Programming in C/C++

Patrick Ho
peiqistar@gmail.com

(11/30/2014)

Dynamic Programming (DP)

- A method for solving complex problems by breaking them down into simpler subproblems (optimal substructure).
- Whenever you have a recursive method that takes a certain set of parameters
  - Where the return value of this function is uniquely determined by this set of parameters
  - stores them in memory for later use.
- “Memorize” the return value of the method for that set of parameters using an array (overlapping subproblems)
- Programs can often be speed up by exponential factors using this method
- Usually has recurrent formula
- Now, some examples: since numbers produced by these methods grow exponentially, let’s take everything modulo “MOD”, (a big integer)

memset / memcpy

Coding Tips

- memcpy(void *dest, void *src, int size)
  - Copies “size” bytes from “src” to “dest”
- memset(void *arr, int val, int num)
  - Sets “num” bytes starting from “arr” to “val”

  - Using memset on characters always sets everything to exactly val
    - Characters are one byte each
  - Using memset on doubles should only be done with val = 0, which sets everything to 0
    - Otherwise, just use iteration to fill in values
  - Using memset on integers can be done with:
    - Val = 0 sets everything to 0
    - Val = -1 or 255 sets everything to -1

Fibonacci (naïve)

```
int fib(int i) { // assumes i >= 0
        if (i == 0) return 0;
        if (i == 1) return 1;
        return (fib(i - 1) + fib(i - 2)) % MOD;
}
```

- Very easy to understand
- Incredibly slow
  - Exponential time on i

Fibonacci (recursive)

```
int mem[MAX_NUM];     // remember used number
memset(mem, -1, sizeof(mem));
int fib(int i) { // assumes i >= 0
        if (i == 0) return 0;
        if (i == 1) return 1;
        if (mem[i] != -1) return mem[i]; // used
        return mem[i] = (fib(i - 1) + fib(i - 2)) % MOD;
}
```

- Elegant, understandable
- Risk overflowing stack memory

Fibonacci (iterative)

```
int mem[MAX_NUM];
mem[0] = 0;
mem[1] = 1;
for (int i = 2; i < MAX_NUM; ++i) {
        mem[i] = (mem[i - 1] + mem[i - 2]) % MOD;
}
```

- Faster, no risk of overflowing stack memory
- Becomes less understandable for more complex problems
- Order of iteration matters!
- **When using dynamic programming, a well-reasoned choice must be made between iteration and recursion**

Binomial Coefficient(n choose k)

- Finally there is a formula using factorials that is easy to remember:
  - C(n,k) = n! / (k! * (n-k)!)

- Factorial function use recursion
  - F (n) = n * F(n-1)

- Recursive formula: (Used in DP)
  - C(n, k) = C(n-1, k-1) + C(n-1, k)
  - C(n,0) = 1 for all n >= 0
  - C(0, k) = 0 for all k > 0

Binomial Coefficient

(recursion in 2D)

```
int mem[MAX_NUM][MAX_NUM];
memset(mem, -1, sizeof(mem));
int choose(int n, int k) { // assumes n, k >= 0
        if (k == 0) return 1;
        if (n == 0) return 0;
        if (mem[n][k] != -1) return mem[n][k];
        return mem[n][k] = (choose(n - 1, k - 1) +
                choose(n - 1, k)) % MOD;
}
```

Binomial Coefficient

(iteration in 2D)

```
int mem[MAX_NUM][MAX_NUM];
for (int i = 0; i < MAX_NUM; i++) {
            mem[i][0] = 1;
}
for (int I = 1; I < MAX_NUM; i++) {
            mem[0][i] = 0;
}
for (int i = 1; i < MAX_NUM; i++) {
            for (int j = 1; j < MAX_NUM; j++) {
                        mem[i][j] = (mem[i – 1][j – 1] +
                                    mem[i – 1][j]) % MOD;
            }
}
```

The base case is slightly more complicated in this case

optimAs problems grow more complex, it is generally better to use the recursive version of the recurrence

However, if izations are required (e.g. optimizing an $O(n^2)$ algorithm to an $O(n\log n)$ algorithm), these are generally only feasible through use of iteration

1-0 Knapsack Problem

- Given some items, pack the knapsack to get the maximum total value
- Each item has value/price(Vi) and weight(Wi)
- Each item can only be selected or rejected
- Weight capacity limit: W (sum of Ni*Wi < W)
- **Optimum max value: Max(sum of Ni*Vi)**

Case:	Item 1:	$3	(2 lb)
	Item 2:	$5	(4 lb)
	Item 3:	$4	(3 lb)
	Item 4:	$6	(5 lb)
	W:		(5 lb)

