Dynamic Programming

- Divide & Conquer
  - Divide a program instance into independent small instances
  - A top-down method
- Dynamic Programming
  - Sometimes it’s hard to divide an instance into independent small instances
  - Some instances may overlap due to inherent program structure
    - A top-down method may introduce redundancies
  - Dynamic programming: a bottom-up method
    - Build up solution from small subinstances
    - Avoid duplicated calculation

Example: Binomial Coefficient

- We want to calculate $\binom{n}{k}$ which can be defined as follows.

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

A recursive algorithm

```
int C(n,k)
{
  if (k==0 || k==n) return 1;
  else return C(n-1,k-1)+C(n-1,k);
}
```

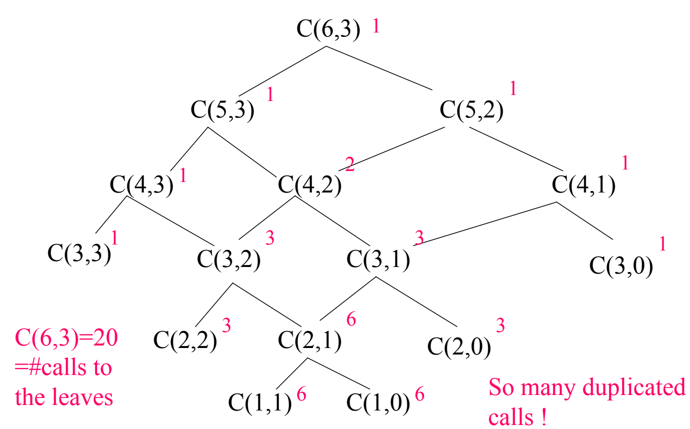
- Cost

$$T(n,k) > T(n-1,k-1) + T(n-1,k)$$

$$T(n,k) \in \Omega(\binom{n}{k})$$

In worst case k=n/2, this is asymptotically $\Omega(2^n / n)$

What’s the problem



A solution using dynamic programming

- We instead calculate bottom-up by filling the following table

n \ k	0	1	2	…….	k-1	k
0	1					
1	1	1				
2	1	2	1			
…						
n-1					C(n-1,k-1)	C(n-1,k)
n						C(n.k)

Cost: time Θ(nk) and space Θ(k)

0-1 Knapsack

- n objects 1, 2, ..., n. Object i has weight w_i and value v_i
- The knapsack can carry a weight not exceeding W.
- Cannot split an object
- Maximize the total value
  - Maximize $\sum_{i=1}^{n} x_i v_i \quad subject \quad to \quad \sum_{i=1}^{n} x_i w_i \le W,$

where v_i , $w_i > 0$ and $x_i \in \{0,1\}$ for $1 <= i <= n$

The greedy algorithm is no longer optimal

object	1	2	3
w_i	6	5	5
v_i	8	5	5

W=10

Key to dynamic programming

- The principle of optimality
  - In an optimal sequence of decisions or choices, each subsequences must be also optimal
- 0-1 knapsack
  - C[i,j] is the maximum value if the weight limit is j and only objects 1 to i are available
  - C[i,j] = max(C[i-1,j], C[i-1, j-w_i]+v_i);

Dynamic programming

- Set up a table C[0..n, 0..W] with one row for each available object and one column for each weight from 0 to W. Specifically, C[0, j] = 0 for all j.
- C[i,j] is the maximum value if the weight limit is j and only objects 1 to i are available
  - $C[i,j] = \max(C[i-1,j], C[i-1, j-w_i]+v_i);$
- C[n,W] will be the solution

Example

Weight limit	0	1	2	3	4	5	6	7	8	9	10	11
$w_1=1$ $v_1=1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2=2$ $v_2=6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3=5$ $v_3=18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4=6$ $v_4=22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5=7$ $v_5=28$	0	1	6	7	7	18	22	28	29	34	25	40

Algorithm

```
Knapsack0-1(v, w, n, W)
{
   for (w = 0; w<=W; w++) {
      c[0,w] = 0;
   }
   for (i=1;  i <=n; i++) {
       c[i,0] = 0
      for (w=1;  w<=W; w++) {
         if (w[i] <  w) {
            if  (c[i-1,w-w[i]]+v[i] > c[i-1,w])
                c[i,w] = c[i-1,w-w[i]] + v[i];
            else  c[i,w] = c[i-1,w]
         } else  c[i,w] = c[i-1,w]
      } // for w
   } for i
}
```

The run time performance of this algorithm is $\Theta(nW)$

Finding the objects

```
i=n;
k=W;
while (i>0 && k>0) {
  if (C[i,k] <> C[i-1,k]) {
       mark the i-th object as in knapsack;
       i = i-1;
       k = k-w[i];
  } else
      i = i-1;
}
```

Cost: O(n+W)

Example

Weight limit	0	1	2	3	4	5	6	7	8	9	10	11
$w_1=1$ $v_1=1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2=2$ $v_2=6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3=5$ $v_3=18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4=6$ $v_4=22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5=7$ $v_5=28$	0	1	6	7	7	18	22	28	29	34	25	40

=

Example

Weight limit	0	1	2	3	4	5	6	7	8	9	10	11
$w_1=1$ $v_1=1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_1=2$ $v_1=6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_1=5$ $v_1=18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_1=6$ $v_1=22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_1=7$ $v_1=28$	0	1	6	7	7	18	22	28	29	34	25	40

≠

Example

Weight limit	0	1	2	3	4	5	6	7	8	9	10	11
$w_1=1$ $v_1=1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2=2$ $v_2=6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3=5$ $v_3=18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4=6$ $v_4=22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5=7$ $v_5=28$	0	1	6	7	7	18	22	28	29	34	25	40

≠