



HUST

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

ONE LOVE. ONE FUTURE.



APPLIED ALGORITHMS



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

APPLIED ALGORITHMS

DYNAMIC PROGRAMMING

ONE LOVE. ONE FUTURE.

CONTENTS

- Introduction to dynamic programming
- Examples
- Dynamic Programming with Bitmasking (TSP)

History of dynamic programming

- R.E.Bellman (1920-1984)



RICHARD BELLMAN ON THE BIRTH OF DYNAMIC PROGRAMMING

STUART DREYFUS

University of California, Berkeley, IEOR, Berkeley, California 94720, dreyfus@ieor.berkeley.edu

What follows concerns events from the summer of 1949, when Richard Bellman first became interested in multistage decision problems, until 1955. Although Bellman died on March 19, 1984, the story will be told in his own words since he left behind an entertaining and informative autobiography, *Eye of the Hurricane* (World Scientific Publishing Company, Singapore, 1984), whose publisher has generously approved extensive excerpting.

During the summer of 1949 Bellman, a tenured associate professor of mathematics at Stanford University with a developing interest in analytic number theory, was consulting for the second summer at the RAND Corporation in Santa Monica. He had received his Ph.D. from Princeton in 1946 at the age of 25, despite various war-related activities during World War II—including being assigned by the Army to the Manhattan Project in Los Alamos. He had already exhibited outstanding ability both in pure mathematics and in solving applied problems arising from the physical world. Assured of a successful conventional academic career, Bellman, during the period under consideration, cast his lot instead with the kind of applied mathematics later to be known as operations research. In those days applied practitioners were regarded as distinctly second-class citizens of the mathematical fraternity. Always one to enjoy controversy, when invited to speak at various university mathematics department seminars, Bellman delighted in justifying his choice of applied over pure mathematics as being motivated by the real world's greater challenges and mathematical demands.

what RAND was interested in. He suggested that I work on multistage decision processes. I started following that suggestion" (p. 157).