1-0 Knapsack Problem

Input: 1st line is # of items and weight capacity; 2nd line to end is each item id, value and weight

```
4    5
1    3    2
2    5    4
3    4    3
4    6    5
```

Output: 1st line is max value; 2nd has a list of item number

```
7
1,3
```

Knapsack Problem

brute-force approach

- Even in 1-0 Knapsack Problem, which each item can only be selected or rejected, there are *2 power n* combinations of items.
- Too slow for big case

Knapsack Problem

Recursive Formula

$$f(i,w)=\begin{cases} f(i-1,w) & (\text{if } wi > w) \\ \text{Max}[\ vi + f(i-1,w-wi)\ ;\ f(i-1,w)\ ] & (\text{else}) \end{cases}$$

0-1 Knapsack Algorithm

- Using recursion function
  - Advantage ?
  - Limitation ?
- Using iteration
  - Advantage ?
  - Limitation ?
- Demo code (Study10Knapsack)

Knapsack Problem

- Given some items, pack the knapsack to get the maximum total value
- Each item has value/price(Vi) and weight(Wi)
- Final pick number of each item (Ni)
- Weight capacity limit: W (sum of Ni*Wi < W)

- **Optimum max value: Max(sum of Ni*Vi)**

Case:	Item 1:	$5	(3 lb)
	Item 2:	$7	(4 lb)
	Item 3:	$8	(5 lb)
	W:		(10 lb)

Knapsack Problem

Recursive Formula

$$f(i,w)=\begin{cases} f(i-1,w) & (\text{if } wi > w) \\ \text{Max}[\, vi + f(i,w-wi) \,;\, f(i-1,w) \,] & (\text{else}) \end{cases}$$

ONE Item i + optimum combination of weight w-wi

NO Item i + optimum combination items 1 to i-1

Knapsack Problem

Working on Table

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

w

f(i,w)

i

Knapsack Problem

Table

	1	2	3	4	5	6	7	8	9	10
1			Using only item 1							
2			Using only item 1&2							
3			Using only item 1&2&3							

