Life can only be understood backwards; but it must be lived forwards.

Soren Kierkegaard
Dynamic programming

- It has nothing to do with “programming languages”
• Problem: Input \( w_1 \ w_2 \ \ldots \ w_n \), \( t \) each \( 0 \leq w_i \leq k \)

• Output: Number of inputs \( x \in \{0,1\}^n \): \( \sum w_i x_i = t \)

• Let's try a recursive approach...
Problem: Input \( w_1, w_2, \ldots, w_n \), \( t \) each \( 0 \leq w_i \leq k \)

Output: Number of inputs \( x \in \{0,1\}^n : \sum w_i x_i = t \)

\( S(i,s) := \) number of inputs \( x \in \{0,1\}^i \) such that \( \sum_{j \leq i} w_j x_j = s \)

Structure of solutions: \( S(i,s) = S(i-1,s) + S(i-1,s-w_i) \) \( i = n, \ldots \)

Gives recursive approach: \( T(n) = \) ?
Problem: Input \( w_1 \ w_2 \ldots \ w_n \) , \( t \) each \( 0 \leq w_i \leq k \)

Output: Number of inputs \( x \in \{0,1\}^n : \sum w_i x_i = t \)

\( S(i,s) := \) number of inputs \( x \in \{0,1\}^i \) such that \( \sum_{j \leq i} w_j x_j = s \)

Structure of solutions: \( S(i,s) = S(i-1,s) + S(i-1,s–w_i) \) \( i = n, \ldots \)

Gives recursive approach: \( T(n) = 2 \ T(n-1) \Rightarrow T(n) \geq 2^n \)

How can we do faster when \( k = n \) ?
Problem: Input $w_1 \ w_2 \ \ldots \ \ w_n , \ t$ each $0 \leq w_i \leq k$

Output: Number of inputs $x \in \{0,1\}^n : \sum w_i \ x_i = t$

$S(i,s) := \text{number of inputs } x \in \{0,1\}^i \text{ such that } \sum_{j \leq i} w_j \ x_j = s$

Structure of solutions: $S(i,s) = S(i-1,s) + S(i-1,s–w_i) \ i = n,...$

Gives recursive approach: $T(n) = 2 \ T(n-1) \Rightarrow T(n) \geq 2^n$

How can we do faster when $k = n$?

Stop solving the same problems over and over again!

Total sum is $\leq kn$, so there really are only $\ ?$ different $S(i,t)$
Problem: Input \(w_1, w_2, \ldots, w_n\), \(t\) each \(0 \leq w_i \leq k\)

Output: Number of inputs \(x \in \{0,1\}^n : \sum w_i x_i = t\)

\[S(i,s) := \text{number of inputs } x \in \{0,1\}^i \text{ such that } \sum_{j \leq i} w_j x_j = s\]

Structure of solutions: \(S(i,s) = S(i-1,s) + S(i-1,s-w_i)\) \(i = n,\ldots\)

Gives recursive approach: \(T(n) = 2 \cdot T(n-1) \Rightarrow T(n) \geq 2^n\)

How can we do faster when \(k = n\) ?

Stop solving the same problems over and over again!

Total sum is \(\leq kn\), so there really are only \(kn^2\) different \(S(i,t)\)
Just solve all of those!
- Problem: Input $w_1 \ w_2 \ \ldots \ w_n$, $t$ each $0 \leq w_i \leq k$
- Output: Number of inputs $x \in \{0,1\}^n : \sum w_i \ x_i = t$

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sum $s$

$\mathit{Algorithm}$

- Fill first column
- (for $i = 2 \ldots n$)
  (for $s = 0 \ldots kn$)
**Problem:** Input $w_1 w_2 \ldots w_n$, $t$ each $0 \leq w_i \leq k$

**Output:** Number of inputs $x \in \{0,1\}^n : \sum w_i x_i = t$

```
  1
  
  
  
  i = 1 \ldots n
```

- Fill first column
- (for $i = 2 \ldots n$)
  (for $s = 0 \ldots kn$)
  
  \[ S(i,s) = S(i-1,s) + S(i-1,s-w_i) \]

- $T(n) = ?$
Problem: Input $w_1, w_2, \ldots, w_n, t$ each $0 \leq w_i \leq k$

Output: Number of inputs $x \in \{0,1\}^n : \sum w_i x_i = t$

- Fill first column
  - (for $i = 2 \ldots n$)
    - (for $s = 0 \ldots kn$)
      - $S(i,s) = S(i-1,s) + S(i-1,s-w_i)$

$T(n) = O(kn^2)$
Steps for dynamic programming approach:

- **Identify subproblems** (here $S(i,s)$)

- **Count subproblems** (here $kn^2$)

