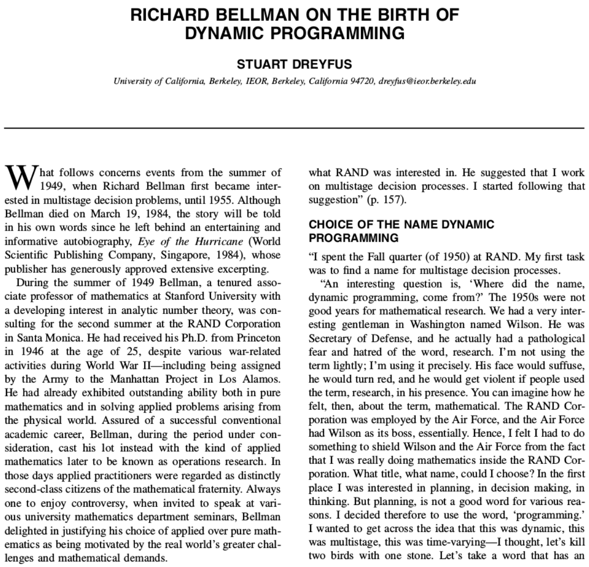
ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Dynamic Programming

APPLIED ALGORITHMS



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

Hình: R.E.Bellman (1920-1984)

Some basic algorithm paradigms

- Exhautive search
- Divide and conquer
- Dynamic programming
- Greedy

Each algorithm is applied to solve different kinds of problems

What is dynamic programming?

- is an algorithm
- Some similarities between two algorithms: Dynamic programming and Divide and Conquer
- Divide and conquer:
  - Divide the problem into *independent* subproblems
  - Solve each subproblem (recursively)
  - Combine solutions of subproblems to obtain the solution of the problem
- Dynamic programming:
  - Divide the problem into *overlapping* subproblems
  - Solve each subproblem (recursively)
  - Combine solutions of subproblems to obtain the solution of the problem
  - *Do not solve the same problem more than once*

Dynamic programming formulation

1. Find dynamic programming formulation based on subproblems
2. Implement dynamic programming formulation:
   Simply convert the formulation to a recursive function
3. Store the results of calculated functiona

Comments

Step 1: Finding the dynamic programming formulation is the hardest and most important step. Doing steps 2 and 3, we can apply the following general scheme:

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;
}
```

Comments

- Using recursive function to implement dynamic programming formulation is a natural and simple programming approach, we call it Top-Down programming approach, suitable for dynamic programming newbies.
- When fimilar with dynamic programming, we can practice Bottom-Up approach, gradually building solutions from subproblems to parent problems
- The above steps only find the optimal value of the problem. If it is necessary to show elements of the optimal solution, we need to perform addition Tracing step. The Tracing step should simulate Step2 of recursive implementation and find elements of optimal solution based on the information of subproblems stored in array memmory

Fibonacci calculation

The first 2 numbers of Fibonacci sequences are 1 and 1. Each other number is calculated as the sum of the two numbers immediately preceding it in the sequence

- Requirement: Calculate the n Fibonacci number
- Solve the problem by using Dynamic programming

1. Find dynamic programming formulation:

$$\begin{aligned}\mathrm{Fib}(1) &= 1\\ \mathrm{Fib}(2) &= 1\\ \mathrm{Fib}(n) &= \mathrm{Fib}(n-2) + \mathrm{Fib}(n-1)\end{aligned}$$

Fibonacci calculation

2. Implement dynamic programming formulation

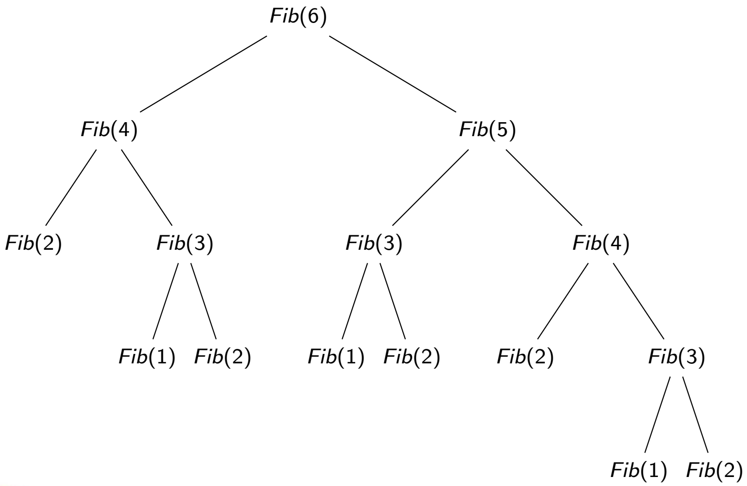
```
int Fib(int n) {
    if (n <= 2)
        return 1;

