Median
Selecting k-th smallest

Problem: Given numbers $x_1, \ldots, x_n$ and an integer $1 \leq k \leq n$
output the $k$-th smallest number
\[ \text{Sel}([x_1, \ldots, x_n], k) \]

A simple algorithm: Sort the numbers in time $O(n \log n)$ then return the $k$-th smallest in the array.

Can we do better?

Yes, in time $O(n)$ if $k = 1$ or $k = 2$.

Can we do $O(n)$ for all possible values of $k$?
An Idea

Choose a number $w$ from $x_1, \ldots, x_n$

Define

- $S_< (w) = \{ x_i : x_i < w \}$
- $S_= (w) = \{ x_i : x_i = w \}$
- $S_> (w) = \{ x_i : x_i > w \}$

Solve the problem recursively as follows:

- If $k \leq |S_< (w)|$, output $Sel(S_< (w), k)$
- Else if $k \leq |S_< (w)| + |S_= (w)|$, output $w$
- Else output $Sel(S_> (w), k - |S_< (w)| - |S_= (w)|)$

Ideally want $|S_< (w)|, |S_> (w)| \leq n/2$. In this case ALG runs in $O(n) + O \left( \frac{n}{2} \right) + O \left( \frac{n}{4} \right) + \cdots + O(1) = O(n)$. 

Can be computed in linear time
How to choose w?

Suppose we choose w uniformly at random similar to the pivot in quicksort. Then, \( \mathbb{E}[|S_<(w)|] = \mathbb{E}[|S_>(w)|] = n/2 \). Algorithm runs in \( O(n) \) in expectation. Can we get \( O(n) \) running time deterministically?

- Partition numbers into sets of size 3.
- Sort each set (takes \( O(n) \))
- \( w = \text{Sel(midpoints, } n/6) \)
Assume all numbers are distinct for simplicity.

How to lower bound $|S_<(w)|, |S_>(w)|$?

- $|S_<(w)| \geq 2 \left( \frac{n}{6} \right) = \frac{n}{3}$
- $|S_>(w)| \geq 2 \left( \frac{n}{6} \right) = \frac{n}{3}$.

So, what is the running time?
Assume all numbers are distinct for simplicity.

**Asymptotic Running Time?**

- If $k \leq |S_{<}(w)|$, output $Sel(S_{<}(w), k)$
- Else if $k \leq |S_{<}(w)| + |S_{\leq}(w)|$, output $w$
- Else output $Sel(S_{>}(w), k - |S_{<}(w)| - |S_{\leq}(w)|)$

Where $\frac{n}{3} \leq |S_{<}(w)|, |S_{>}(w)| \leq \frac{2n}{3}$

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n) \Rightarrow T(n) = O(n \log n)$$
An Improved Idea

Partition into \( \frac{n}{5} \) sets. Sort each set and set \( w = Sel(midpoints, \frac{n}{10}) \)

- \( |S_{<}(w)| \geq 3 \left( \frac{n}{10} \right) = \frac{3n}{10} \)
- \( |S_{>}(w)| \geq 3 \left( \frac{n}{10} \right) = \frac{3n}{10} \)

\[
T(n) = T \left( \frac{n}{5} \right) + T \left( \frac{7n}{10} \right) + O(n) \Rightarrow T(n) = O(n)
\]
An Improved Idea

Sel(S, k) {
    n ← |S|
    If (n < ??) return ??
    Partition S into n/5 sets of size 5
    Sort each set of size 5 and let M be the set of medians, so |M|=n/5
    Let w=Sel(M,n/10)
    For i=1 to n{
        If x_i < w add x to S_(<w)
        If x_i > w add x to S_(>w)
        If x_i = w add x to S_(=w)
    }
    If (k ≤ |S_(<w)|)
        return Sel(S_(<w),k)
    else if (k ≤ |S_(<w)| + |S_(=w)|)
        return w;
    else
        return Sel(S_(>w),k − |S_(<w)| − |S_(=w)|)
}
Weighted Interval Scheduling
Interval Scheduling

- Job $j$ starts at $s(j)$ and finishes at $f(j)$ and has weight $w_j$
- Two jobs compatible if they don’t overlap.
- Goal: find maximum weight subset of mutually compatible jobs.

![Diagram of intervals]

Time: 0 1 2 3 4 5 6 7 8 9 10 11
Unweighted Interval Scheduling: Review

Recall: Greedy algorithm works if all weights are 1:
- Consider jobs in ascending order of finishing time
- Add job to a subset if it is compatible with prev added jobs.

Observation: Greedy ALG fails spectacularly if arbitrary weights are allowed:

[Diagram showing two schedules with different weights, illustrating the failure of the greedy algorithm.]
Weighted Job Scheduling by Induction

Suppose 1, ..., n are all jobs. Let us use induction:

IH: Suppose we can compute the optimum job scheduling for < n jobs.