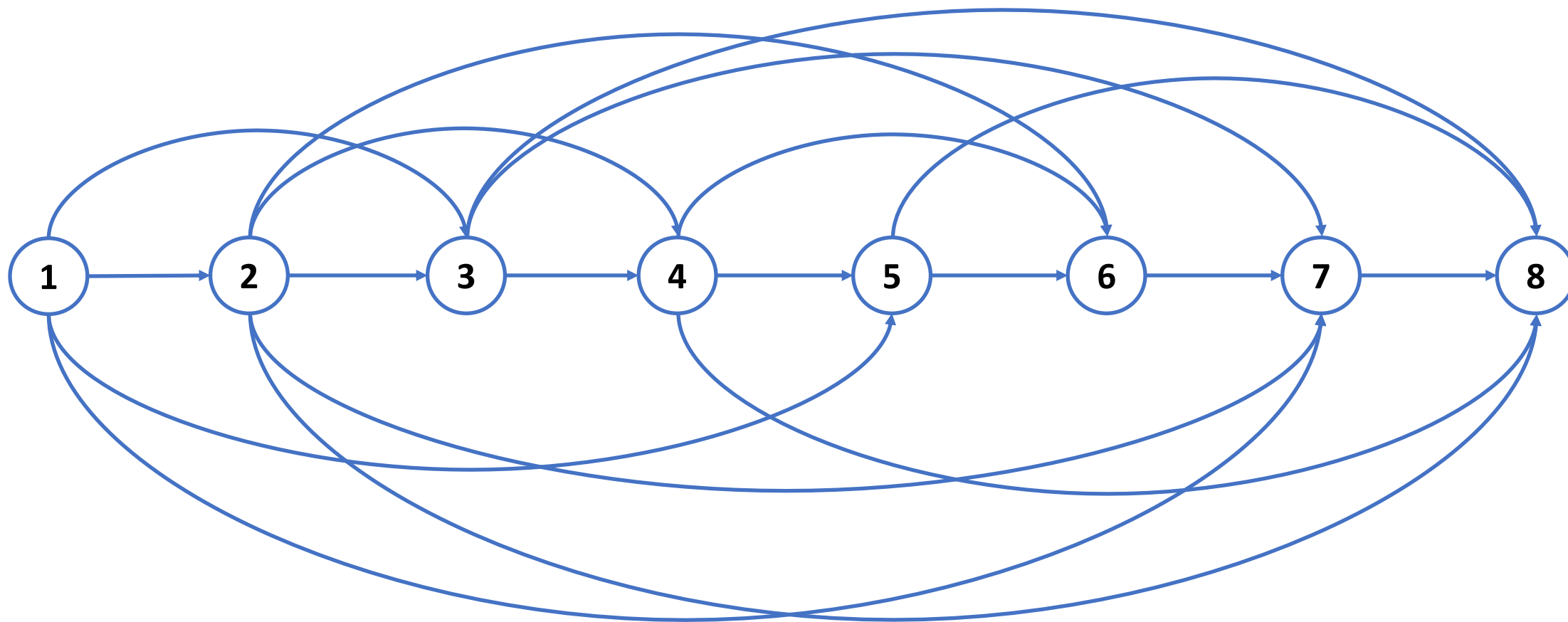
CHOICE OF THE NAME DYNAMIC PROGRAMMING

"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes.

"An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an

What is dynamic programming ?

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



What is dynamic programming ?

- Dynamic programming
 - Divide the original problem into ***overlapping*** subproblems
 - Recurrence relations
 - Solution to a subproblem is specified based on solutions to smaller subproblems
 - Top-down approach (by recursion):
 - Solve subproblems are solved recursively
 - Base-case: solutions to the smallest subproblems are computed trivially (no recursion)
 - Bottom-up:
 - Smallest subproblems are solved trivially
 - Specify the solution to each subproblem based on the solutions to smaller subproblems (using recurrence relations)
 - ***Do not find the solution to the same problem more than one time (memorization)***

Top-Down implementation with memorized recursion

```
map <problem, value> Memory;
value DP(problem P) {
    if (is_base_case(P))
        return base_case_value(P);

    if (Memory.find(P) != Memory.end())
        return Memory [P];

    value result = some value ;
    for (problem Q in subproblems(P))
        result = Combine(result, DP(Q));

    Memory [P] = result ;
    return result ;
}
```


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:

$$\text{Fib}(1) = 1$$

$$\text{Fib}(2) = 1$$

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

Fibonacci numbers

- Store calculated results

```
map<int, int> mem;
```

```
int Fib(int n){  
    if (n <= 2) return 1;  
  
    if (mem.find(n) != mem.end())  
        return mem[n];  
  
    int res = Fib(n - 1) + Fib(n - 2);  
    mem[n] = res;  
  
    return res;  
}
```

Fibonacci numbers

```
const int N = 1e4 + 5;
int mem[N];
memset(mem, -1, sizeof(mem));

int Fib(int n){
    if (n <= 2) return 1;

    if (mem[n] != -1) return mem[n];

    int res = Fib(n - 1) + Fib(n - 2);
    mem[n] = res;

    return res;
}
```

- What is the complexity ?

Fibonacci numbers: Complexity

- There are n possible inputs for the recursive function: $1, 2, \dots, n$
- 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 $O(n \times f)$, 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 $O(n)$

CONTENTS

- Introduction to dynamic programming
- Fibonacci numbers
- **Money change**
- The longest ascending subsequence (Dãy con tăng dài nhất)
- Longest common subsequence (Dãy con chung dài nhất)
- Bitmask

Money change

- Given a set of coins with denominations D_1, D_2, \dots, 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

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, \dots, D_i are allowed to be used.
- Base case:

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

$$MinCoin(i, 0) = 0$$

$$MinCoin(0, x) = \infty$$

- Recurrence relation:

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

Money change: Implementation

```
const int INF = 1e9; const int N = 20;
const int XMAX = 1e5 + 5;
int D[N], mem[N][XMAX];
memset(mem, -1, sizeof(mem));

int MinCoin(int i, int x){
    if (x < 0 || i == 0) return INF;
    if (x == 0) return 0;

    if (mem[i][x] != -1) return mem[i][x];
    int res = INF;
    res = min(res, 1 + MinCoin(i, x - D[i]));
    res = min(res, MinCoin(i - 1, x));

    return mem[i][x] = 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

```
void Trace(int i, int x) {  
    if (x <= 0 || i == 0) return;  
  
    if (mem[i][x] == 1 + mem[i][x - D[i]]){  
        cout << D[i] << ' ';  
        Trace(i, x - D[i]);  
    } else Trace(i - 1, x);  
}
```

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

CONTENTS

- Introduction to dynamic programming
- Fibonacci numbers
- Money change
- **The longest increasing subsequence**
- Longest common subsequence
- Bitmask

Longest increasing subsequence (LIS)

- Given a sequence of n integers $A[1], A[2], \dots, A[n]$. 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

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!

Longest increasing subsequence (LIS)

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

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

Longest increasing subsequence (LIS)