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

    return res;
}
```

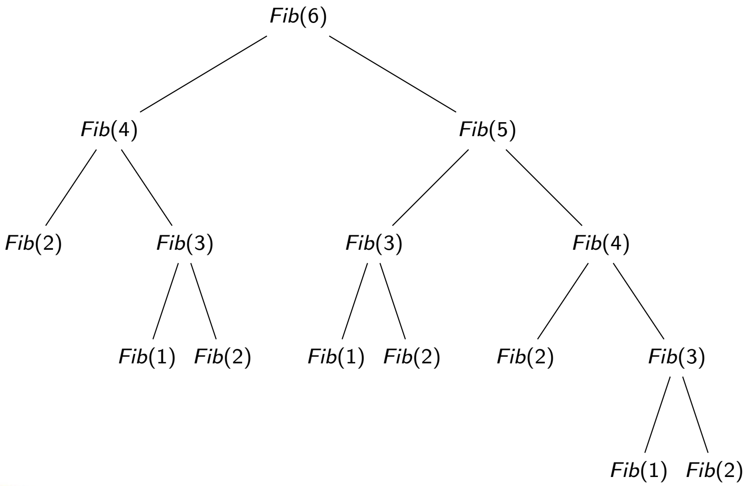
Fibonacci calculation

- What is the complexity?



Fibonacci calculation

- What is the complexity? Exponential function, almost $\mathcal{O}(2^n)$



Fibonacci calculation

3. Store results of calculated functions

```
map<int, int> Mem;

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

    if (Mem.find(n) != Mem.end())
        return Mem[n];

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

    Mem[n] = res;
    return res;
}
```

Fibonacci sequence

```
int iMem[1001];
for (int i = 1; i <= 1000; i++)
    iMem[i] = -1;

int Fib(int n) {
    if (n <= 2)
        return 1;
    if (iMem[n] != -1)
        return iMem[n];

    int res = Fib(n - 2) + Fib(n - 1);
    iMem[n] = res;
    return res;
}
```

- What is the complexity now?

Fibonacci calculation: The complexity

- We have n input possibilities for recursive function: 1, 2, . . . , n.
- For each input:
  - either we need to calculate the result and then store it
  - or we just look up the result from the memory if the corresponding problem of this input has been already solved
- Each input is calculated at most once
- The computation time is $\mathcal{O}(n \times f)$, where f is the computation time of recursive function for an input, with the assumption that the previously calculated results will be retrieved directly from memory, in only $\mathcal{O}(1)$
- Since it only costs a constant amount of computation for an input of the function, so $f = O(1)$
- The total computation time is $\mathcal{O}(n)$

Maximum subarray

- Given an array of integers $\mathrm{A}[1], \mathrm{A}[2], \ldots, \mathrm{A}[n]$, find a subarray of the array so that the total sum of numbers in this subarray is maximum

-16	7	-3	0	-1	5	-4

Maximum subarray

- Given an array of integers $\mathrm{A}[1], \mathrm{A}[2], \ldots, \mathrm{A}[n]$, find a subarray of the array so that the total sum of numbers in this subarray is maximum

-16	7	-3	0	-1	5	-4

- The sum of maximum subarray in this array is 8

Maximum subarray

- Given an array of integers $\mathrm{A}[1], \mathrm{A}[2], \ldots, \mathrm{A}[n]$, find a subarray of the array so that the total sum of numbers in this subarray is maximum

-16	7	-3	0	-1	5	-4

- The sum of maximum subarray in this array is 8
- Remind:
  - If using Divide and conquer, it requires $\mathcal{O}(n \log n)$
  - Is it possible to get better computation time with Dynamic programming?

The subarray with maximum sum: Wrong formulation

- The first step is to fund dynamic programming formulation
- Denote $\mathrm{MaxSum}(i)$ the sum of numbers in the maximum subarray among those selected in sequence $1, \ldots, i$
- Basic step: $\mathrm{MaxSum}(1) = \max(0, A[1])$
- Inductive step: $\mathrm{MaxSum}(i)$ has any connections with $\mathrm{MaxSum}(i-1)$?
- Is it possible to combine solutions of subproblems of size less than i into solutions of problems of size i?
- The answer is not entirely obvious ...

The subarray with maximum sum: formulation

Build new objective function:

- Denote $\mathrm{MaxSum}(i)$ the sum of numbers in the maximum subarray among those selected in sequence $1, \ldots, i$, *and this subarray must be ended at i*
- Basic step: $\mathrm{MaxSum}(1) = A[1]$
- Inductive step:
  $\mathrm{MaxSum}(i) = \max(A[i], A[i] + \mathrm{MaxSum}(i-1))$
- Thus the formulation is: $\max_{1 \le i \le n} \{ \mathrm{MaxSum}(i) \}$

The subarray with maximum sum: Implementation

- Next step is to implement Dynamic programming formulation