IS: Goal: For any n jobs we can compute OPT.

Case 1: Job n is not in OPT.
-- Then, just return OPT of 1, ..., n - 1.

Case 2: Job n is in OPT.
-- Then, delete all jobs not compatible with n and recurse.

Q: Are we done?
A: No, How many subproblems are there? Potentially $2^n$ all possible subsets of jobs.

This idea works for any Optimization problem.

For NP-hard problems there is no ordering to reduce # subproblems.
**Sorting to Reduce Subproblems**

**Sorting Idea:** Label jobs by finishing time \( f(1) \leq \cdots \leq f(n) \)

**IS:** For jobs 1, ..., \( n \) we want to compute OPT

**Case 1:** Suppose OPT has job \( n \).
- So, all jobs \( i \) that are not compatible with \( n \) are not OPT
- Let \( p(n) \) = largest index \( i < n \) such that job \( i \) is compatible with \( n \).
- Then, we just need to find OPT of 1, ..., \( p(n) \)
Sorting Idea: Label jobs by finishing time $f(1) \leq \cdots \leq f(n)$

IS: For jobs 1, ..., $n$ we want to compute OPT

Case 1: Suppose OPT has job $n$.
- So, all jobs $i$ that are not compatible with $n$ are not OPT
- Let $p(n) =$ largest index $i < n$ such that job $i$ is compatible with $n$.
- Then, we just need to find OPT of 1, ..., $p(n)$

Take best of the two

Case 2: OPT does not select job $n$.
- Then, OPT is just the OPT of 1, ..., $n - 1$

Q: Have we made any progress (still reducing to two subproblems)?
A: Yes! This time every subproblem is of the form 1, ..., $i$ for some $i$
So, at most $n$ possible subproblems.
Weighted Job Scheduling by Induction

**Sorting Idea:** Label jobs by finishing time \( f(1) \leq \cdots \leq f(n) \)
Def \( OPT(j) \) denote the weight of OPT solution of \( 1, \ldots, j \)

To solve \( OPT(j) \):

**Case 1:** \( OPT(j) \) has job \( j \).
- So, all jobs \( i \) that are not compatible with \( j \) are not \( OPT(j) \).
- Let \( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).
- So \( OPT(j) = OPT(p(j)) + w_j \).

**Case 2:** \( OPT(j) \) does not select job \( j \).
- Then, \( OPT(j) = OPT(j - 1) \).

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left( w_j + OPT(p(j)), OPT(j - 1) \right) & \text{o. w.}
\end{cases}
\]
Input: \( n, \, s(1), \ldots, s(n) \) and \( f(1), \ldots, f(n) \) and \( w_1, \ldots, w_n \).

Sort jobs by finish times so that \( f(1) \leq f(2) \leq \cdots f(n) \).

Compute \( p(1), p(2), \ldots, p(n) \)

\[
OPT(j) \{ \\
\quad \text{if } ( \, j = 0 \, ) \text{ return 0} \\
\quad \text{else} \text{ return } \max (w_j + \, OPT(p(j)), \, OPT(j - 1)) .
\}
\]
Recursive Algorithm Fails

Even though we have only $n$ subproblems, we do not store the solution to the subproblems

- So, we may re-solve the same problem many many times.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence

\[ p(1) = 0, p(j) = j - 2 \]
Algorithm with Memoization

Memorization. Compute and Store the solution of each sub-problem in a cache the first time that you face it. lookup as needed.

Input: \( n, s(1), \ldots, s(n) \) and \( f(1), \ldots, f(n) \) and \( w_1, \ldots, w_n. \)

Sort jobs by finish times so that \( f(1) \leq f(2) \leq \cdots f(n). \)

Compute \( p(1), p(2), \ldots, p(n) \)

for \( j = 1 \) to \( n \)
    \( M[j] = \text{empty} \)
\( M[0] = 0 \)

\( OPT(j) \) {
    if (\( M[j] \) is empty)
        \( M[j] = \max (w_j + OPT(p(j)), OPT(j - 1)) \).
    return \( M[j] \)
}

In practice, you may get stack overflow if \( n \gg 10^6 \) (depends on the language).
You can also avoid recursion
• recursion may be easier conceptually when you use induction

**Input:** \( n, s(1), ..., s(n) \) and \( f(1), ..., f(n) \) and \( w_1, ..., w_n \).

**Sort** jobs by finish times so that \( f(1) \leq f(2) \leq ... f(n) \).

**Compute** \( p(1), p(2), ..., p(n) \)

\[
OPT(j) \{
    M[0] = 0
    for \ j = 1 \ to \ n
        M[j] = \max (w_j + M[p(j)], M[j-1]).
\}

Output \( M[n] \)

**Claim:** \( M[j] \) is value of \( OPT(j) \)

**Timing:** Easy. Main loop is \( O(n) \); sorting is \( O(n \log n) \).
Example

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).
\( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left( w_j + OPT(p(j)), OPT(j - 1) \right) & \text{o.w.}
\end{cases}
\]
Example

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).

\( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left( w_j + OPT(p(j)), OPT(j-1) \right) & \text{otherwise}
\end{cases}
\]

<table>
<thead>
<tr>
<th>( j )</th>
<th>( w_j )</th>
<th>( p(j) )</th>
<th>( OPT(j) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Example

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).

\( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left( w_j + OPT(p(j)), OPT(j-1) \right) & \text{o.w.}
\end{cases}
\]
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

$p(j) = \text{largest index } i < j \text{ such that job } i \text{ is compatible with } j$.

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left( w_j + OPT(p(j)), OPT(j - 1) \right) & \text{o.w.}
\end{cases}
\]
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

$p(j) = \text{largest index } i < j \text{ such that job } i \text{ is compatible with } j$.

\[ OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \{ w_j + OPT(p(j)), OPT(j - 1) \} & \text{otherwise}
\end{cases} \]
Example

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).
\( p(j) \) = largest index \( i < j \) such that job \( i \) is compatible with \( j \).

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left( w_j + OPT(p(j)), OPT(j - 1) \right) & \text{o.w.}
\end{cases}
\]

<table>
<thead>
<tr>
<th>( j )</th>
<th>( w_j )</th>
<th>( p(j) )</th>
<th>( OPT(j) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Example

Label jobs by finishing time: \( f(1) \leq \cdots \leq f(n) \).  

\( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left( w_j + OPT(p(j)), OPT(j-1) \right) & \text{o.w.} 
\end{cases}
\]

<table>
<thead>
<tr>
<th>( j )</th>
<th>( w_j )</th>
<th>( p(j) )</th>
<th>( OPT(j) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Diagram showing job durations and finish times.
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.
$p(j) = \text{largest index } i < j \text{ such that job } i \text{ is compatible with } j$.

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left( w_j + OPT(p(j)), OPT(j-1) \right) & \text{o.w.}
\end{cases}
\]
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$.

$p(j) = \text{largest index } i < j \text{ such that job } i \text{ is compatible with } j$. 

$$OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max (w_j + OPT(p(j)), OPT(j-1)) & \text{o.w.}
\end{cases}$$

<table>
<thead>
<tr>
<th>j</th>
<th>$w_j$</th>
<th>p(j)</th>
<th>OPT(j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>
Example

Label jobs by finishing time: $f(1) \leq \cdots \leq f(n)$. $p(j)$ = largest index $i < j$ such that job $i$ is compatible with $j$.

\[ OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left( w_j + OPT(p(j)), OPT(j - 1) \right) & \text{o.w.}
\end{cases} \]
Dynamic Programming

• Give a solution of a problem using smaller (overlapping) sub-problems where
  the parameters of all sub-problems are determined in-advance

• Useful when the same subproblems show up again and again in the solution.
Knapsack Problem
Knapsack Problem

Given $n$ objects and a "knapsack."
Item $i$ weighs $w_i > 0$ kilograms and has value $v_i > 0$.
Knapsack has capacity of $W$ kilograms.

**Goal:** fill knapsack so as to maximize total value.

**Ex:** OPT is $\{3, 4\}$ with value 40.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

$W = 11$

**Greedy:** repeatedly add item with maximum ratio $v_i/w_i$.

**Ex:** $\{5, 2, 1\}$ achieves only value = 35 $\Rightarrow$ greedy not optimal.
Dynamic Programming: First Attempt

Let $OPT(i) = \text{Max value of subsets of items } 1, \ldots, i \text{ of weight } \leq W$.

Case 1: $OPT(i)$ does not select item $i$
- In this case $OPT(i) = OPT(i - 1)$

Case 2: $OPT(i)$ selects item $i$
- In this case, item $i$ does not immediately imply we have to reject other items
- The problem does not reduce to $OPT(i - 1)$ because we now want to pack as much value into box of weight $\leq W - w_i$

Conclusion: We need more subproblems, we need to strengthen IH.
Stronger DP (Strengthening Hypothesis)

Let $OPT(i, w) = \text{Max value of subsets of items } 1, ..., i \text{ of weight } \leq w$