```
const int N = 1e4 + 5;
int a[N], mem[N];
memset(mem, -1, sizeof(mem));

int LIS(i) {
    if (mem[i] != -1) return mem[i];

    int res = 1;
    for (int j = 1; j < i; j++){
        if (a[j] < a[i])
            res = max(res, 1 + LIS(j));
    }
    mem[i] = res;

    return res;
}
```

Longest increasing subsequence (LIS)

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

```
int ans = 0, pos = 0;

for (int i = 1; i <= n; i++){
    if (ans < mem[i]){
        ans = mem[i];
        pos = i;
    }
}
cout << ans << endl;
```


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 $O(n \log n)$.
- Another improved method is to use a new dynamic programming formula that incorporates binary search which also gives $O(n \log n)$
- Trace?

Longest increasing subsequence (LIS): improvement using binary search

- Scan the sequence $A[1], A[2], \dots, A[n]$ from left to right
 - For each index i , maintain a list L of increasing subsequences $L[1], L[2], \dots, L[k]$, of lengths 1, 2, 3, \dots , k of the sequence $A[1], A[2], \dots, A[i]$.
 - Each increasing subsequence maintains that last element $x[1], x[2], \dots, x[k]$ which are the smallest possible (we have the property: $x[1] < x[2] < \dots < x[k]$)
 - When moving from index i to index $i+1$, we update L as follows:
 - Perform the binary search on $x[1], x[2], \dots, x[k]$ to find the index j of the smallest element which is greater or equal to $A[i+1]$ (can use the `lower_bound(.)` function of C++)
 - If no index found, then create new list $L[k+1]$ by adding $A[i+1]$ at the end of $L[k]$, add $L[k+1]$ at the end of L
 - Otherwise, update $x[j] = A[i+1]$

Longest increasing subsequence (LIS): improvement using binary search

- Example $A = [4, 2, 6, 3, 8, 4, 5, 9, 6, 1]$
 - $i = 1: A[1] = 4 \rightarrow L[1] = [4]$
 - $i = 2: A[2] = 2 \rightarrow L[1] = [2]$
 - $i = 3: A[3] = 6 \rightarrow L[1] = [2], L[2] = [2, 6]$
 - $i = 4: A[4] = 3 \rightarrow L[1] = [2], L[2] = [2, 3]$ (replace $[2, 6]$ by $[2, 3]$ in which 3 is smaller than 6)
 - $i = 5: A[5] = 8 \rightarrow L[1] = [2], L[2] = [2, 3], L[3] = [2, 3, 8]$ (create new list $L[3]$ by adding 8 at the end of $L[2]$)
 - $i = 6: A[6] = 4 \rightarrow L[1] = [2], L[2] = [2, 3], L[3] = [2, 3, 4]$ (replace $[2, 3, 8]$ by $[2, 3, 4]$ in which 4 is smaller than 8)
 - $i = 7: A[7] = 5 \rightarrow L[1] = [2], L[2] = [2, 3], L[3] = [2, 3, 4], L[4] = [2, 3, 4, 5]$ (create new list $L[4]$)
 - $i = 8: A[8] = 9 \rightarrow L[1] = [2], L[2] = [2, 3], L[3] = [2, 3, 4], L[4] = [2, 3, 4, 5], L[5] = [2, 3, 4, 5, 9]$
 - $i = 9: A[9] = 6 \rightarrow L[1] = [2], L[2] = [2, 3], L[3] = [2, 3, 4], L[4] = [2, 3, 4, 5], L[5] = [2, 3, 4, 5, 6]$
 - $i = 10: A[10] = 1 \rightarrow L[1] = [1], L[2] = [2, 3], L[3] = [2, 3, 4], L[4] = [2, 3, 4, 5], L[5] = [2, 3, 4, 5, 6]$

Longest increasing subsequence (LIS): improvement using binary search

