HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

FUNDAMENTALS OF OPTIMIZATION

ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

FUNDAMENTALS OF OPTIMIZATION

Week 3: Divide and Conquer & Dynamic Programming

ONE LOVE. ONE FUTURE.

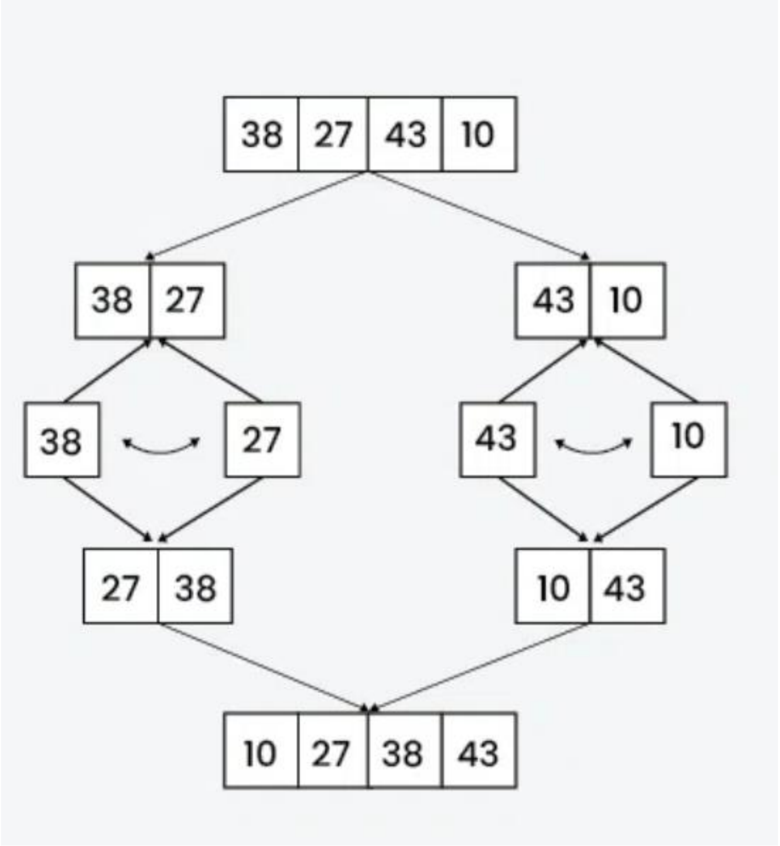
Outline

1. **Divide and Conquer**
   - **Introduction to Divide and Conquer (D&C)**
   - Example: Multiplication of two big numbers
   - Example: Allocate pages to students
   - Decrease and Conquer
2. Dynamic Programing
   - Introduction to Dynamic Programming
   - Example: Fibonacci numbers
   - Example: Money change
   - Example: The longest ascending subsequence
   - Example: Longest common subsequence

Divide and Conquer Algorithm

Divide and Conquer algorithm is a problem-solving strategy that involves.

- **Divide** : Break the given problem into smaller non-overlapping problems.
- **Conquer** : Solve Smaller Problems
- **Combine** : Use the Solutions of Smaller Problems to find the overall result.



Divide and Conquer Algorithm

- Complexity analysis
  - $T(n)$: running time of input size n

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq n_c \\ aT(n/b) + D(n) + C(n) & \text{if } n \geq n_c, \end{cases}$$

- Consider: $T(n) = aT(n/b) + n^k$ với a, b, c, k are positive constants and $a \geq 1, b \geq 2$:

→ $T(n) = \begin{cases} O(n^{\log_b a}), \text{ nếu } a > b^k \\ O(n^k \log n), \text{ nếu } a = b^k \\ O(n^k), \text{ nếu } a < b^k \end{cases}$

Example: Karatsuba Algorithm for Multiplication of two big numbers

- **Problem Statement:** Given two very large positive integers **a** and **b** (with up to 10,000 digits), calculate and print their product.
- **Input Format**
  - **Line 1**: Contains the integer **a**.
  - **Line 2**: Contains the integer **b**.
- **Output Format:** Print the result of **a × b**.
- **Example**
  - **Input**