```
int A[1001];

int MaxSum(int i) {
    if (i == 1)
        return A[i];

    int res = max(A[i], A[i] + MaxSum(i - 1));
    return res;
}
```

The subarray with maximum sum: Implementation

- Next step is to implement Dynamic programming formulation

```
int A[1001];

int MaxSum(int i) {
    if (i == 1)
        return A[i];

    int res = max(A[i], A[i] + MaxSum(i - 1));
    return res;
}
```

- Do not to use array `Memory` for memorization, why?

The subarray with maximum sum: Implementation

- Next step is to implement Dynamic programming formulation

```
int A[1001];

int MaxSum(int i) {
    if (i == 1)
        return A[i];

    int res = max(A[i], A[i] + MaxSum(i - 1));
    return res;
}
```

- Do not to use array `Memory` for memorization, why?
- But we need array `Memory` for Tracing step!

The subarray with maximum sum: Implementation

```
int A[1001];
int iMem[1001];
bool bMark[1001];
memset(bMark, 0, sizeof(bMark));

int MaxSum(int i) {
    if (i == 1)
        return A[i];
    if (bMark[i])
        return iMem[i];

    int res = max(A[i], A[i] + MaxSum(i - 1));
    iMem[i] = res;
    bMark[i] = true;
    return res;
}
```

The subarray with maximum sum: Result

- The main program needs to call $\mathrm{MaxSum}(n)$ once, this recursive function will calculate all values for $\mathrm{MaxSum}(i), 1 \leq i \leq n$
- The solution to problem is the maximum value among values $\mathrm{MaxSum}(i)$ which are already stored in $\mathrm{iMem}[i]$ after the recursive call

```
int ans = 0;
for (int i = 0; i < n; i++) {
    ans = max(ans, iMem[i]);
}
cout << ans;
```

- If the problem requires to find maximum subarrays of different arrays, then we need to clear the memory at the end of the calculation in each array.

The subarray with maximum sum: The complexity

- There are n input possibilities for recursive function
- Each input is calculated in $\mathcal{O}(1)$
- The total computation time is $\mathcal{O}(n)$

The subarray with maximum sum: The complexity

- There are *n* input possibilities for recursive function
- Each input is calculated in $\mathcal{O}(1)$
- The total computation time is $\mathcal{O}(n)$
- How to know the maximum subarray consisting of which numbers?

The subarray with maximum sum: Tracing by recursion

- Do the same as the main recursive function and use the existing array `iMem` to trace back

```
void Trace(int i) {
  if (i != 1 && iMem[i] == A[i] + iMem[i-1]) {
     Trace(i - 1);
  }
  cout << A[i] << " ";
}
```

- The complexity of tracing function?

The subarray with maximum sum: Tracing by recursion

- Do the same as the main recursive function and use the existing array iMem to trace back

```
void Trace(int i) {
  if (i != 1 && iMem[i] == A[i] + iMem[i-1]) {
     Trace(i - 1);
  }
  cout << A[i] << " ";
}
```

- The complexity of tracing function? $\mathcal{O}(n)$

The subarray with maximum sum: Tracing by loop