```
#include <bits/stdc++.h>
using namespace std;
vector<int> A;
int n;
vector<int> x;
void input(){
    scanf("%d",&n);
    for(int i = 0; i < n; i++){
        int v; scanf("%d",&v); A.push_back(v);
    }
}
```

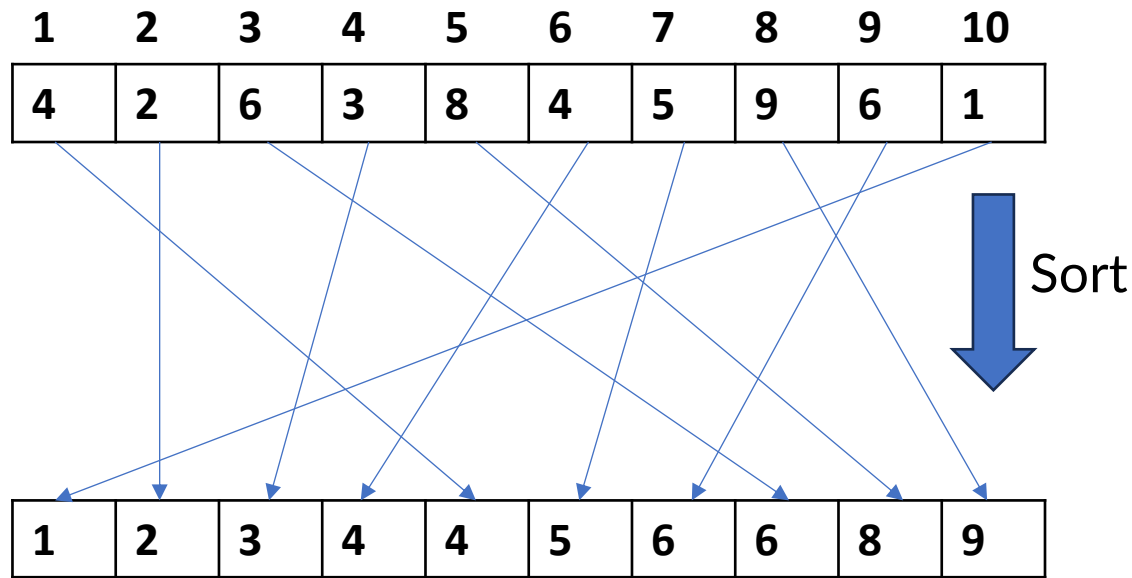
```
int main(){
    input();
    for(int i = 0; i < n; i++){
        vector<int>::iterator p =
            lower_bound(x.begin(), x.end(), A[i]);
        int lb = p - x.begin();
        if(lb == x.size()){ x.push_back(A[i]); }
        else{ x[lb] = A[i]; }
    }
    int res = x.size();
    cout << res;
    return 0;
}
```

Longest increasing subsequence (LIS): improvement using segment trees

- Sort the sequence in non-decreasing order of values, maintain indices of the elements in the original sequence:
 - $\text{index}[i]$ is the index of element $A[i]$ in the sorted sequence
 - If two elements $A[i] = A[j]$ with $i < j$, then $A[i]$ is located after $A[j]$ in the sorted sequence: $\text{index}[i] > \text{index}[j]$

Longest increasing subsequence (LIS): improvement using segment trees

- Example: $A = [4, 2, 6, 3, 8, 4, 5, 9, 6, 1]$



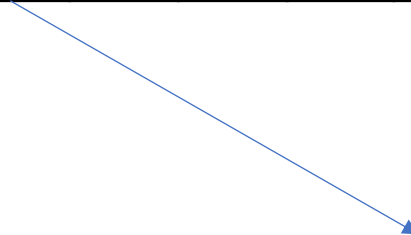
	1	2	3	4	5	6	7	8	9	10
A	4	2	6	3	8	4	5	9	6	1
index	5	2	8	3	9	4	6	10	7	1

- Scan the original sequence from left to right, for each index i :
 - compute the length $LIS[i]$ of the longest increasing subsequence of $A[1], A[2], \dots, A[i]$:
 - $LIS[i] = \text{result}[\text{index}[i]] = \max(\text{result}[1], \dots, \text{result}[\text{index}[i] - 1]) + 1$
 - Use a segment tree to query $\max(\text{result}[1], \dots, \text{result}[\text{index}[i] - 1])$

Longest increasing subsequence (LIS): improvement using segment trees

- Example: $A = [4, 2, 6, 3, 8, 4, 5, 9, 6, 1]$
- Step $i = 1$: $\text{index}[1] = 5$, $\text{LIS}[1] = \text{result}[5] = \max(\text{result}[1..4] + 1) = 1$

	1	2	3	4	5	6	7	8	9	10
A	4	2	6	3	8	4	5	9	6	1
index	5	2	8	3	9	4	6	10	7	1




	1	2	3	4	5	6	7	8	9	10
SA	1	2	3	4	4	5	6	6	8	9
result	0	0	0	0	1	0	0	0	0	0

Longest increasing subsequence (LIS): improvement using segment trees

- Example: $A = [4, 2, 6, 3, 8, 4, 5, 9, 6, 1]$
- Step $i = 2$: $\text{index}[2] = 2$, $\text{LIS}[2] = \text{result}[2] = \max(\text{result}[1..1] + 1) = 1$

	1	2	3	4	5	6	7	8	9	10
A	4	2	6	3	8	4	5	9	6	1
index	5	2	8	3	9	4	6	10	7	1



	1	2	3	4	5	6	7	8	9	10
SA	1	2	3	4	4	5	6	6	8	9
result	0	1	0	0	1	0	0	0	0	0

Longest increasing subsequence (LIS): improvement using segment trees

- Example: $A = [4, 2, 6, 3, 8, 4, 5, 9, 6, 1]$
- Step $i = 3$: $\text{index}[3] = 8$, $\text{LIS}[3] = \text{result}[8] = \max(\text{result}[1..7] + 1) = 2$

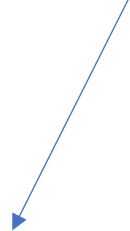
	1	2	3	4	5	6	7	8	9	10
A	4	2	6	3	8	4	5	9	6	1
index	5	2	8	3	9	4	6	10	7	1

	1	2	3	4	5	6	7	8	9	10
SA	1	2	3	4	4	5	6	6	8	9
result	0	1	0	0	1	0	0	2	0	0