    123

    654
  - **Output**

    80442

Example: Karatsuba Algorithm for Multiplication of two big numbers

- Multiplication of 2 big numbers *A* and *B* (containing *n* digits)
- $A = A_1 \text{x} 10^{n/2} + A_2$
- $B = B_1 \text{x} 10^{n/2} + B_2$
- $A \text{ x } B = (A_1 \text{x} 10^{n/2} + A_2) \text{ x } (B_1 \text{x} 10^{n/2} + B_2) = A_1 \text{ x } B_1 \text{x} 10^{n} + (A_1 \text{ x } B_2 + A_2 \text{ x } B_1) \text{x} 10^{n/2} + A_2 \text{ x } B_2$
- $A_1 \text{ x } B_2 + A_2 \text{ x } B_1 = (A_1 + A_2) \text{ x } (B_1 + B_2) - A_1 \text{ x } B_1 - A_2 \text{ x } B_2$
- $A \text{ x } B = A_1 \text{ x } B_1 \text{x} 10^{n} + ((A_1 + A_2) \text{ x } (B_1 + B_2) - A_1 \text{ x } B_1 - A_2 \text{ x } B_2) \text{x} 10^{n/2} + A_2 \text{ x } B_2$
- Complexity:
  - $T(n) = 3T(n/2) + O(n)$
  - $T(n) = O(n^{\log_2 3})$

Example: Allocate pages to students

Given an array arr[] and an integer k, where arr[i] denotes the number of pages of a book and k denotes total number of students. All the books need to be allocated to k students in contiguous manner, with each student getting at least one book. The task is to minimize the maximum number of pages allocated to a student. If it is not possible to allocate books to all students, return -1.

- **Input**
  - Line 1: An array arr, elements are separated by a space
  - Line 2: The number of student k
- **Output:** One integer that is maximum number of pages allocated to a student
- **Example**
  - **Input**

    12 34 67 90

    2
  - **Output**

    113

Example: Allocate pages to students

- The idea is to **iterate** over **all possible** page limits, or maximum pages that can be allocated to a student.
- The **minimum** possible page limit is the **highest** page count among all books, as the book with the most pages must be assigned to some student.
- The **maximum** possible page limit is the **sum** of pages of all books, It is in the case when all books are given to a single student.
- To find the number of students that will be allocated books for a page limit, we start assigning books to the first student until the page limit is reached, then we move to the next student and so on. As soon as we find the first page limit with which we can allocate books to all k students, we will return it.

Decrease and Conquer

- Given a binary sequence *X* of length *n* which can be divided into 2 parts: the prefix contains only 0 and the suffix contains only 1.
    - Example: 0000000011111111111111111
- Goal: Find the index of the first 1-bit (from left to right)

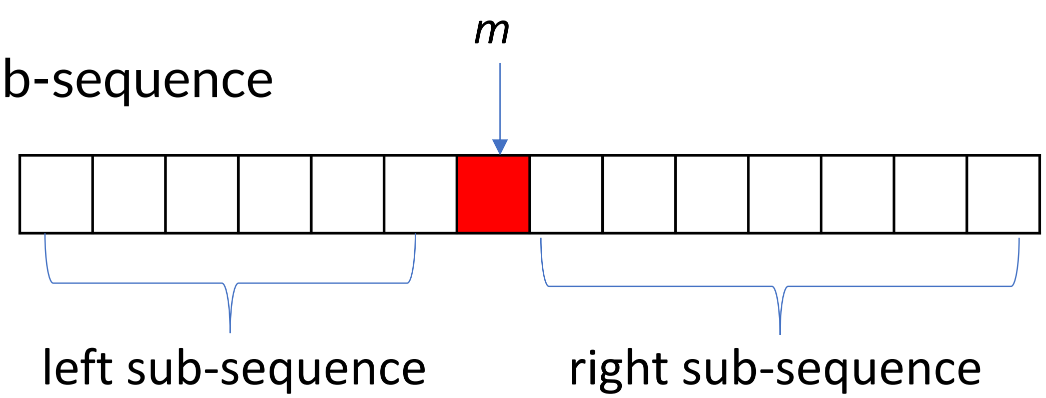
Decrease and Conquer

- Given a binary sequence *X* of length *n* which can be divided into 2 parts: the prefix contains only 0 and the suffix contains only 1.
  - Example: 0000000011111111111111111
- Goal: Find the index of the first 1-bit (from left to right)
  - Example