```
int ans = 0, pos = -1;
for (int i = 0; i < n; i++) {
    ans = max(res, iMem[i]);
    if (ans == iMem[i]) pos = i;
}

cout << ans << endl;
int first = pos, last = pos, sum = A[first];
while (sum != res){
    --first;
    sum += A[first];
}
cout << first << " " << last;
```

Cash exchange

- Given a set of coins in denominations $D_1, D_2, \ldots, D_n$, and a denomination x. Find the minimum number of coins to exchange for the value x?
- Similar to bin packing problem?
- Exist the greedy algorithm for this Cash exchange problem?

Cash exchange

- Given a set of coins in denominations $D_1, D_2, \ldots, D_n$, and a denomination x. Find the minimum number of coins to exchange for the value x?
- Similar to bin packing problem?
- Exist the greedy algorithm for this Cash exchange problem?
- The bin packing problem studied in `Discrete Maths` is solved by using Branch and Bound algorithm. Greedy algorithm is not guarantee to give the optimal solution, even in many cases it does not give the solution..
- Try using Dynamic programming !
- Finally give the comments for different approaches to solve this problem

Cash exchange: Dynamic programming formulation

First step: build Dynamic programming formulation

- Denote $\mathrm{MinCoin}(i, x)$ the minimum number of coins required to exchange for denomination x if we can only use denominations D_1, $\ldots$, D_i
- Basic steps:
  - $\mathrm{MinCoin}(i, x) = \infty$ nếu $x < 0$
  - $\mathrm{MinCoin}(i, 0) = 0$
  - $\mathrm{MinCoin}(0, x) = \infty$
- Inductive step:

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

Cash exchange: Implementation

```
int INF = 100000;
int D[11];

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

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

    return res;
}
```

Cash exchange: Implementation

```
int INF = 100000;
int D[11];
int iMem[11][10001];
memset(iMem, -1, sizeof(iMem));

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

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

Cash exchange: The complexity

- The complexity?

Cash exchange: The complexity

- The complexity?
- The number of input possibilities is $n \times x$
- Each input is processed in $\mathcal{O}(1)$, assuming each recursive call executes in constant time
- The total computation time is $\mathcal{O}(n \times x)$

Cash exchange: The complexity

- The complexity?
- The number of input possibilities is $n \times x$
- Each input is processed in $\mathcal{O}(1)$, assuming each recursive call executes in constant time
- The total computation time is $\mathcal{O}(n \times x)$
- How to determine which coins used in the optimal solution ?
- Let's trace back the recurstion

Cash exchange: Tracing by recursion

```
void Trace(int i, int x) {
    if (x < 0) return;
    if (x == 0) return;
    if (i == 0) return;

    int res = INF;
    if (iMem[i][x] == 1 + iMem[i][x - D[i]]){
        cout << D[i] << " ";
        Trace(i, x - D[i]);
    } else {
        Trace(i-1, x);
    }
}
```

- Call `Trace(n,x);`
- The complexity of tracing function?

Cash exchange: Tracing by recursion

```
void Trace(int i, int x) {
    if (x < 0) return;
    if (x == 0) return;
    if (i == 0) return;

    int res = INF;
    if (iMem[i][x] == 1 + iMem[i][x - D[i]]){
        cout << D[i] << " ";
        Trace(i, x - D[i]);
    } else {
        Trace(i-1, x);
    }
}
```

- Call `Trace(n,x);`
- The complexity of tracing function? $\mathcal{O}(\max(n, x))$

Cash exchange: Tracing by loop

```
int ans = iMem[n][x];
cout << ans << endl;
for (int i = n, k = 0; k < ans; ++k) {
    if (iMem[i][x] == 1 + iMem[i][x-D[i]]){
        cout << D[i] << " ";
        x -= D[i];
    } else {
        --i;
    }
}
```

Longest increasing subsequence

- Given an array of n integers $A[1], A[2], \ldots, A[n]$, find the length of the longest increasing subsequence?
- Definition: If deleting either 0 or some elements from array A, then we obtain a subsequence of A
- Ví dụ: $A = [2, 0, 6, 1, 2, 9]$
- $[2, 6, 9]$ 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
- $[9, 0]$ **not** sequence of A
- $[7]$ **not** sequence of A

Longest increasing subsequence

- An increasing subsequence of A is a subsequence of A in which elements are strictly increasing from left to right
- $[2, 6, 9]$ and $[1, 2, 9]$ are two increasing subseqeces of $A = [2, 0, 6, 1, 2, 9]$
- How to calculate the length of the longest increasing subsequence?
- There are 2^n subsequence, the easiest approach is to browse all these subsequences
- The algorithm with the complexity of $\mathcal{O}(n \times 2^n)$, which can only give the result in acceptable time for $n \leq 23$
- Let's try Dynamic programming !