Longest increasing subsequence (LIS): improvement using segment trees

- Example: $A = [4, 2, 6, 3, 8, 4, 5, 9, 6, 1]$
- Step $i = 4$: $\text{index}[4] = 3$, $\text{LIS}[4] = \text{result}[3] = \max(\text{result}[1..2] + 1) = 2$

	1	2	3	4	5	6	7	8	9	10
A	4	2	6	3	8	4	5	9	6	1
index	5	2	8	3	9	4	6	10	7	1



	1	2	3	4	5	6	7	8	9	10
SA	1	2	3	4	4	5	6	6	8	9
result	0	1	2	0	1	0	0	2	0	0

Longest increasing subsequence (LIS): improvement using segment trees

- Example: $A = [4, 2, 6, 3, 8, 4, 5, 9, 6, 1]$
- Step $i = 5$: $\text{index}[5] = 9$, $\text{LIS}[5] = \text{result}[9] = \max(\text{result}[1..8] + 1) = 3$

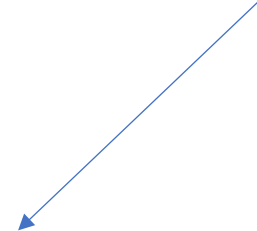
	1	2	3	4	5	6	7	8	9	10
A	4	2	6	3	8	4	5	9	6	1
index	5	2	8	3	9	4	6	10	7	1

	1	2	3	4	5	6	7	8	9	10
SA	1	2	3	4	4	5	6	6	8	9
result	0	1	2	0	1	0	0	2	3	0

Longest increasing subsequence (LIS): improvement using segment trees

- Example: $A = [4, 2, 6, 3, 8, 4, 5, 9, 6, 1]$
- Step $i = 6$: $\text{index}[6] = 4$, $\text{LIS}[6] = \text{result}[4] = \max(\text{result}[1..3] + 1) = 3$

	1	2	3	4	5	6	7	8	9	10
A	4	2	6	3	8	4	5	9	6	1
index	5	2	8	3	9	4	6	10	7	1

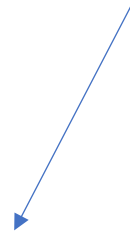


	1	2	3	4	5	6	7	8	9	10
SA	1	2	3	4	4	5	6	6	8	9
result	0	1	2	3	1	0	0	2	3	0

Longest increasing subsequence (LIS): improvement using segment trees

- Example: $A = [4, 2, 6, 3, 8, 4, 5, 9, 6, 1]$
- Step $i = 7$: $\text{index}[7] = 6$, $\text{LIS}[7] = \text{result}[6] = \max(\text{result}[1..5] + 1) = 4$

	1	2	3	4	5	6	7	8	9	10
A	4	2	6	3	8	4	5	9	6	1
index	5	2	8	3	9	4	6	10	7	1



	1	2	3	4	5	6	7	8	9	10
SA	1	2	3	4	4	5	6	6	8	9
result	0	1	2	3	1	4	0	2	3	0

Longest increasing subsequence (LIS): improvement using segment trees

- Example: $A = [4, 2, 6, 3, 8, 4, 5, 9, 6, 1]$
- Step $i = 8$: $\text{index}[8] = 10$, $\text{LIS}[8] = \text{result}[10] = \max(\text{result}[1..9] + 1) = 5$

	1	2	3	4	5	6	7	8	9	10
A	4	2	6	3	8	4	5	9	6	1
index	5	2	8	3	9	4	6	10	7	1

	1	2	3	4	5	6	7	8	9	10
SA	1	2	3	4	4	5	6	6	8	9
result	0	1	2	3	1	4	0	2	3	5

Longest increasing subsequence (LIS): improvement using segment trees

- Example: $A = [4, 2, 6, 3, 8, 4, 5, 9, 6, 1]$
- Step $i = 9$: $\text{index}[9] = 7$, $\text{LIS}[9] = \text{result}[7] = \max(\text{result}[1..6] + 1) = 5$

	1	2	3	4	5	6	7	8	9	10
A	4	2	6	3	8	4	5	9	6	1
index	5	2	8	3	9	4	6	10	7	1

	1	2	3	4	5	6	7	8	9	10
SA	1	2	3	4	4	5	6	6	8	9
result	0	1	2	3	1	4	5	2	3	5

Longest increasing subsequence (LIS): improvement using segment trees

- Example: $A = [4, 2, 6, 3, 8, 4, 5, 9, 6, 1]$
- Step $i = 10$: $\text{index}[10] = 1$, $\text{LIS}[10] = \text{result}[1] = \max(\text{result}[0..0] + 1) = 1$

	1	2	3	4	5	6	7	8	9	10
A	4	2	6	3	8	4	5	9	6	1
index	5	2	8	3	9	4	6	10	7	1

	1	2	3	4	5	6	7	8	9	10
SA	1	2	3	4	4	5	6	6	8	9
result	1	1	2	3	1	4	5	2	3	5

Longest increasing subsequence (LIS): Tracing using recursion