0	0	0	0	0	0	0	0	0	1	1	1	1	1

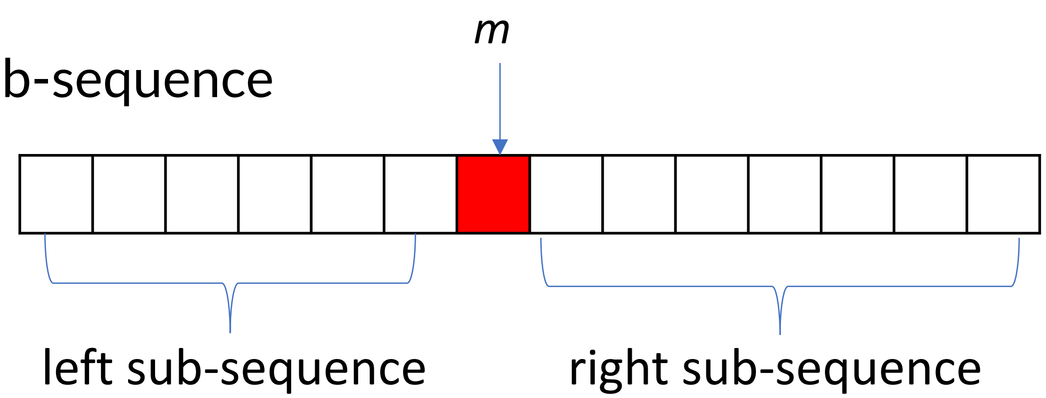
Decrease and Conquer

- Given a binary sequence *X* of length *n* which can be divided into 2 parts: the prefix contains only 0 and the suffix contains only 1.
  - Example: 0000000011111111111111
- Goal: Find the index of the first 1-bit (from left to right)
- Decrease and conquer
  - Let *m* the middle position of *X*
  - Consider the bit *X*[*m*] in the middle of *X*
    - If *X*[*m*] = 0 then find in the result in the right sub-sequence
    - If *X*[*m*] = 1
      - If *X*[*m*-1] = 0 then return *m*
      - Otherwise, find the result in the left sub-sequence



Decrease and Conquer

- Given a binary sequence *X* of length *n* which can be divided into 2 parts: the prefix contains only 0 and the suffix contains only 1.
  - Example: 000000001111111111111111
- Goal: Find the index of the first 1-bit (from left to right)
- Decrease and conquer
  - Let *m* the middle position of *X*
  - Consider the bit *X*[*m*] in the middle of *X*
    - If *X*[*m*] = 0 then find in the result in the right sub-sequence
    - If *X*[*m*] = 1
      - If *X*[*m*-1] = 0 then return *m*
      - Otherwise, find the result in the left sub-sequence
  - Time complexity: O(log*n*)

