Dynamic Programming

- Sequence of decisions.
- Problem state.
- Principle of optimality.

Sequence Of Decisions

- As in the greedy method, the solution to a problem is viewed as the result of a sequence of decisions.
- Unlike the greedy method, decisions are not made in a greedy and binding manner.

0/1 Knapsack Problem

(section 15.2.1, p.715 of Text)

Let $x_i = 1$ when item i is selected and let $x_i = 0$ when item i is not selected.

$$\text{maximize} \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq c$$

$$\text{and } x_i = 0 \text{ or } 1 \text{ for all } i$$

All profits and weights are positive.

Sequence Of Decisions

- Decide the x_i values in the order $x_1, x_2, x_3, \ldots, x_n$
- Decide the x_i values in the order $x_n, x_{n-1}, x_{n-2}, \ldots, x_1$
- Decide the x_i values in the order $x_1, x_n, x_2, x_{n-1}, \ldots$
- Or any other order.

Problem State

- The state of the 0/1 knapsack problem is given by
  - the weights and profits of the available items
  - the capacity of the knapsack
- When a decision on one of the x_i values is made, the problem state changes.
  - item i is no longer available
  - the remaining knapsack capacity may be less

Problem State

- Suppose that decisions are made in the order $x_1, x_2, x_3, \ldots, x_n$.
- The initial state of the problem is described by the pair (1, c).
  - Items 1 through n are available (the weights, profits and n are implicit).
  - The available knapsack capacity is c.
- Following the first decision the state becomes one of the following:
  - (2, c) … when the decision is to set $x_1 = 0$.
  - (2, c-w_1) … when the decision is to set $x_1 = 1$.

Principle of Optimality

- An optimal solution satisfies the following property:
  - No matter what the first decision is, the remaining decisions are optimal with respect to the state that results from this decision.
- Dynamic programming may be used only when the principle of optimality holds.

0/1 Knapsack Problem

- Suppose that decisions are made in the order x_1, x_2, x_3, …, x_n.
- Let $x_1 = a_1$, $x_2 = a_2$, $x_3 = a_3$, …, $x_n = a_n$ be an optimal solution.
- If $a_1 = 0$, then following the first decision the state is (2, c).
- a_2, a_3, …, a_n must be an optimal solution to the knapsack instance given by the state (2,c).

$x_1 = a_1 = 0$

$$\text{maximize} \sum_{i=2}^{n} p_i x_i$$

$$\text{subject to} \sum_{i=2}^{n} w_i x_i \leq c$$

and $x_i = 0$ or 1 for all i

- If not, this instance has a better solution b_2, b_3, …, b_n.

$$\sum_{i=2}^{n} p_i b_i > \sum_{i=2}^{n} p_i a_i$$

$x_1 = a_1 = 0$

- $x_1 = a_1, x_2 = b_2, x_3 = b_3, \ldots, x_n = b_n$ is a better solution to the original instance than is $x_1 = a_1, x_2 = a_2, x_3 = a_3, \ldots, x_n = a_n$.
- So $x_1 = a_1, x_2 = a_2, x_3 = a_3, \ldots, x_n = a_n$ cannot be an optimal solution … a contradiction with the assumption that it is optimal.

$x_1 = a_1 = 1$

- Next, consider the case $a_1 = 1$. Following the first decision the state is $(2, c\text{-}w_1)$.
- $a_2, a_3, \ldots, a_n$ must be an optimal solution to the knapsack instance given by the state $(2, c - w_1)$.

$x_1 = a_1 = 1$

$$\text{maximize} \sum_{i=2}^{n} p_i x_i$$

$$\text{subject to} \sum_{i=2}^{n} w_i x_i \le (c - w_1)$$

and $x_i = 0$ or 1 for all i

- If not, this instance has a better solution $b_2, b_3, \ldots, b_n$.

$$\sum_{i=2}^{n} p_i b_i > \sum_{i=2}^{n} p_i a_i$$

$x_1 = a_1 = 1$

- $x_1 = a_1, x_2 = b_2, x_3 = b_3, \ldots, x_n = b_n$ is a better solution to the original instance than is $x_1 = a_1, x_2 = a_2, x_3 = a_3, \ldots, x_n = a_n$.
- So $x_1 = a_1, x_2 = a_2, x_3 = a_3, \ldots, x_n = a_n$ cannot be an optimal solution … a contradiction with the assumption that it is optimal.

0/1 Knapsack Problem

- Therefore, no matter what the first decision is, the remaining decisions are optimal with respect to the state that results from this decision.
- The principle of optimality holds and dynamic programming may be applied.

Dynamic Programming Recurrence

- Let f(i,y) be the profit value of the optimal solution to the knapsack instance defined by the state (i,y).
  - Items i through n are available.
  - Available capacity is y.
- For the time being assume that we wish to determine only the value of the best solution.
  - Later we will worry about determining the x_is that yield this maximum value.
- Under this assumption, our task is to determine f(1,c).