```
void Trace(int i) {  
    for (int j = 1; j < i; j++){  
        if (a[j] < a[i] && mem[i] == 1 + mem[j]){  
            Trace(j);  
            break;  
        }  
    }  
    cout << a[i] << ' ' ;  
}
```

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

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

```
void Trace(int i) {  
    for (int j = i - 1; j >= 1; j--){  
        if (a[j] < a[i] && mem[i] == 1 + mem[j]){  
            Trace(j);  
            break;  
        }  
    }  
    cout << a[i] << ' ' ;  
}
```

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

Longest increasing subsequence (LIS): Tracing using loop

```
stack<int> stk;
int i = pos;

for (int k = 0; k < ans; k++) {
    stk.push(i);
    for(int j = i - 1; j >= 1; j--) {
        if (a[j] < a[i] && mem[i] == 1 + mem[j]) {
            i = j;
            break;
        }
    }
}

while (!stk.empty()) {
    cout << stk.top() << ' ';
    stk.pop();
}
```

CONTENTS

- Introduction to dynamic programming
- Fibonacci numbers
- Money change
- The longest ascending subsequence (Dãy con tăng dài nhất)
- **Longest common subsequence (Dãy con chung dài nhất)**
- Dynamic Programming with Bitmasking (TSP)

Longest common subsequence (Dãy con chung dài nhất)

- Given two strings (or two integer arrays) of n and m elements $X[1], X[2], \dots, X[n]$ và $Y[1], Y[2], \dots, Y[m]$. Find the length of the longest common subsequence of the two strings.
- Example: $X = \text{"abcb"}$, $Y = \text{"bdcab"}$
- The longest common subsequence of X and Y is là "bcb" which has the length 3.

Longest common subsequence: Dynamic programming formular

- Let $LCS(i, j)$ be the length of the longest common subsequence of the sequences $X[1], X[2], \dots, X[i]$ and $Y[1], Y[2], \dots, Y[j]$.

- Basic step:

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

- Inductive step:

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

Longest common subsequence: Implement

```
const int N = 1e4 + 5;
string X, Y;
int mem[N][N];
memset(mem, -1, sizeof(mem));

int LCS(int i, int j) {
    if (i == 0 || j == 0) return 0;
    if (mem[i][j] != -1) return mem[i][j];
    int res = 0;
    res = max(res, LCS(i - 1, j));
    res = max(res, LCS(i, j - 1));
    if (X[i-1] == Y[j-1]) {
        res = max(res, 1 + LCS(i - 1, j - 1));
    }
    return mem[i][j] = res;
}
```

Longest common subsequence: Example

iMem	j	0	1	2	3	4	5
i		Y[j]	<u>b</u>	d	<u>c</u>	a	<u>b</u>
0	X[i]	0	0	0	0	0	0
1	a	0 ↘	0	0	0	1	1
2	<u>b</u>	0	1 →	1 ↘	1	1	2
3	<u>c</u>	0	1	1	2 →	2 ↘	2
4	<u>b</u>	0	1	1	2	2	3

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

Longest common subsequence: Complexity

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

Longest common subsequence: Tracing by using recursion

