Dynamic Programming

- Sequence of decisions.
- Problem state.
- Principle of optimality.
- Dynamic Programming Recurrence Equations.
- Solution of recurrence equations.

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

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

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

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

and $x_i = 0$ or 1 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, …, x_n.
- Decide the x_i values in the order x_n, x_{n-1}, x_{n-2}, …, x_1.
- Decide the x_i values in the order x_1, x_n, x_2, x_{n-1}, …
- 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, …, 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 $x_1 = 1$.

Problem State

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

Principle Of Optimality

- An optimal solution satisfies the following property:
  - No matter what the first decision, 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 $(2,c)$.

$$x_1 = a_1 = 0$$

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

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

and $x_i = 0$ or 1 for all i

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

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

$$x_1 = a_1 = 1$$

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

subject to $\sum_{i=2}^{n} w_i x_i <= c - w_1$

and $x_i = 0$ or 1 for all i

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

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

0/1 Knapsack Problem

- Therefore, no matter what the first decision, 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 $w_n <= y$, $f(n,y) = p_n$.
- 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 $f(i,y) = f(i+1,y)$.

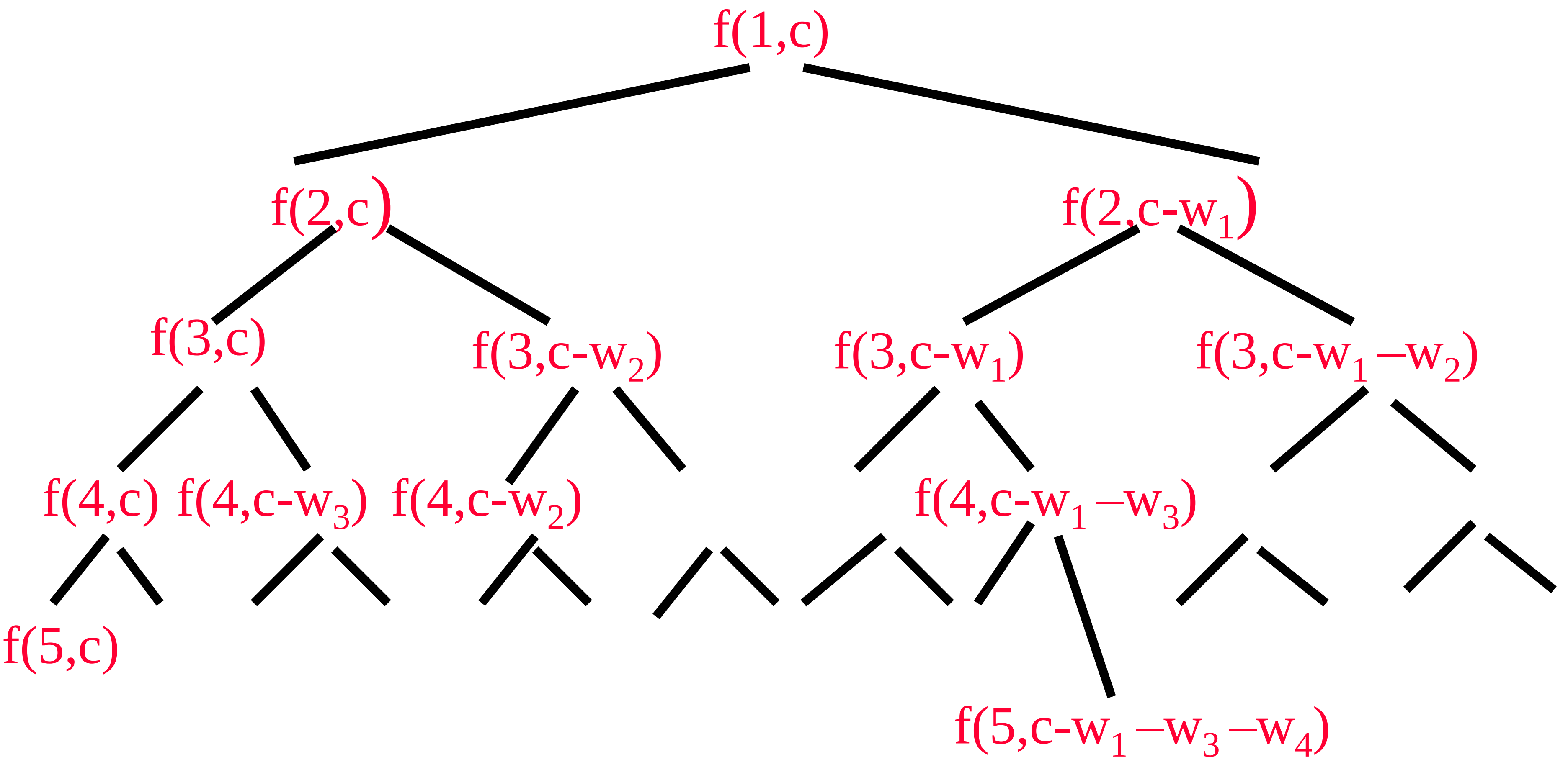
Dynamic Programming Recurrence

- The only other possibility for the first decision is $x_i = 1$.
- The case $x_i = 1$ can arise only when $y >= w_i$.
- 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 $y < w_i$.
  - $f(i,y) = \max\{f(i+1,y), f(i+1,y-w_i) + p_i\}$, $y >= w_i$.

