Discrete Optimization

The Knapsack Problem: Dynamic Programming

Goals of the Lecture

- How to find the **BEST** knapsack solutions
- Using dynamic programming

Dynamic Programming

- Widely used optimization technique
  - for certain classes of problems
  - heavily used in computational biology

- Basic principle
  - divide and conquer
  - bottom up computation

Dynamic Programming

- Basic conventions and notations
  - assume that `I` = {1,2,...,n}
  - O(k,j) denotes the optimal solution to the knapsack problem with capacity k and items [1..j]

$$
\begin{aligned}
&\text{maximize} && \textstyle\sum_{i \in 1..j} v_i \; x_i \\
&\text{subject to} && \\
& && \textstyle\sum_{i \in 1..j} w_i x_i \leq k \\
& && x_i \in \{0,1\} \quad (i \in 1..j)
\end{aligned}
$$

- We are interested in finding out the best value O(K,n)

Recurrence Relations (Bellman Equations)

- Assume that we know how to solve
  - $O(k,j-1)$ for all k in 0..K
- We want to solve $O(k,j)$
  - We are just considering one more item, i.e., item j.
- If $w_j \leq k$, there are two cases
  - Either we do not select item j, then the best solution we can obtain is $O(k,j-1)$
  - Or we select item j and the best solution is $v_j + O(k-w_j,j-1)$
- In summary
  - $O(k,j) = \max(O(k,j-1), v_j + O(k-w_j,j-1))$ if $w_j \leq k$
  - $O(k,j) = O(k,j-1)$ otherwise
- Of course
  - $O(k,0) = 0$ for all k

Recurrence Relations

- We can write a simple program

```
int O(int k,int j) {
   if (j == 0)
     return 0;
   else if (wj <= k)
     return max(O(k,j-1),vj + O(k-wj,j-1));
   else
     return O(k,j-1)
}
```

- How efficient is this approach?

Recurrence Relations - Fibonacci Numbers

- We can write a simple program for finding fibonacci numbers

```
int fib(int n) {
   if (n == 0 || n == 1)
      return 1;
   else
         return fib(n-2) + fib(n-1);
}
```

- How efficient is this approach?
  - we are solving many times the same subproblem
    - fib(n-1) requires fib(n-2) which we have already solved
    - fib(n-3) requires fib(n-4) which we have already solved

Dynamic Programming

- Compute the recursive equations bottom up
  - start with zero items
  - continue with one item
  - then two items
  - ...
  - then all items

$$\begin{array}{ll}
\text{maximize} & 5x_1 + 6x_2 + 3x_3 \\
\text{subject to} & \\
& 4x_1 + 5x_2 + 2x_3 \leq 9 \\
& x_i \in \{0, 1\} \quad (i \in 1..3)
\end{array}$$

Dynamic Programming - Example

- How to find which items to select?

Capacity	0
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Take items 1 and 2

Trace back

Dynamic Programming - Example

$$
\begin{array}{ll}
\text{maximize} & 16x_1 + 19x_2 + 23x_3 + 28x_4 \\
\text{subject to} & 2x_1 + 3x_2 + 4x_3 + 5x_4 \le 7 \\
 & x_i \in \{0,1\} \quad (i \in 1..4)
\end{array}
$$

$$x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1$$

Capacity	0	1	2	3	4
0					
1					
2					
3					
4					
5					
6					
7					

Dynamic Programming

- What is the complexity of this algorithm?
  - time to fill the table
  - i.e., O(K n)

- Is this polynomial?
  - How many bits does K need to be represented on a computer?
    - log(K) bits
  - Hence the algorithm is in fact exponential in terms of the input size
    - pseudo-polynomial algorithm
    - “efficient” when K is small

Until Next Time