```
void Trace(int i, int j) {  
    if (i == 0 || j == 0) return;  
    if (X[i-1] == Y[j-1] && mem[i][j] == 1 + mem[i-1][j-1]) {  
        Trace(i - 1, j - 1);  
        cout << X[i-1];  
        return;  
    }  
    if (mem[i][j] == mem[i-1][j]) {  
        Trace(i - 1, j);  
        return;  
    }  
    if (mem[i][j] == mem[i][j-1]) {  
        Trace(i, j - 1);  
        return;  
    }  
}
```

Longest common subsequence: Tracing by using recursion

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

Longest common subsequence: Tracing by using loop

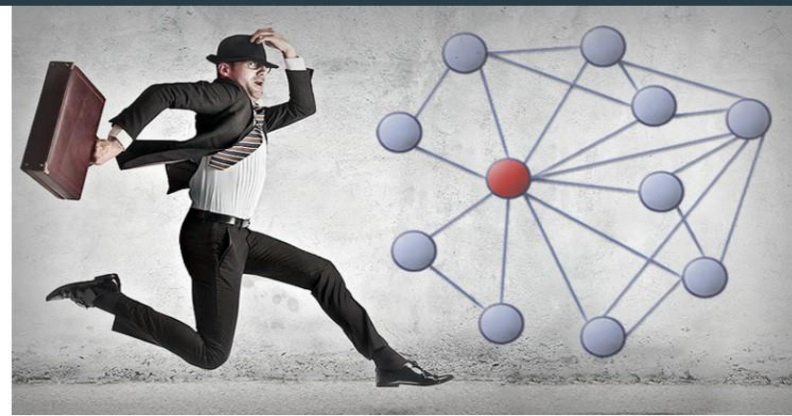
```
void IterTrace() {
    int i = X.size(), j = Y.size();
    stack<int> stk;
    int ans = LCS(i, j);
    for (int k = 0; k < ans; k++) {
        if (X[i - 1] == Y[j - 1] && mem[i][j] == 1 + mem[i-1][j-1]) {
            stk.push(X[i - 1]);    i--; j--; continue;
        }
        if (mem[i][j] == mem[i-1][j]) {
            i--; continue;
        }
        if (mem[i][j] == mem[i][j-1]) {
            j--; continue;
        }
    }
    while (!stk.empty()){
        cout << stk.top() << ' ';
        stk.pop();
    }
}
```

CONTENTS

- Introduction to dynamic programming
- Fibonacci numbers
- Money change
- The longest increasing subsequence (Dãy con tăng dài nhất)
- Longest common subsequence (Dãy con chung dài nhất)
- **Dynamic Programming with Bitmasking (TSP)**

- Do you remember the bitmask representation for subsets?
- Each subset of a set of n elements is represented by an integer in the range $0, 1, \dots, 2^n - 1$
- This can help implement dynamic programming methods easily on subsets

Traveling salesman problem



Applying the Traveling Salesman Problem to Business Analytics

Published on April 2, 2016

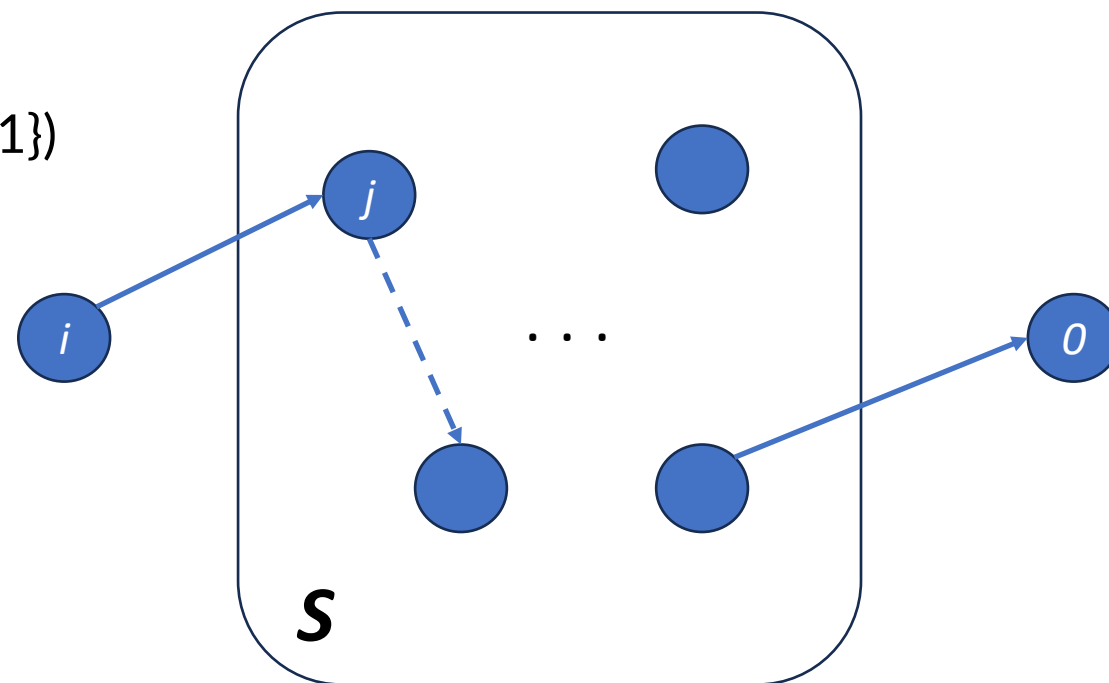
- *“The history and evolution of solutions to the Traveling Salesman Problem can provide us with some valuable concepts for business analytics and algorithm development”*
- *“Just as the traveling salesman makes his journey, new analytical requirements arise that require a journey into the development of solutions for them. As the data science and business analytics landscape evolves with new solutions, we can learn from the history of these journeys and apply the same concepts to our ongoing development”*

Traveling salesman problem

- Given a graph of n vertices $\{0, 1, \dots, n - 1\}$ and the weight value $C_{i,j}$ on each pair of vertices i, j . Find a cycle that goes through all the vertices of the graph, each vertex exactly once, so that the sum of the weights on that cycle is minimum.

Traveling salesman problem: Dynamic programming

- Without loss of generality, assume the cycle begins and ends at vertex 0.
- Let $TSP(i, S)$ be the shortest route starting from vertex i , visiting all vertices in S and returning to vertex 0.
- Base case: $TSP(i, \{\}) = C_{i,0}$
- Recurrence relation:
 - $TSP(i, S) = \min_{j \in S} \{ C_{i,j} + TSP(j, S \setminus \{j\}) \}$
- Solution to the original problem is $TSP(0, \{1, 2, \dots, n-1\})$



Traveling salesman problem: Dynamic programming

- Bitmasking:
 - A subset of $\{0, 1, 2, \dots, n-1\}$ is represented by a sequence of bits $b_{n-1}b_{n-2}\dots b_0$
 - $b_i = 1$ means that i belongs to the subset, and $b_i = 0$ means that i does not belong to the subset
 - The sequence $b_{n-1}b_{n-2}\dots b_0$ is represented by a non-negative integer N
 - Perform bit manipulation on N to reflect the corresponding set operations: insertion, removal, existence checking

