678-张三

Submission email: chengyuma@stu.xjtu.edu.cn

Submission time: 3.29 24:00

Dynamic Programming

DP : Longest Increasing Subsequence, LIS

Given an integer array, find the length of *the longest strictly increasing subsequence*. The subsequence does not need to be contiguous, but the order must be preserved.

E.g. nums = [10,9,2,5,3,4,7] → LIS is [2,3,4,7], length 4.

```
def length_of_lis(nums):
    n = len(nums)
    dp = [1] * n
    for i in range(n):
        for j in range(i):
            if nums[j] < nums[i]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp) if n else 0
```

$$dp[i] = 1 + \max_{j<i; nums[j]<nums[i]} \{dp[j]\}$$

Basic Steps of Dynamic Programming

Step 1: Define the State (State Definition)

Step 2: Determine the *State Transition Equation*

Step 3: Set the *Initial Conditions* and Boundaries

Step 4: Determine the Computation Order

Step 5: Extract the Final Answer

DP : 0-1 Knapsack

Given *n* items, each with *weight w[i]* and *value v[i]*, and a knapsack with *capacity W*, each item can be selected at most once. Find the maximum total value that can be obtained without exceeding the capacity.

E.g. Weights [2,3,6,5], Values [6,3,5,4], Capacity 9 → Maximum value is 11

```
def knapsack(weights, values, W):
    n = len(weights)
    dp = [0] * (W + 1)
    for i in range(n):
        w, v = weights[i], values[i]
        for j in range(W, w - 1, -1):
            dp[j] = max(dp[j], dp[j - w] + v)
    return dp[W]
```

$$dp[i][j] = \max\left(dp[i-1][j], dp[i-1][j-w[i]] + v[i]\right) (j \geq w[i])$$

DP：Edit Distance

Given two strings *s1* and *s2*, find the *minimum number of operations* required to convert s1 to s2. Allowed operations: *insert* a character, *delete* a character, *replace* a character.

E.g. S1 'abcf', S2 'bcfe', (abcf-bcf-bcfe)

```
def edit_distance(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0]*(n+1) for _ in range(m+1)]
    for i in range(m+1):
        dp[i][0] = i
    for j in range(n+1):
        dp[0][j] = j
    for i in range(1, m+1):
        for j in range(1, n+1):
            if s1[i-1] == s2[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])
    return dp[m][n]
```


$$dp[i][j] = \begin{cases} j & if\ i = 0 \\ i & if\ j = 0 \\ \min(dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]+cost) & otherwise \end{cases}$$

DP : Longest Common Subsequence, LCS

Given two strings *s1* and *s2*, find the length of the *longest common subsequence* (not necessarily contiguous).

E.g. S1 'abcde', S2 'ace', → LCS is 'ace'

```
def lcs(s1, s2):
    m, n = len(s1), len(s2)
    dp = [[0]*(n+1) for _ in range(m+1)]
    for i in range(1, m+1):
        for j in range(1, n+1):
            if s1[i-1] == s2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    return dp[m][n]
```


$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1 & if\ s1[i-1] = s2[j-1] \\ \max(dp[i-1][j],\ dp[i][j-1]) & otherwise \end{cases}$$

DP : Matrix Chain Multiplication

Given a sequence of matrix dimensions *p[n]*, where the i-th matrix has dimensions *p[i-1] × p[i]*, find the optimal *parenthesization* that minimizes the number of scalar multiplications.

E.g. Dimensions p[n]=[10, 20, 30, 40] (three matrices: 10 × 20, 20 × 30, 30 × 40)
→ Optimal is 18,000 (10x20x30+10x30x40) multiplications.

```
def matrix_chain_order(p):
    n = len(p) - 1
    dp = [[0]*(n+1) for _ in range(n+1)]
    for length in range(2, n+1):
        for i in range(1, n - length + 2):
            j = i + length - 1
            dp[i][j] = float('inf')
            for k in range(i, j):
                cost = dp[i][k] + dp[k+1][j] + p[i-1]*p[k]*p[j]
                if cost < dp[i][j]:
                    dp[i][j] = cost
    return dp[1][n]
```


$$dp[i][j] = \min_{i \le k < j} \{dp[i][k] + dp[k+1][j] + p[i-1] \cdot p[k] \cdot p[j]\}$$

DP : Stone Merge

Given *n* piles of stones arranged in a line, each merge combines two adjacent piles, with the cost equal to the sum of their *weights*. Find the minimum total cost to merge all piles into one.