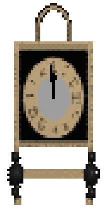
Recursive Code

```
/** @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

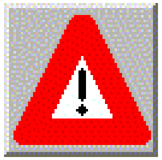


- Let $t(n)$ be the time required when n items are available.
- $t(0) = t(1) = a$, where a is a constant.
- When $t > 1$,

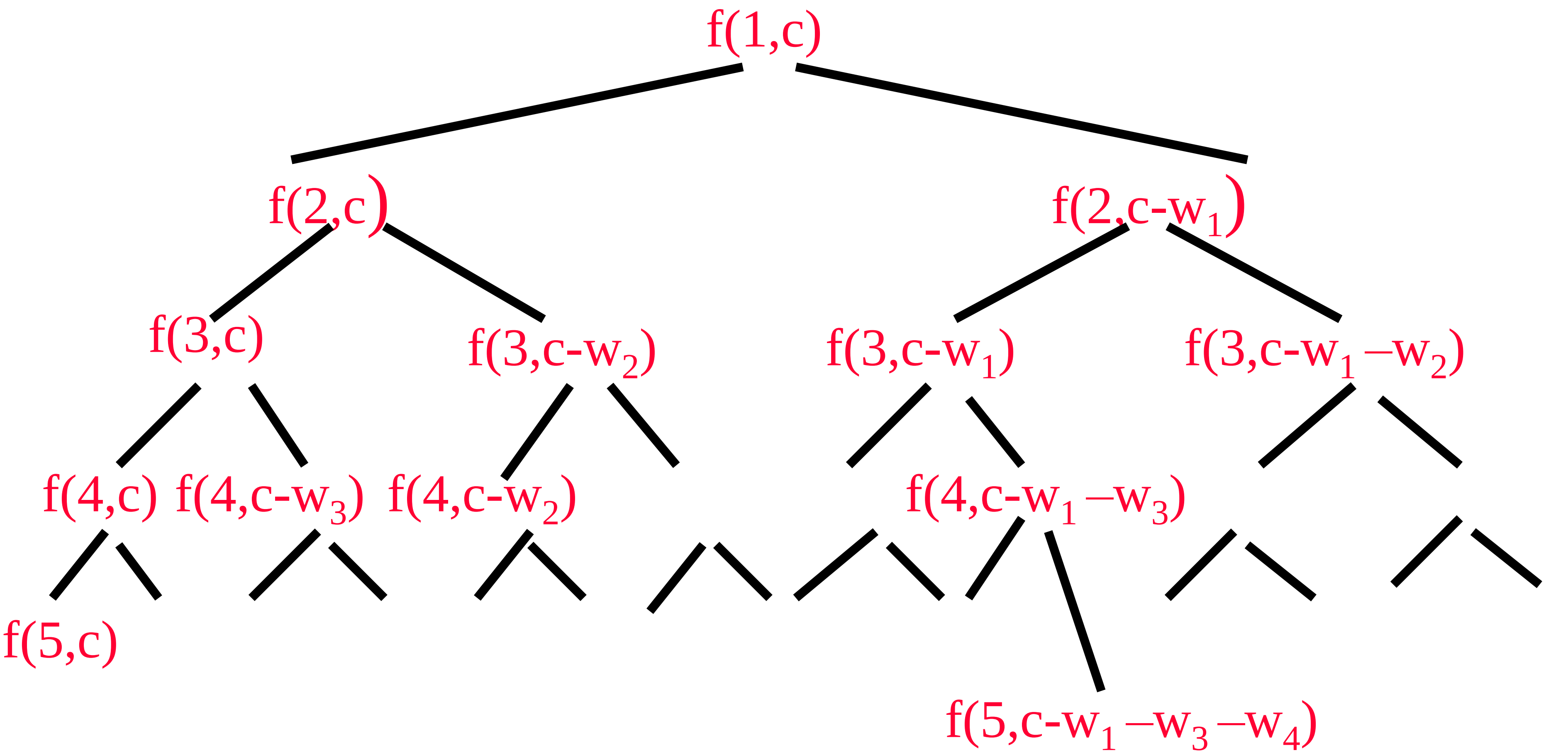
  $t(n) <= 2t(n-1) + b$,

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

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



Reducing Run Time



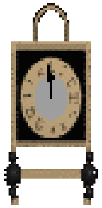
Integer Weights Dictionary

- 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



- $t(n) = O(cn)$.
- Analysis done in text.
- Good when cn is small relative to 2^n.
- $n = 3$, $c = 1010101$

  $w = [100102, 1000321, 6327]$

  $p = [102, 505, 5]$
- $2^n = 8$
- $cn = 3030303$

Contd…

Let $f_j(y)$ be the value of an optimal solution to KNAP$(1, j, y)$. Since the principle of optimality holds, we obtain

$$f_n(m) = \max\ \{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\} \tag{5.14}$$

For arbitrary $f_i(y)$, $i > 0$, Equation 5.14 generalizes to

$$f_i(y) = \max\ \{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\} \tag{5.15}$$

Equation 5.15 can be solved for $f_n(m)$ by beginning with the knowledge $f_0(y) = 0$ for all y and $f_i(y) = -\infty, y < 0$. Then $f_1, f_2, \ldots, f_n$ can be successively computed using (5.15).

ordered set $S^i = \{(f(y_j), y_j) | 1 \leq j \leq k\}$ to represent $f_i(y)$. Each member of S^i is a pair (P, W), where $P = f_i(y_j)$ and $W = y_j$. Notice that $S^0 = \{(0,0)\}$. We can compute S^{i+1} from S^i by first computing

$$S_1^i = \{(P, W) | (P - p_i, W - w_i) \in S^i\} \tag{5.16}$$

Now, S^{i+1} can be computed by merging the pairs in S^i and S_1^i together. Note that if S^{i+1} contains two pairs (P_j, W_j) and (P_k, W_k) with the property that $P_j \leq P_k$ and $W_j \geq W_k$, then the pair (P_j, W_j) can be discarded because of (5.15). Discarding or purging rules such as this one are also known as *dominance rules*. Dominated tuples get purged. In the above, (P_k, W_k) dominates (P_j, W_j).