Dynamic Programming Recurrence

- f(n,y) is the value of the optimal solution to the knapsack instance defined by the state (n,y).
  - Only item n is available.
  - Available capacity is y.
- If $w_n \leq y$, $f(n,y) = p_n$.
- If $w_n > y$, $f(n,y) = 0$.

Dynamic Programming Recurrence

- Suppose that $i < n$.
- f(i,y) is the value of the optimal solution to the knapsack instance defined by the state (i,y).
  - Items i through n are available.
  - Available capacity is y.
- Suppose that in the optimal solution for the state (i,y), the first decision is to set $x_i = 0$.
- From the principle of optimality (we have shown that this principle holds for the knapsack problem), it follows that $f(i,y) = f(i+1,y)$.

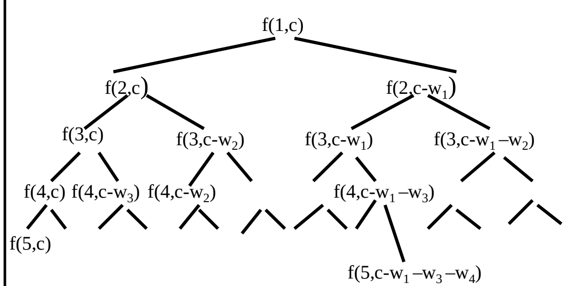
Dynamic Programming Recurrence

- The only other possibility for the first decision is $x_i = 1$.
- The case $x_i = 1$ can arise only when $y \geq w_i$.
- From the principle of optimality, it follows that $f(i,y) = f(i+1,y-w_i) + p_i$.
- Combining the two cases, we get
  - $f(i,y) = f(i+1,y)$ whenever $y < w_i$.
  - $f(i,y) = \max\{f(i+1,y), f(i+1,y-w_i) + p_i\}$, $y \geq w_i$.

Recursive Code

```
/** @return f(i,y) */
private static int f(int i, int y)
{
    if (i == n) return (y < w[n]) ? 0 : p[n];
    if (y < w[i]) return f(i + 1, y);
    return Math.max(f(i + 1, y),
                    f(i + 1, y - w[i]) + p[i]);
}
```

Recursion Tree



Time Complexity

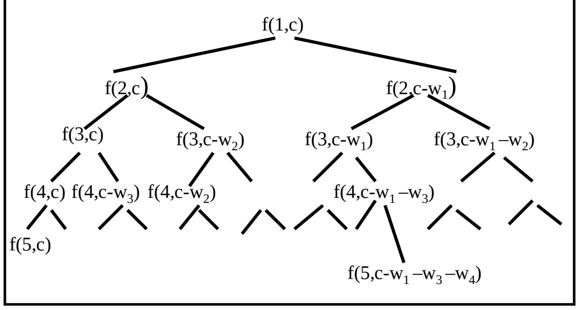
- Let t(n) be the time required when n items are available.
- $t(0) = t(1) = a$, where a is a constant.
- When $t > 1$,

  $t(n) \leq 2t(n-1) + b$,

  where b is a constant.
- $t(n) = O(2^n)$.

Solving dynamic programming recurrences recursively can be hazardous to run time.

Reducing Run Time



Time Complexity

- Level i of the recursion tree has up to 2^{i-1} nodes.
- At each such node an f(i,y) is computed.
- Several nodes may compute the same f(i,y).
- We can save time by not recomputing already computed f(i,y)s.
- Save computed f(i,y)s in a dictionary.
  - Key is (i, y) value.
  - f(i, y) is computed recursively only when (i,y) is not in the dictionary.
  - Otherwise, the dictionary value is used.

Integer Weights

- Assume that each weight is an integer.
- The knapsack capacity c may also be assumed to be an integer.
- Only f(i,y)s with $1 \le i \le n$ and $0 \le y \le c$ are of interest.
- Even though level i of the recursion tree has up to 2^{i-1} nodes, at most c+1 represent different f(i,y)s.

Integer Weights Dictionary

- Use an array fArray[][] as the dictionary.
- fArray[1:n][0:c]
- fArray[i][y] = -1 iff f(i,y) not yet computed.
- This initialization is done before the recursive method is invoked.
- The initialization takes O(cn) time.

No Recomputation Code

```
private static int f(int i, int y)
{
    if (fArray[i][y] ≥ 0) return fArray[i][y];
    if (i == n) {fArray[i][y] = (y < w[n]) ? 0 : p[n];
                 return fArray[i][y];}
    if (y < w[i]) fArray[i][y] = f(i + 1, y);
    else fArray[i][y] = Math.max(f(i + 1, y),
                               f(i + 1, y - w[i]) + p[i]);
    return fArray[i][y];
}
```

Time Complexity

- t(n) = O(cn).
- Good when cn is small relative to 2^n.
- n = 3, c = 1010101
  w = [100102, 1000321, 6327]
  p = [102, 505, 5]
- 2^n = 8
- cn = 3030303