Example: Stones[4,2,3,6] → Optimal cost is 29, (4+(2+3))+6 cost is 5+9+15=29.
→ ((4+2)+3)+6 cost is 6+9+15=30, (4+2)+(3+6) cost is also 6+9+15=30

```
def stone_merge(stones):
    n = len(stones)
    prefix = [0]*(n+1)
    for i in range(n):
        prefix[i+1] = prefix[i] + stones[i]
    def sum_range(i, j):
        return prefix[j+1] - prefix[i]
    dp = [[0]*n for _ in range(n)]
    for length in range(2, n+1):
        for i in range(n - length + 1):
            j = i + length - 1
            dp[i][j] = float('inf')
            for k in range(i, j):
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + sum_range(i,j))
    return dp[0][n-1]
```


$$dp[i][j] = \min_{i \le k < j} \{dp[i][k] + dp[k+1][j] + S(i,j)\}$$

DP：Box Stacking

Given various boxes (*length, width, height*), unlimited supply of each and everybox can be rotated arbitrarily (any permutation of dimensions). Boxes can be stacked only if the base dimensions (length and width) of the *lower box are strictly greater than those of the upper box*. Find the maximum achievable height.

E.g. Given 3 boxes (length, width, height)：A: (4, 6, 7)、B: (1, 2, 3)、C: (3, 5, 4)
The maximum total height is max(dp) = 20

DP : Box Stacking

```
def max_stack_height(boxes):
    # boxes: list of (l, w, h)
    rotations = []
    for l, w, h in boxes:
        rotations.append((l, w, h))
        rotations.append((l, h, w))
        rotations.append((w, l, h))
        rotations.append((w, h, l))
        rotations.append((h, l, w))
        rotations.append((h, w, l))
    # 使长≥宽，过滤
    boxes = [(max(l,w), min(l,w), h) for l,w,h in rotations]
    # 按底面积降序（或先长后宽）
    boxes.sort(key=lambda x: x[0]*x[1], reverse=True)
    n = len(boxes)
    dp = [0] * n
    for i in range(n):
        l, w, h = boxes[i]
        dp[i] = h
        for j in range(i):
            lj, wj, hj = boxes[j]
            if lj > l and wj > w:
                dp[i] = h+max(0, dp[j])
    return max(dp)
```


$$dp[i] = h_i + \max(0, \max_{\substack{j<i \\ l_j>l_i, w_j>w_i}} dp[j])$$

Top-Down and Bottom-Up Approaches

Dynamic Programming can be implemented in two main ways:

- **Top-Down (Memorization):** Start from the original problem and recursively solve subproblems, storing computed results in a memo (e.g., array or dictionary).
  - Advantages: Intuitive, follows the natural recursive formulation.
  - Disadvantages: Recursion overhead, risk of stack overflow for deep recursion.
- **Bottom-Up (Tabulation):** Start from the smallest subproblems and iteratively fill the DP table according to the computation order.
  - Advantages: High efficiency, no recursion stack risk.
  - Disadvantages: Requires careful determination of computation order.

Notice: Both approaches are essentially equivalent; the choice depends on personal preference and the specific problem.

Diff of DC and DP

Understanding their differences helps deepen our understanding of the Divide and Conquer approach:

Feature	Divide and Conquer	Dynamic Programming
Subproblem Nature	Subproblems are **independent**, with no overlap.	Subproblems **overlap**; ***the solution to one subproblem may be used multiple times.***
Solution Approach	Typically, a **top-down** recursive process.	Typically, both ***top-down*** and ***bottom-up*** use iterative process, or recursion with memorization.
Typical Applications	*Merge Sort, Quick Sort, Closest Pair of Points*	*Knapsack Problem, LSI,LCS, Edit Distance, Matrix Chain Multiplication, Stone Merge, Box-Stacking*
Core Overhead	Primarily in the combination step	Primarily in ***state transition and storage*** of all subproblem solutions

Summary

- **Divide and Conquer :** It simplifies complex problems through recursive *decomposition* and then reconstructs the solution to the original problem through *combination*
- **Dynamic Programming (in follow-up courses):** a method used to solve problems that exhibit **overlapping subproblems** and **optimal substructure**. The core idea is to decompose the original problem into a series of interdependent subproblems, and then solve them in a **bottom-up** or **top-down** manner while storing the results (typically in arrays or hash tables) to avoid redundant calculations. *This often reduces the time complexity from exponential to polynomial*.

Homework 5

Xiao Ming is participating in the “11.11” shopping festival. He has a limited budget and wants to buy some items to maximize his total happiness. However, he has a special requirement: the number of **Category A** items he buys must not exceed the number of **Category B** items (for example, Category A could be electronics and Category B daily necessities; he wants at least as many daily necessities as electronics).

There are n types of items, each with a price p_i in yuan), a happiness value h_i , and a category t_i (A and B) . Xiao Ming has a total budget of M yuan. He can buy at most one of each item.

Help him choose items so that the total cost does not exceed M, *the number of Category A items ≤ the number of Category B items*, and the total happiness is maximized. If there are multiple optimal solutions, output any one. If no solution exists, output 0.

Homework 5

Test case: 1) M=10 and 2) M=15

	Price	Happiness value	Category
Item1	3	5	A
Item2	4	7	B
Item3	2	4	A
Item4	5	8	B
Item5	3	6	B

Submission requirements:

1: Description of the implementation approach
2: Performance test results and analysis
3: Difficulties encountered and solutions

Homework naming: HW5-Class X-stu ID-name, e.g. HW5-Class01-12345678-张三

Submission email: chengyuma@stu.xjtu.edu.cn

Submission time: 4.09 24:00