Longest increasing subsequence: Dynamic programming formulation

- Denote $\mathrm{LIS}(i)$ the length of the longest increasing subsequence of array $A[1], \ldots, A[i]$
- Basic step: $\mathrm{LIS}(1) = 1$
- Inductive step: $\mathrm{LIS}(i) = ?$

Longest increasing subsequence: Dynamic programming formulation

- Denote $\mathrm{LIS}(i)$ the length of the longest increasing subsequence of array $A[1], \ldots, A[i]$
- Basic step: $\mathrm{LIS}(1) = 1$
- Inductive step: $\mathrm{LIS}(i) = ?$
- If setting the objective function like that will encounter the same problem as the maximum subarray problem above, let's change the objective function a bit

Longest increasing subsequence: Dynamic programming formulation

- Denote $\mathrm{LIS}(i)$ the length of the longest increasing subsequence of array $A[1], \ldots, A[i]$, *which ends at i*
- Basic step: not need
- Inductive step:
  $\mathrm{LIS}(i) = \max(1, \max_{j \text{ s.t. } A[j]<A[i]}\{1 + \mathrm{LIS}(j)\})$

Longest increasing subsequence: Implementation

```
int A[1001];
int iMem[1001];
memset(iMem, -1, sizeof(iMem));

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

    int res = 1;
    for (int j = 1; j < i; ++j) {
        if (A[j] < A[i]) {
            res = max(res, 1 + LIS(j));
        }
    }
    iMem[i] = res;
    return res;
}
```

Longest increasing subsequence: Implementation

- The length of the longest increasing subsequence is the maximum value among all $\mathrm{LIS}(i)$:

```
int ans = 0, pos = 0;
for (int i = 1; i <= n; i++) {
    ans = max(ans, iMem[i]);
    if (ans == iMem[i]) pos = i;
}

cout << ans;
```

Longest increasing subsequence: The complexity

- There are n possibilities for input
- Each input is processed in $\mathcal{O}(n)$
- The total computation time is $\mathcal{O}(n^2)$
- It could be possible to run the problem with $n \leq 10\ 000$, that is much better than exhautive search!
- Applying the segment tree structure to the above method could improve the complexity to $\mathcal{O}(n \log n)$
- Another improved method is to combine Dynamic Programming with binary search which also gives complexity $\mathcal{O}(n \log n)$
- Tracing ?

Longest increasing subsequence: Tracing by recursion

```
void Trace(int i) {
     int res = 1;
     for (int j = 1; j < i; j++) {
       if (A[j] < A[i] && iMem[i] == 1 + iMem[j]) {
         Trace(j);
         break;
       }
     }
     cout << i << " ";
}
```

- Call `Trace(pos);`
- The complexity of tracing function?

Longest increasing subsequence: Tracing by recursion

```
void Trace(int i) {
     int res = 1;
     for (int j = 1; j < i; j++) {
       if (A[j] < A[i] && iMem[i] == 1 + iMem[j]) {
         Trace(j);
         break;
       }
     }
     cout << i << " ";
}
```

- Call `Trace(pos);`
- The complexity of tracing function? is $\mathcal{O}(n^2)$
- Can improve to $\mathcal{O}(n)$ by using an array to memorize at each position i the position j creating value $\max(LIS(i))$, so reduce the loop `for` in this tracing function

Longest increasing subsequence: Improved Tracing by recursion

```
void Trace(int i) {
     int res = 1;
     for (int j = i-1; j >= 1; j++) {
       if (A[j] < A[i] && iMem[i] == 1 + iMem[j]) {
         Trace(j);
         break;
       }
     }
     cout << i << " ";
}
```

- Call `Trace(pos);`
- The complexity of tracing function?