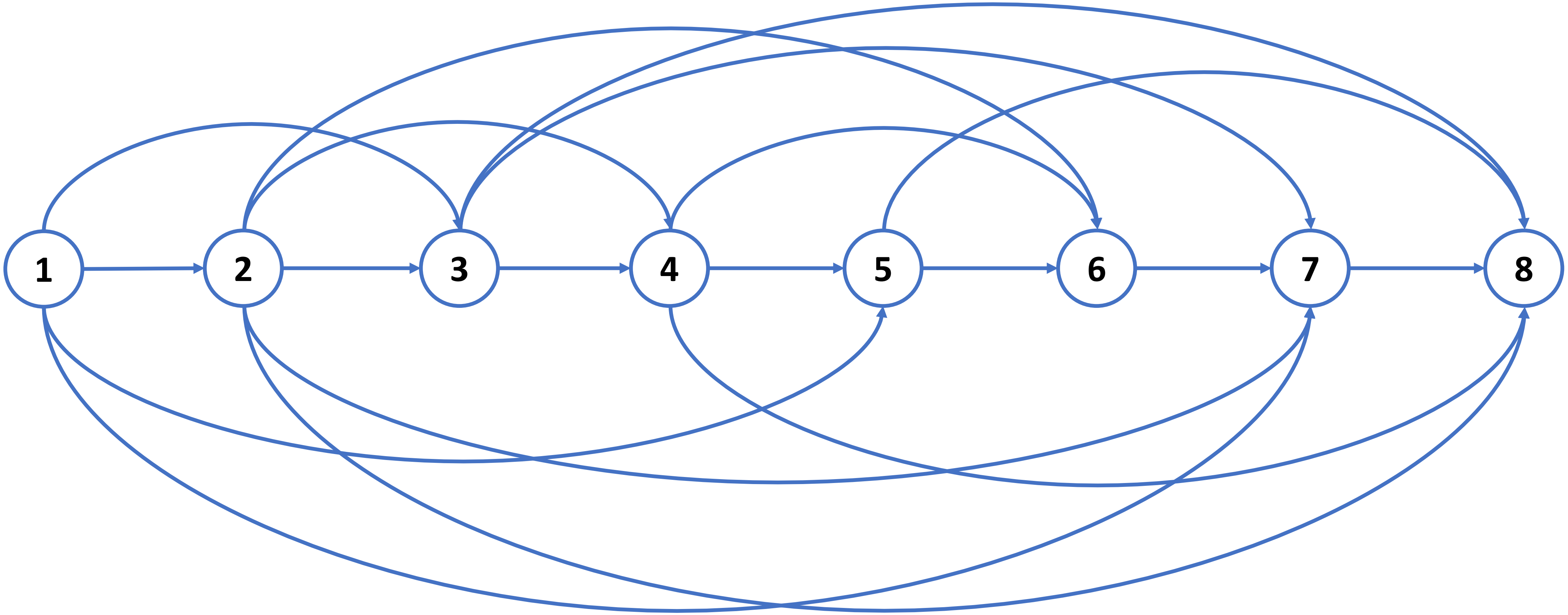


Outline

1. Divide and Conquer
    - Introduction to Divide and Conquer (D&C)
    - Example: Multiplication of two big numbers
    - Example: Allocate pages to students
    - Decrease and Conquer
2. **Dynamic Programing**
    - **Introduction to Dynamic Programming**
    - Example: Fibonacci numbers
    - Example: Money change
    - Example: The longest ascending subsequence
    - Example: Longest common subsequence

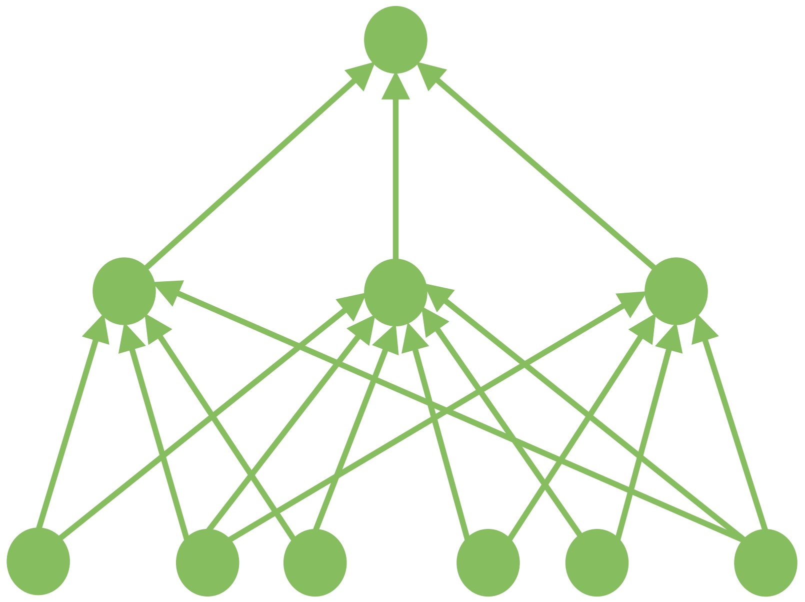
What is dynamic programming ?

- How many ways to travel from point 1 to point 8?



What is dynamic programming

- **Dynamic Programming** is a problem-solving strategy that involves.
  - **Divide** : Break the given problem into smaller overlapping problems.
  - **Solve** : Solve Smaller Problems
  - **Combine** : Use the Solutions of Smaller Problems to find the overall result.
- **Principle**: Each subproblem is attacked at most once.



