Dynamic Programming (2)

By: Aminul Islam

Based on Chapter 3 of Foundations of Algorithms

Objectives

- Describe the Dynamic Programming Technique
- Contrast the Divide and Conquer and Dynamic Programming approaches to solving problems
- Identify when dynamic programming should be used to solve a problem

Bionomial Coefficient Problem

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$ where $0 \leq k \leq n$

- This is difficult to calculate for large values of n
- We can eliminate the use of $n!$ using the following formula:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 1 & \text{if } k = 0 \text{ or } k = n \end{cases}$$

Bionomial Coefficient using D&C

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 1 & \text{if } k = 0 \text{ or } k = n \end{cases}$$

```
Problem:  Compute the binomial coefficient
Inputs:  nonnegative integers n and k, where k ≤ n
Outpts:  bin, the binomial coefficient (n k)
int bin (int n, int k)
{
   if (k == 0||n == k)
    return 1;
   else
    return bin(n − 1, k − 1) + bin(n − 1, k);
}
```

- What's the time complexity and order of complexity?
- D&C approach is not efficient. Why?

Bionomial Coefficient using Dynamic Programming (1)

- Establish a recursive property
- Solve an instance in the bottom-up fashion

Bionomial Coefficient using Dynamic Programming (2)

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 1 & \text{if } k = 0 \text{ or } k = n \end{cases}$$

	0	1	2	3	4	j	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6				
i							
n							

Problem: Compute the binomial coefficient

Inputs: nonnegative integers n and k, where $k \leq n$

Outpts: $binD$, the binomial coefficient $\binom{n}{k}$

```
int binD (int n, int k)
{
   index i, j;
   int B[0..n][0..k];
   for (i=0; i≤n; i++)
    for (j=0; j≤min(i,k); j++)
     if (j == 0 || j == i)
       B[i][j] = 1;
     else
       B[i][j] = B[i−1][j−1] + B[i−1][j];
   return B[n][k];
}
```

Order of complexity is

Knapsack problem

Definition: Given items of different values and weights and a knapsack that can carry a fixed weight, find the most valuable set of items that fit in the knapsack.

Formal Definition: There is a knapsack of capacity $W > 0$ and N items. Each item has value $v_i > 0$ and weight $w_i > 0$. Find the selection of items ($\delta_i = 1$ if selected, 0 if not) that fit, $\sum_{i=1}^{N} \delta_i w_i \leq W$, and the total value, $\sum_{i=1}^{N} \delta_i v_i$, is maximized.

There are two versions of the problem:

- Fractional knapsack problem (Items are divisible) and
- 0-1 knapsack problem (Items are indivisible)

Example of Fractional Knapsack problem

item (i)	1	2	3	4
value (v)	100	20	60	40
weight (w)	3	2	4	1
$\frac{v}{w}$	33.3	10	15	40

- Assume Knapsack's max weight capacity is $W = 5$
- How to fill the knapsack with items (or fractions of items) such that the value is maximum?

Algorithm:

- Sort the items by $\frac{\text{value}}{\text{weight}}$ in descending order
- Keep picking the items from this ordered list and if the last item cannot be picked in total, split it up and pick the fraction that fit in the knapsack

items picked=

value= Order of Complexity =

total weight=

Example of 0-1 Knapsack problem

item (i)	1	2	3	4
value (v)	100	20	60	40
weight (w)	3	2	4	1

- Assume Knapsack's max weight capacity is $W = 5$
- How to fill the knapsack with items such that the value is maximum?

items picked=

value=

weight=

Order of complexity=

1	2	3	4	W	V
0	0	0	0	0	0
0	0	0	1	1	40
0	0	1	0	4	60
0	0	1	1	5	100
0	1	0	0	2	20
0	1	0	1	3	60
0	1	1	0	6	80
0	1	1	1	7	120
1	0	0	0	3	100
1	0	0	1	4	140140
1	0	1	0	7	160
1	0	1	1	8	200
1	1	0	0	5	120
1	1	0	1	6	160
1	1	1	0	9	180
1	1	1	1	10	220

* 1 means item is picked and 0 means item is not picked

Here, we went through all possible solutions/cases and picked the one that maximizes the value. This is known as Brute force algorithm. What if $W = 7$?

Optimization Problems

Optimization problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.