Longest increasing subsequence: Improved Tracing by recursion

```
void Trace(int i) {
      int res = 1;
      for (int j = i-1; j >= 1; j++) {
         if (A[j] < A[i] && iMem[i] == 1 + iMem[j]) {
            Trace(j);
            break;
         }
      }
      cout << i << " ";
}
```

- Call `Trace(pos);`
- The complexity of tracing function? $\mathcal{O}(n)$

Longest increasing subsequence: Tracing by loop

```
stack<int> S;
for (int i = pos, k = 0; k < ans; ++k) {
    S.push(i);
    for (int j = 1; j < i; ++j){
        if (A[j] < A[i] && iMem[j]+1 == iMem[i]) {
            i = j;
            break;
        }
    }
    while (!S.empty()){
        cout << S.back() << " ";
        S.pop();
    }
}
```

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Longest common subsequence

- Given two strings (or two integer arrays) of n elements $X[1], \ldots,$ $X[n]$ and m elements $Y[1], \ldots, Y[m]$, find the length of the longest common subsequence of X and Y

- $X =$"$\texttt{a}\underline{\texttt{bcb}}$"
- $Y =$"$\underline{\texttt{b}}\texttt{d}\underline{\texttt{c}}\texttt{a}\underline{\texttt{b}}$"

- The length of the longest common subsequence of X and Y, "`bcb`", is 3

Longest common subsequence: Dynamic programming formulation

- Denote $\mathrm{LCS}(i,j)$ the length of the longest common subsequence of $X[1], \ldots, X[i]$ and $Y[1], \ldots, Y[j]$
- Basic step:
  - $\mathrm{LCS}(0,j) = 0$
  - Basic step: $\mathrm{LCS}(i,0) = 0$
- Inductive step:

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

Longest common subsequence: Implementation

```
string X = "abcb",
       Y = "bdcab";
int iMem[1001][1001];
memset(iMem, -1, sizeof(iMem));

int LCS(int i, int j) {
    if (i == 0 || j == 0)  return 0;
    if (iMem[i][j] != -1)  return iMem[i][j];

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

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

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

Longest common subsequence: The complexity

- There are n possibilities for input
- Each input is processed in $\mathcal{O}(1)$
- The total time is $\mathcal{O}(n \times m)$

Longest common subsequence: The complexity

- There are n possibilities for input
- Each input is processed in $\mathcal{O}(1)$
- The total time is $\mathcal{O}(n \times m)$
- How to know exactly which elements are in the longest common subsequence?

Longest common subsequence: Tracing by recursion

```
void Trace(int i, int j) {
    if (i == 0 || j == 0)  return;

    if (iMem[i][j] == iMem[i-1][j]) {
        Trace(i-1, j);
        return;
    }
    if (iMem[i][j] == iMem[i][j-1]) {
        Trace(i, j-1);
        return;
    }
    if (X[i] == Y[j] && iMem[i][j] == 1 + iMem[i-1][j-1]) {
        Trace(i-1, j-1);
        cout << A[i] << " ";
        return;
    }
}
```

- The complexity of tracing function?

Longest common subsequence: Tracing by recursion

```
void Trace(int i, int j) {
    if (i == 0 || j == 0)  return;

    if (iMem[i][j] == iMem[i-1][j]) {
        Trace(i-1, j);
        return;
    }
    if (iMem[i][j] == iMem[i][j-1]) {
        Trace(i, j-1);
        return;
    }
    if (X[i] == Y[j] && iMem[i][j] == 1 + iMem[i-1][j-1]) {
        Trace(i-1, j-1);
        cout << A[i] << " ";
        return;
    }
}
```

- The complexity of tracing function? $\mathcal{O}(n+m)$

Longest common subsequence: Tracing by loop