W

i

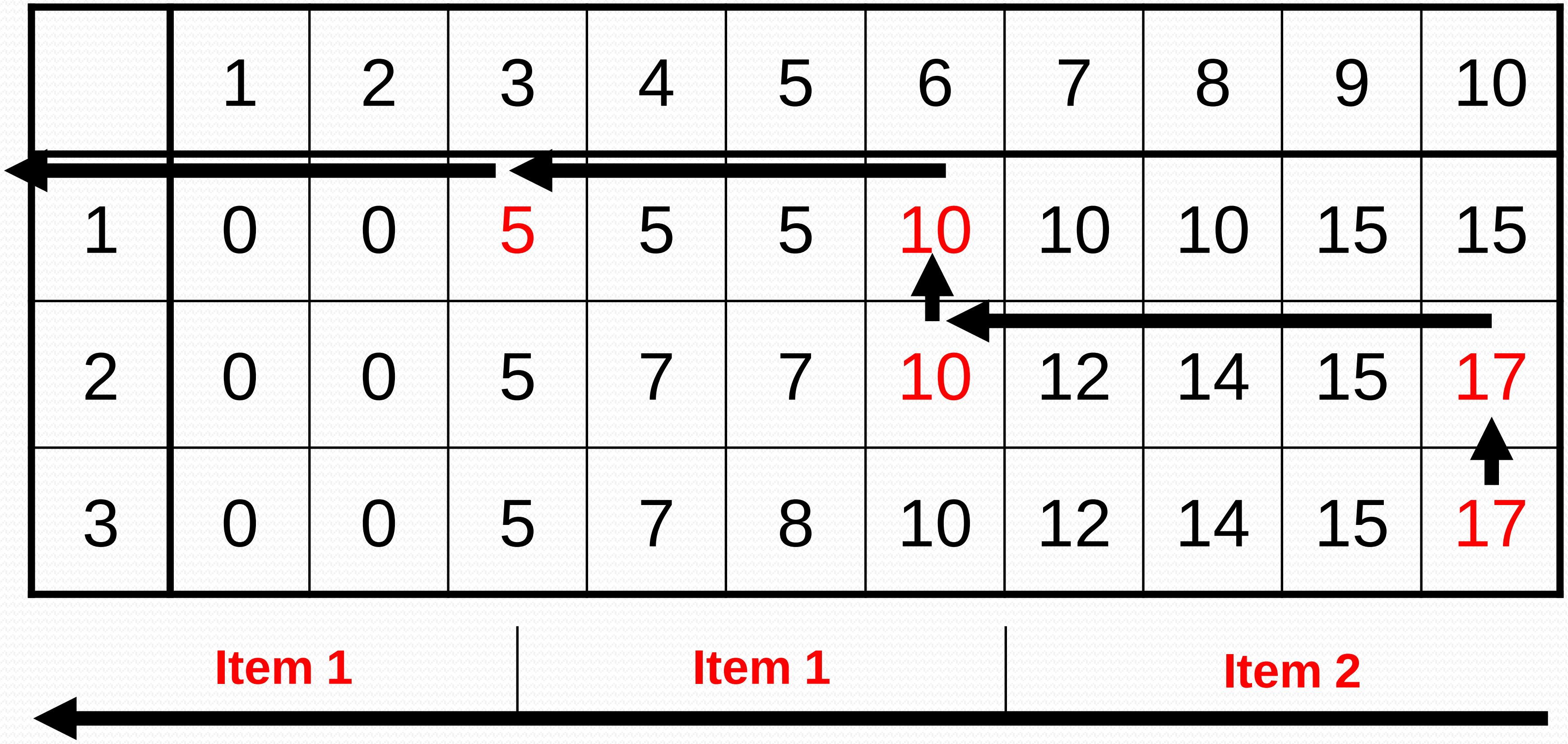
Knapsack Problem

COMPLETED TABLE

	1	2	3	4	5	6	7	8	9	10
1	0	0	5	5	5	10	10	10	15	15
2	0	0	5	7	7	10	12	14	15	17
3	0	0	5	7	8	10	12	14	15	17

Knapsack Problem

Path



	1	2	3	4	5	6	7	8	9	10
1	0	0	5	5	5	10	10	10	15	15
2	0	0	5	7	7	10	12	14	15	17
3	0	0	5	7	8	10	12	14	15	17

Optimal: 2 x Item 1 + 1 x Item 2

Longest Increasing Subsequence (LIS)

- The input is a sequence of numbers(A1,A2,…,An).
- A subsequence is any subset of these numbers taken in order, of the form (Ai1, Ai2, …, Aik).
- An increasing subsequence is one in which the numbers are getting strictly larger.

- The task is to find the increasing subsequence of greatest length.

- e.g.

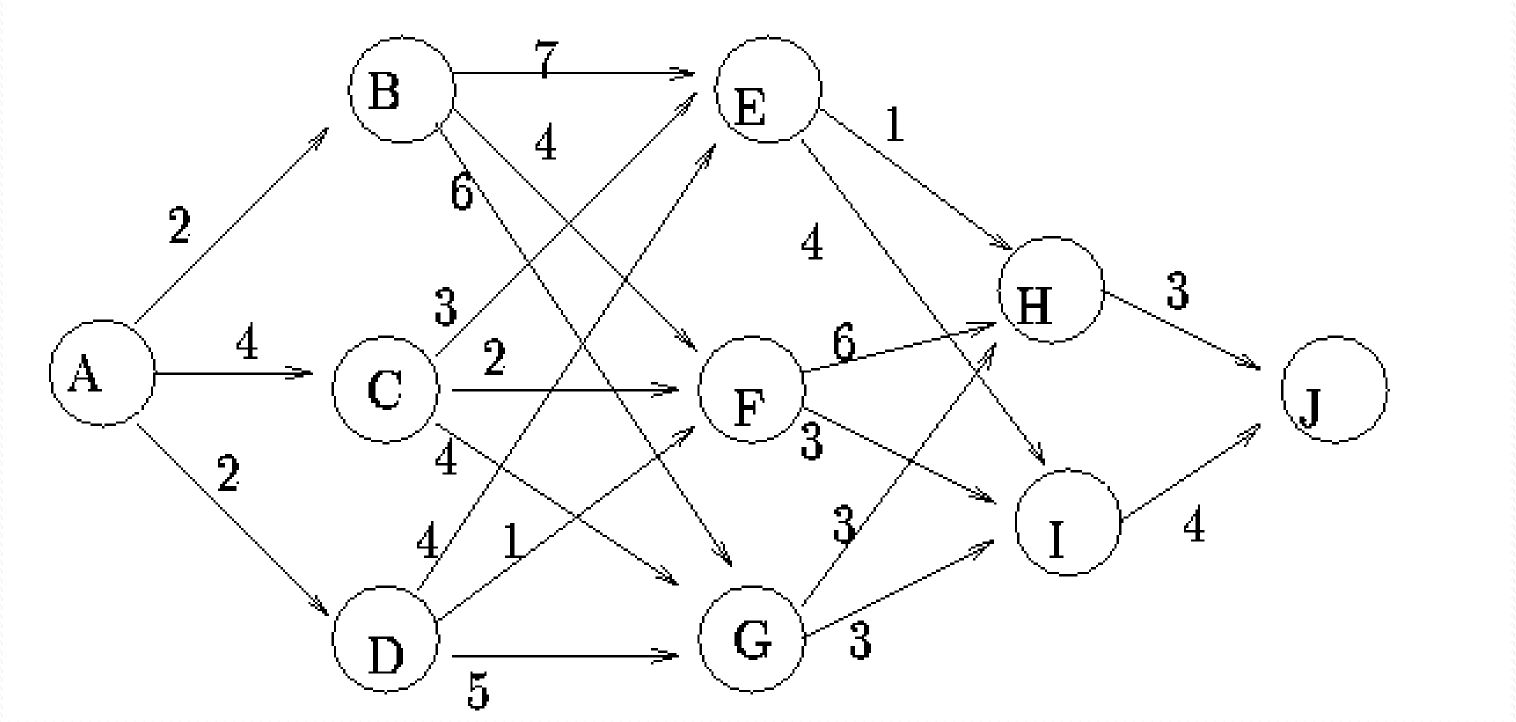
  length of LIS for { 10, 22, 9, 33, 21, 50, 41, 60, 80 } is 6, and LIS is {10, 22, 33, 50, 60, 80}.