Top-Down implementation with memorized recursion

```
# Dictionary to store computed results (Memoization)
Memory = {}

def DP(P):
    # Base case check
    if is_base_case(P):
        return base_case_value(P)

    # Check if the result is already computed
    if P in Memory:
        return Memory[P]

    # Initialize result
    result = some_value  # Replace 'some_value' with an appropriate initial value

    # Solve subproblems and combine results
    for Q in subproblems(P):
        result = Combine(result, DP(Q))

    # Store result in Memory and return
    Memory[P] = result
    return result
```

Example: Fibonacci numbers

- *The first two numbers of Fibonacci sequence are 1 and 1. All other numbers in the sequence are calculated as the sum of the two numbers immediately preceding them in the sequence.*
- **Requirement:** Calculate the n^{th} Fibonacci number
- Try to solve the problem by using dynamic programming

1. Find recursive formular:

$$\mathrm{Fib}(1) = 1$$
$$\mathrm{Fib}(2) = 1$$
$$\mathrm{Fib}(n) = \mathrm{Fib}(n-2) + \mathrm{Fib}(n-1)$$

Example: Fibonacci numbers

```python
# Dictionary to store computed Fibonacci values (Memoization)
mem = {}

def Fib(n):
    if n <= 2:
        return 1  # Base case: Fib(1) = 1, Fib(2) = 1

    if n in mem:  # Check if the value is already computed
        return mem[n]

    # Compute and store the Fibonacci number
    res = Fib(n - 1) + Fib(n - 2)
    mem[n] = res

    return res

# Example usage
print(Fib(n))
```

What is the complexity ?

Example: Fibonacci numbers

- There are *n* possible inputs for the recursive function: *1*, *2*, ..., *n*
- For each input:
  - Either the results are calculated and stored
  - Or retrieve it from memory if it has been calculated before
- Each input will be calculated at most once
- The computation time is *O(n x f)*, where *f* is the computation time of the function for one input, assuming that the previously computed result will be retrieved directly from memory, in only *O(1)*
- As we only spend a constant amount of calculation on one input of the function, so *f = O(1)*
- Total computation time is *O(n)*

Example: Money change

- Given a set of coins with denominations $D_1, D_2, \ldots, D_n$ and an amount of money X. Find the minimum number of coins to exchange for X.
- Like the knapsack problem?
- Is there a greedy algorithm to solve this problem?
- The knapsack problem learned in Discrete Mathematics is solved using the branch and bound algorithm. The greedy algorithm is not certain of providing the optimal solution, and in many cases cannot even provide the solution...
- Try using the Dynamic Programming method!
- Finally, make comments on different approaches to solving this problem

Example: Money change - dynamic programming formular

First step: build the dynamic programming

- Let *MinCoin(i, x)* be the minimum amount of money needed to exchange denomination *x* if only the denominations D_1, D_2, ..., D_i are allowed to be used.
- Base (

$$\mathrm{MinCoin}(i, x) = \infty \text{ nếu } x < 0$$
$$\mathrm{MinCoin}(i, 0) = 0$$
$$\mathrm{MinCoin}(0, x) = \infty$$

- Recurrence $\mathrm{MinCoin}(i, x) = \min \begin{cases} 1 + \mathrm{MinCoin}(i, x - D_i) \\ \mathrm{MinCoin}(i - 1, x) \end{cases}$

Money change: Implementation

```python
# Constants
INF = int(1e9)
N = 20
XMAX = int(1e5) + 5

# Coin denominations array
D = [0] * N  # Placeholder, should be initialized with actual coin values

# Memoization table (Initialized with -1)
mem = [[-1] * XMAX for _ in range(N)]

def MinCoin(i, x):
    if x < 0 or i == 0:
        return INF  # Impossible case
    if x == 0:
        return 0  # Base case: no coins needed for amount 0

    if mem[i][x] != -1:
        return mem[i][x]  # Return already computed value

    res = INF
    res = min(res, 1 + MinCoin(i, x - D[i]))  # Take the current coin
    res = min(res, MinCoin(i - 1, x))  # Skip the current coin

    mem[i][x] = res  # Store computed value
    return res
```