**Case 1: $OPT(i, w)$ selects item $i$**
- In this case, $OPT(i, w) = v_i + OPT(i - 1, w - w_i)$

**Case 2: $OPT(i, w)$ does not select item $i$**
- In this case, $OPT(i, w) = OPT(i - 1, w)$.

Therefore,

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i - 1, w) & \text{if } w_i > w \\ \max(OPT(i - 1, w), v_i + OPT(i - 1, w - w_i)) & \text{o.w.,} \end{cases}$$
DP for Knapsack

**Recursive**

```plaintext
Compute-OPT(i, w)
    if M[i, w] == empty
        if (i==0)
            M[i, w] = 0
        else if (w_i > w)
            M[i, w] = Comp-OPT(i-1, w)
        else
            M[i, w] = max {Comp-OPT(i-1, w), v_i + Comp-OPT(i-1, w-w_i)}
    return M[i, w]
```

**Non-recursive**

```plaintext
for w = 0 to W
    M[0, w] = 0
for i = 1 to n
    for w = 1 to W
        if (w_i > w)
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i]}
    return M[n, W]
```
**DP for Knapsack**

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

The table represents the dynamic programming solution for the knapsack problem. The 2D array `M[i, w]` is filled with values that represent the maximum value that can be achieved with a subset of the first `i` items and a weight limit of `w`.

**Algorithm**

1. Initialize the base cases:
   - `M[0, w] = 0` for all `w`.
   - `M[i, 0] = 0` for all `i`.

2. For each item `i` and weight `w` with `0 ≤ i ≤ n` and `0 ≤ w ≤ W`, calculate:
   - If the weight of the current item, `w_i`, is greater than the current weight, `w`, then:
     
     \[
     M[i, w] = M[i-1, w]
     \]
   
   - Else, calculate:
     
     \[
     M[i, w] = \max\{M[i-1, w], v_i + M[i-1, w-w_i]\}
     \]

3. The final answer is `M[n, W]`.

Example:

```
Item: 1, 2, 3, 4, 5
Value: 1, 6, 18, 22, 28
Weight: 1, 2, 5, 6, 7

Matrix:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>φ</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>{1}</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{1,2}</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{1,2,3}</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{1,2,3,4}</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{1,2,3,4,5}</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

For instance, to find the maximum value that can be achieved with a weight of `W = 11`:

\[
M[5, 11] = \max\{M[4, 11], v_5 + M[4, 11-w_5]\} = \max\{0, 28 + 0\} = 28
\]

The final value `M[5, 11]` is 28.
### DP for Knapsack

#### Table

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

#### Algorithm

if \( w_i > w \)

\[
M[i, w] = M[i-1, w]
\]

else

\[
M[i, w] = \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \}
\]
**DP for Knapsack**

**Item** | **Value** | **Weight**
---|---|---
1 | 1 | 1  
2 | 6 | 2  
3 | 18 | 5  
4 | 22 | 6  
5 | 28 | 7

**M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i]}**

**OPT:** \{4, 3\}

value = 22 + 18 = 40

if \(w_i > w\)

\[M[i, w] = M[i-1, w]\]

else

\[M[i, w] = \max \{M[i-1, w], v_i + M[i-1, w-w_i]\}\]
### DP for Knapsack

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

#### OPT:
- \( \{ 4, 3 \} \)
- Value: \( 22 + 18 = 40 \)

#### Algorithm:
- \( \text{if } (w_i > w) \)\n  - \( M[i, w] = M[i-1, w] \)
- \( \text{else} \)\n  - \( M[i, w] = \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \} \)
## DP for Knapsack

### Table:

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

### Optimal Solution:

\[
\text{OPT: } \{ 4, 3 \}, \quad \text{value} = 22 + 18 = 40
\]

### DP Algorithm:

\[
\begin{align*}
\text{if } & (w_i > w) \\
\quad & M[i, w] = M[i-1, w] \\
\text{else} & \\
\quad & M[i, w] = \max \{M[i-1, w], v_i + M[i-1, w-w_i]\}
\end{align*}
\]
### DP for Knapsack

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

**OPT:** \{4, 3\}

\[
\text{value} = 22 + 18 = 40
\]

\[
\begin{align*}
\text{if } (w_i > w) & \quad \text{then} \\
M[i, w] &= M[i-1, w] \\
\text{else} & \\
M[i, w] &= \max \{M[i-1, w], v_i + M[i-1, w-w_i]\}
\end{align*}
\]

\[
W = 11
\]
Knapsack Problem: Running Time

Running time: $\Theta(n \cdot W)$

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete.

Knapsack approximation algorithm:
There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum in time $\text{Poly}(n, \log W)$. 
DP Ideas so far

• You may have to define an ordering to decrease subproblems

• You may have to strengthen DP, equivalently the induction, i.e., you have may have to carry more information to find the Optimum.

• This means that sometimes we may have to use two dimensional or three dimensional induction