Longest Increasing Subsequence (LIS)

- **Optimal Substructure:**
  Let arr[0..n-1] be the input array and L(i) be the length of the LIS till index i such that arr[i] is part of LIS and arr[i] is the last element in LIS, then L(i) can be recursively written as.
  L(i) = { 1 + Max (L(j)) } where j < i and arr[j] < arr[i] and if there is no such j then L(i) = 1
  To get LIS of a given array, we need to return max(L(i)) where 0 < i < n
  So the LIS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.
- See detail,
  - http://www.geeksforgeeks.org/dynamic-programming-set-3-longest-increasing-subsequence/

Shortest Path

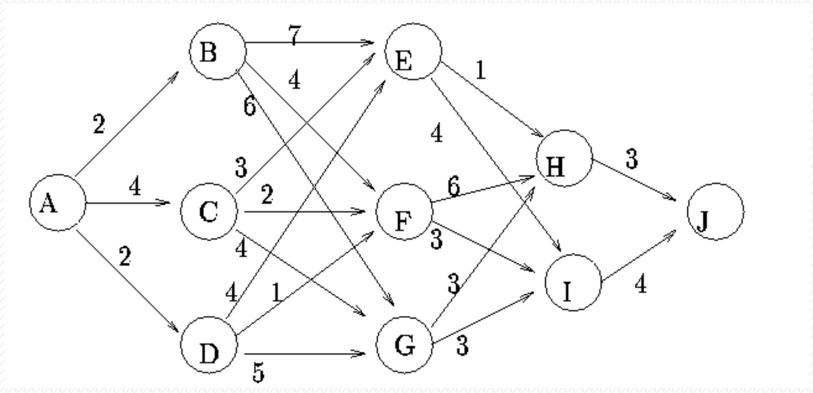
- A special graph with no crossover connection between different stage



Shortest Path

- Input:

```
20
A B 2
A C 4
A D 2
B E 7
B F 4
B G 6
C E 3
C F 2
C G 4
D E 4
D F 1
D G 5
E H 1
E I 4
F H 6
F I 3
G H 3
D I 3
H J 3
I J 4
```



- Output:

```
10
A->D->F->I->J
```

Shortest Path

- Stage 1 contains node A, stage 2 contains nodes B, C, and D, stage 3 contains node E, F, and G, stage 4 contains H and I, and stage 5 contains J.
- The states in each stage correspond just to the node names. So stage 3 contains states E, F, and G.
- If we let *S* denote a node in stage *j*, we can write be the shortest distance from node *S* to the destination *J*:

$$f_j(S) = \min_{\text{nodes } z \text{ in stage } j+1} \{c_{SZ} + f_{j+1}(Z)\}$$

- where Csz denotes the length of arc *SZ*. This gives the recursion needed to solve this problem from target node (J).

Good Training Sites

- codeforces.com
  - virtual contests, solutions, but no difficulty category
- codercharts.com
  - clean difficulty category, but stopped in 2012
- codechef.com
  - difficulty category problems, and someone’s solution
- poj.org
  - more collection problems with discussion
- http://train.usaco.org/usacogate
  - best, clean algorithm category