```
int ans = iMem[n][m];
cout << ans << endl;
stack<int> S;
for (int i = n, j = m, k = 0; k < ans; ++k) {
    if (X[i] == Y[j] && iMem[i][j] == 1 + iMem[i-1][j-1]){
        S.push(X[i]);
        --i; --j; continue;
    }
    if (iMem[i][j] == iMem[i-1][j]){
        --i; continue;
    }
    if (iMem[i][j] == iMem[i][j-1]){
        --j; continue;
    }
}
while (!S.empty()) {
    cout << S.back() << " ";
    S.pop();
}
```

Dynamic programming with Bitmasks

- Do you remember how to use bitmasks to represent subsets?
- Each subset of the n-element set is represented by an integer number in the range $0, \ldots, 2^n - 1$
- This can make dynamic programming easy 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

Applying the Traveling Salesman Problem to Business Analytics

Published on April 2, 2016

The traveling salesman problem is a classic NP-hard problem, but has many practical applications, especially nowadays with the application of data science and financial analysis. Every time a salesman's itinerary ends, the data is analyzed by algorithms in data science, applied to financial analysis, which can 'machine learning' the previous results to apply to the next development plan.

Traveling salesman problem

- Given a graph of n vertices $\{0,1,\ldots,n-1\}$ and the weight value $C_{i,j}$ on each pair of vertices i, j. Find a cycle that passes through all the vertices of the graph exactly once so that the sum of the weights on that cycle is minimal.
- This problem is NP-hard, there is thus no known polynomial-time deterministic algorithm to solve this problem
- A exhautive search simply traverses all permutations of vertices, thus gives the complexity of $\mathcal{O}(n!)$, but can only run in acceptable time for $n \leq 11$
- Could it be done better with Dynamic Programming?

Traveling salesman problem: Dynamic programming formulation

- Without loss of generality, assuming the cycle starts and ends at the vertex 0
- Denote $\mathrm{TSP}(i, S)$ the least cost to traverse all vertices and return to vertex 0, if the itinerary is currently at vertex i and the traveler has visited all vertices in the set S

- Basic step: $\mathrm{TSP}(i, \mathrm{set\ all\ vertices}) = C_{i,0}$
- Inductive step: $\mathrm{TSP}(i, S) = \min\ _{j \notin S}\ \{\ C_{i,j} + \mathrm{tsp}(j, S \cup \{j\})\ \}$

Traveling salesman problem: Implemnetation

```
const int N = 20;
const int INF = 100000000;
int C[N][N];
int iMem[N][1<<N];
memset(iMem, -1, sizeof(iMem));

int TSP(int i, int S) {
    if (S == ((1 << N) - 1))  return C[i][0];
    if (iMem[i][S] != -1)     return iMem[i][S];

    int res = INF;
    for (int j = 0; j < N; j++) {
        if (S & (1 << j))
            continue;
        res = min(res, C[i][j] + TSP(j, S | (1 << j)));
    }
    iMem[i][S] = res;
    return res;
}
```

Traveling salesman problem: Implemnetation

- We can show the optimal solution by:

```
cout << TSP(0, 1<<0);
```

Traveling salesman problem: The complexity

- There are $n \times 2^n$ possibilities for input
- Each input is processed in $\mathcal{O}(n)$
- The total time is $\mathcal{O}(n^2 \times 2^n)$
- So we can solve the problem in acceptable time for the value of n up to 20

Traveling salesman problem: The complexity

- There are $n \times 2^n$ possibilities for input
- Each input is processed in $\mathcal{O}(n)$
- The total time is $\mathcal{O}(n^2 \times 2^n)$
- So we can solve the problem in acceptable time for the value of n up to 20
- How to accurately give the traveler's itinerary?

Traveling salesman problem: Tracing by recursion

```
void Trace(int i, int S) {
    cout << i << " ";
    if (S == ((1 << N) - 1))  return;

    int res = iMem[i][S];
    for (int j = 0; j < N; j++) {
      if (S & (1 << j))
        continue;
      if (res == C[i][j] + iMem[j][S | (1 << j)]) {
        Trace(j, S | (1 << j));
        break;
      }
    }
}
```

- Call `TSP(0, 1«0);`
- The complexity of tracing function ?

Traveling salesman problem: Tracing by recursion

```
void Trace(int i, int S) {
    cout << i << " ";
    if (S == ((1 << N) - 1))  return;

    int res = iMem[i][S];
    for (int j = 0; j < N; j++) {
      if (S & (1 << j))
        continue;
      if (res == C[i][j] + iMem[j][S | (1 << j)]) {
        Trace(j, S | (1 << j));
        break;
      }
    }
}
```

- Call `TSP(0, 1«0);`
- The complexity of tracing function ? $\mathcal{O}(n^2)$

Traveling salesman problem: Tracing by loop

```
int ans = iMem[0][1];
cout << ans << endl;
stack<int> Stack;
Stack.push(0);
for (int i = 0, S = 1, k = 0; k < n-1; ++k) {
  for (int j = 0; j < n; ++j){
    if (!(S & (1 << j)) &&
      (iMem[i][S] == C[i][j] + iMem[j][S | (1 << j)])) {
         Stack.push(j);
         i = j;
         S = S | (1 << j);
    }
  }
}
while (!Stack.empty()) {
    cout << Stack.back() << " ";
    Stack.pop();
}
```

25 SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attentions!

solct.hust.edu.vn/ fb.com/groups/solct