Example

Example 5.21 Consider the knapsack instance $n = 3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 2, 5)$, and $m = 6$. For these data we have

$$
\begin{aligned}
S^0 &= \{(0,0)\}; S_1^0 = \{(1,2)\} \\
S^1 &= \{(0,0),(1,2)\}; S_1^1 = \{(2,3),(3,5)\} \\
S^2 &= \{(0,0),(1,2),(2,3),(3,5)\}; S_1^2 = \{(5,4),(6,6),(7,7),(8,9)\} \\
S^3 &= \{(0,0),(1,2),(2,3),(5,4),(6,6),(7,7),(8,9)\}
\end{aligned}
$$

If $(P1, W1)$ is the last tuple in S^n, a set of 0/1 values for the x_i's such that $\sum p_i x_i = P1$ and $\sum w_i x_i = W1$ can be determined by carrying out a search through the S^is. We can set $x_n = 0$ if $(P1, W1) \in S^{n-1}$. If $(P1, W1) \notin S^{n-1}$, then $(P1 - p_n, W1 - w_n) \in S^{n-1}$ and we can set $x_n = 1$. This leaves us to determine how either $(P1, W1)$ or $(P1 - p_n, W1 - w_n)$ was obtained in S^{n-1}. This can be done recursively.

Example 5.22 With $m = 6$, the value of $f_3(6)$ is given by the tuple (6, 6) in S^3 (Example 5.21). The tuple (6, 6) $\notin S^2$, and so we must set $x_3 = 1$. The pair (6, 6) came from the pair $(6 - p_3, 6 - w_3) = (1, 2)$. Hence (1, 2) $\in S^2$. Since $(1, 2) \in S^1$, we can set $x_2 = 0$. Since (1, 2) $\notin S^0$, we obtain $x_1 = 1$. Hence an optimal solution is $(x_1, x_2, x_3) = (1, 0, 1)$. □

Algorithm $\mathsf{DKP}(p, w, n, m)$
{
    $S^0 := \{(0, 0)\}$;
    for $i := 1$ **to** $n - 1$ **do**
    {
        $S_1^{i-1} := \{(P, W) | (P - p_i, W - w_i) \in S^{i-1}$ and $W \leq m\}$;
        $S^i := \mathsf{MergePurge}(S^{i-1}, S_1^{i-1})$;
    }
    (PX, WX) :=last pair in S^{n-1};
    $(PY, WY) := (P' + p_n, W' + w_n)$ where W' is the largest W in
        any pair in S^{n-1} such that $W + w_n \leq m$;
    // Trace back for $x_n, x_{n-1}, \ldots, x_1$.
    if $(PX > PY)$ **then** $x_n := 0$;
    else $x_n := 1$;
    $\mathsf{TraceBackFor}(x_{n-1}, \ldots, x_1)$;
}

Algorithm 5.6 Informal knapsack algorithm

Example

1. Generate the sets S^i, $0 \leq i \leq 4$ (Equation 5.16), when $(w_1, w_2, w_3, w_4) = (10, 15, 6, 9)$ and $(p_1, p_2, p_3, p_4) = (2, 5, 8, 1)$.