Money change: Complexity

- Total calculation time is $O(nx)$
- How to determine which coins are used in the optimal solution?
- Let's trace the recursive process back

Money change: Tracing using recursion

```python
def Trace(i, x):
    if x <= 0 or i == 0:
        return

    if mem[i][x] == 1 + mem[i][x - D[i]]:
        print(D[i], end=' ')  # Print the selected coin
        Trace(i, x - D[i])  # Recur with the remaining amount
    else:
        Trace(i - 1, x)  # Move to the next coin type
```

- Call Trace(n, x);
- The complexity of Trace function? *O(max(n, x))*

Example: Longest increasing subsequence (LIS)

- Given a sequence of n integers $A[1], A[2], ..., A[n]$. Find the length of the longest increasing subsequence.
- Definition: If delete 0 or some elements of the sequence A, a subsequence of A will be obtained.

Example: A = [2, 0, 6, 1, 2, 9]

- [2, 6, 9] is a subsequence of A
- [2, 2] is a subsequence of A
- [2, 0, 6, 1, 2, 9] is a subsequence of A
- [] is a subsequence of A
- [9, 0] is not a subsequence of A
- [7] is not a subsequence of A

Example: Longest increasing subsequence (LIS)

- An ascending subsequence of A is a subsequence of A such that the elements are strictly increasing from left to right
- [2, 6, 9] and [1, 2, 9] are two increasing subsequences of A = [2, 0, 6, 1, 2, 9]
- How to calculate length of longest increasing subsequence?
- There are 2^n subsequences, the simplest method is to traverse all of these sequences
  - The complexity of this algorithm is $O(n \times 2^n)$, it thus can only run quickly (e.g., within 1 second) to produce results with $n \leq 23$.
- Try the Dynamic Programming method!

Example: Longest increasing subsequence (LIS)

- Let LIS(i) be the length of the longest increasing subsequence of the array *A[1]*, *A[2]*, ..., *A[i]* that ends at the i^{th} element.
- Base case: LIS(1) = 1
- Recurrence relation:

$$\mathrm{LIS}(i) = \max(1, \max_{j \text{ s.t. } A[j]<A[i]}\{1 + \mathrm{LIS}(j)\})$$

Example: Longest increasing subsequence (LIS)

```python
# Constants
N = int(1e4) + 5

# Arrays for input sequence and memoization
a = [0] * N  # Placeholder, should be initialized with actual values
mem = [-1] * N  # Memoization table, initialized with -1

def LIS(i):
    if mem[i] != -1:
        return mem[i]  # Return memoized result

    res = 1  # Minimum LIS length is 1 (element itself)

    for j in range(1, i):  # Loop through previous elements
        if a[j] < a[i]:  # Check for increasing subsequence
            res = max(res, 1 + LIS(j))

    mem[i] = res  # Store result in memo table
    return res
```

Example: Longest increasing subsequence (LIS)

- The length of the longest increasing subsequence is the largest value among the LIS(i) values.

```python
ans = 0
pos = 0

for i in range(1, n + 1):  # Loop from 1 to n (1-based index)
    if ans < mem[i]:
        ans = mem[i]
        pos = i

print(ans)  # Print the maximum value
```

Example: The longest increasing subsequence: Complexity

- There are n possibilities for input
- Each input is calculated in $O(n)$.
- Total calculation time is $O(n^2)$
- Can be run (within 1 second) up to $n \leq 10000$, much better than the brute force method
- Applying the segment tree structure to the above method will improve the complexity to $O(nlogn)$.
- Another improved method is to use a new dynamic programming formular that incorporates binary search which also gives $O(nlogn)$
- Trace?