```
unsigned int removeItem(unsigned int S, int i){
    return S ^ (1 << i);
}
unsigned int addItem(unsigned int S, int i){
    return S | (1 << i);
}
```

```
bool contains(unsigned int S, int i){
    if(((S >> i) & 1) > 0)
        return true;
    else return false;
}
```

Traveling salesman problem: Dynamic programming

```
int main(){
    scanf("%d",&n);
    for(int i = 0; i <= n-1; i++)
        for(int j = 0; j <= n-1; j++)
            scanf("%d",&d[i][j]);
    unsigned int S = 0;
    for(int i = 1; i <= n-1; i++)
        S = addItem(S,i);
    for(int i = 0; i < 32; i++)
        for(int j = 0; j < N; j++)
            M[i][j] = -1;
    cout << TSP(0,S);
}
```

```
int TSP(int i, unsigned int S){
    if(S == 0){ M[i][S] = d[i][0]; return M[i][S]; }
    if(M[i][S] == -1){// subproblem not solved
        M[i][S] = 100000000;
        for(int j = 0; j <= n-1; j++){
            if(contains(S,j)){
                int A = TSP(j,removeItem(S,j));
                if(M[i][S] > d[i][j] + A){ M[i][S] = d[i][j] + A; }
            }
        }
    }
    return M[i][S];
}
```

A large graphic on the left side of the slide. It features a dark blue background with a circular pattern of red dots of varying sizes, creating a sense of depth and movement. The word "HUST" is centered within this graphic in a white, bold, sans-serif font.

HUST

THANK YOU !