Example: Longest increasing subsequence (LIS): Tracing using recursion

```
def Trace(i):
    for j in range(1, i):  # Loop from 1 to i-1 (1-based index)
        if a[j] < a[i] and mem[i] == 1 + mem[j]:
            Trace(j)  # Recursively find the previous element
            break  # Stop after finding the correct predecessor

    print(a[i], end=' ')  # Print the current element
```

- Call Trace(pos);
- The complexity of Trace function? $O(n^2)$
- Can be improved: $O(n)$

Example: Longest increasing subsequence (LIS): Tracing using recursion (improved)

```
def Trace(i):
    for j in range(i - 1, 0, -1):  # Loop from i-1 down to 1 (inclusive)
        if a[j] < a[i] and mem[i] == 1 + mem[j]:
            Trace(j)  # Recursively find the previous element
            break  # Stop after finding the correct predecessor

    print(a[i], end=' ')  # Print the current element
```

- Call Trace(pos);
- The complexity of Trace function? *O(n)*

Example: Longest common subsequence (Dãy con chung dài nhất)

- Given two strings (or two integer arrays) of *n and m* elements *X[1]*, *X[2]*, ..., *X[n]* và *Y[1]*, *Y[2]*,..., *Y[m]*. Find the length of the longest common subsequence of the two strings.
- Example: X = "a<u>bcb</u>", Y = "<u>b</u>d<u>c</u>a<u>b</u>"
- The longest common subsequence of X and Y is là "bcb" which has the length 3.

Example: Longest common subsequence: Dynamic programming formula

- Let LCS(i, j) be the length of the longest common subsequence of the sequences *X[1], X[2], ..., X[i]* and *Y[1], Y[2], ..., Y[j]*.
- Basic step:

  L(0, j) = L(i, 0) = 0

- Inductive step:

$$\mathrm{LCS}(i,j) = \max \begin{cases} \mathrm{LCS}(i, j-1) \\ \mathrm{LCS}(i-1, j) \\ 1 + \mathrm{LCS}(i-1, j-1) & \text{nếu } X[i] = Y[j] \end{cases}$$

Example: Longest common subsequence: Implement

```python
# Constants
N = int(1e4) + 5

# Strings for comparison
X = ""
Y = ""

# Memoization table (initialized to -1)
mem = [[-1] * N for _ in range(N)]

def LCS(i, j):
    if i == 0 or j == 0:
        return 0  # Base case: LCS of an empty string is 0

    if mem[i][j] != -1:
        return mem[i][j]  # Return memoized result

    res = 0
    res = max(res, LCS(i - 1, j))  # Exclude X[i-1]
    res = max(res, LCS(i, j - 1))  # Exclude Y[j-1]

    if X[i - 1] == Y[j - 1]:  # If characters match
        res = max(res, 1 + LCS(i - 1, j - 1))

    mem[i][j] = res  # Store result in memoization table
    return res
```

Example: Longest common subsequence: Example

iMem	j	0	1	2	3	4	5
i		Y[j]	**b**	**d**	**c**	**a**	**b**
0	X[i]	0	0	0	0	0	0
1	**a**	0 $\searrow$	0	0	0	1	1
2	**b**	0	1 $\rightarrow$	1 $\searrow$	1	1	2
3	**c**	0	1	1	2 $\rightarrow$	2 $\searrow$	2
4	**b**	0	1	1	2	2	3

$$\mathrm{LCS}(i,j) = \max \begin{cases} \mathrm{LCS}(i, j-1) \\ \mathrm{LCS}(i-1, j) \\ 1 + \mathrm{LCS}(i-1, j-1) & \text{nếu } X[i] = Y[j] \end{cases}$$

Example: Longest common subsequence: Complexity

- There are *n x m* possibilities for input
- Each input is calculated in O(1).
- Total calculation time is *O(n x m)*
- How to know exactly which elements belong to the longest common subsequence?

Example: Longest common subsequence: Tracing by using recursion

```python
def Trace(i, j):
    if i == 0 or j == 0:
        return

    if X[i - 1] == Y[j - 1] and mem[i][j] == 1 + mem[i - 1][j - 1]:
        Trace(i - 1, j - 1)
        print(X[i - 1], end='')   # Print matching character
        return

    if mem[i][j] == mem[i - 1][j]:   # Move up if value comes from top
        Trace(i - 1, j)
        return

    if mem[i][j] == mem[i][j - 1]:   # Move left if value comes from left
        Trace(i, j - 1)
        return
```

Example: Longest common subsequence: Tracing by using recursion

- The complexity of Trace function? $O(n + m)$

HUST

hust.edu.vn fb.com/dhbkhn

THANK YOU !