- **Obtain recursion** (here $S(i,s) = S(i-1,s) + S(i-1,s-w_i)$)
  (aka structure of solutions, optimal substructure property)

- The algorithm solves all the subproblems

- Running time = Number of subproblems (here $kn^2$)  
  $\times$ Time to compute recursion (here $O(1)$)
Problem: Have $t$ and $\infty$ piles of coins of values $d_1, \ldots, d_k$

Want to use minimum number of coins to obtain target $t$

Example: $d = (5,4,1)$  $t = 8$  $t = 5+1+1+1$,  $t = 4+4$

Subproblems: $\text{Cost}[c] := \text{number of coins to obtain } c$

Number of subproblems: $t$
Problem: Have \( t \) and \( \infty \) piles of coins of values \( d_1, \ldots, d_k \).

Want to use minimum number of coins to obtain target \( t \).

Example: \( d = (5, 4, 1) \) \( t = 8 \) \( t = 5+1+1+1, \ t = 4+4 \)

Structure of solution: \( \text{Cost}[c] = ? \)

Suppose you obtain \( c \) with some optimal number of coins. If coin of type \( d_i \) was used, then your coins without \( d_i \) must be optimal for \( c - d_i \).
• Problem: Have \( t \) and \( \infty \) piles of coins of values \( d_1 ,... ,d_k \).
• Want to use minimum number of coins to obtain target \( t \).
• Example: \( d = (5,4,1) \) \( t = 8 \) \( t = 5+1+1+1, \ t = 4+4 \).
• Structure of solution: \( \text{Cost}[c] = 1 + \min_{i \leq k} \text{Cost}[c - d_i] \).

Algorithm:
• Initialize vector Cost to 0
• For \( (c = 1..t) \)
Problem: Have $t$ and $\infty$ piles of coins of values $d_1, \ldots, d_k$

Want to use minimum number of coins to obtain target $t$

Example: $d = (5,4,1)$  $t = 8$  $t = 5+1+1+1, \ t = 4+4$

Structure of solution:  $\text{Cost}[c] = 1 + \min_{i \leq k} \text{Cost}[c - d_i]$

Algorithm:

- Initialize vector $\text{Cost}$ to 0
- For $(c = 1..t)$ $\text{Cost}[c] = 1 + \min_{i \leq k} \text{Cost}[c - d_i]$

$T(n) = kn$

To know which coins to use, store $\text{argmin}$

Can reconstruct solution backwards
• Dynamic programming in economics

• Let us plan Bob's next L years.

• At the beginning of each year he owns savings $+ \text{ wage}$

• He must decide how much to consume, rest is saved
  Savings earn interest $(1+\rho)$
  Consuming $C$ yields utility $\log(C)$
  ($\$10K$ vs. $\$20K$ is different from $\$1M$ vs. $\$1M+$\$10K$)

• He wants to maximize sum of utility throughout his lifetime

• He starts with savings $0$. 
• Formulate as problem

• $U[k,i] := \text{utility for years } i, i+1, \ldots, L \text{ if at beginning of year } i$
  
  savings + wage = k

• $U[k,L] := ?$

How much should Bob consume in his last year of life?
• Formulate as problem

• \( U[k,i] := \text{utility for years } i, i+1, \ldots, L \text{ if at beginning of year } i \) savings + wage = \( k \)

• \( U[k,L] := \log(k) \)
  Consumption = \( k \), because at last year \( L \) he spends all

• \( U[k,i] := ? \)
• Formulate as problem

• $U[k,i] := \text{utility for years } i, i+1, \ldots, L \text{ if at beginning of year } i$
  
  savings + wage = $k$

• $U[k,L] := \log(k)$
  
  Consumption = $k$, because at last year $L$ he spends all

• $U[k,i] := \max_c \log(c) + U[(k - c)(1+p) + w, i+1]$
  
  Consumption = $\text{argmax}$

• A recursive algorithm for $U[0,0]$ would take time $T \geq 2^L$

• Dynamic programming takes time $\text{poly}(L,W)$
• Longest common subsequence

• Given two strings $X$, $Y$, want to find a longest subsequence $Z$,
  
  i.e. string $Z$ whose symbols appear in $X$, $Y$ in the same order, but not necessarily consecutively

• Example: $X = XMJYAUZ$
  
  $Y = MZJAWXU$
  
  $Z = MJAU$
• **Definition:** For a string $X = (x_1, x_2, \ldots, x_n)$,
  
  we denote by $X_i$ the prefix $(x_1, x_2, \ldots, x_i)$.

• $X_0$ is the empty subsequence $\emptyset$

• Do not confuse $x$ with $X$

• $\text{LCS}(X_i, Y_j) = \text{longest subsequence of } X_i \text{ and } Y_j$

• Optimal substructure? What if $X_i$ and $Y_j$ end with the same symbol $x_i = y_j$?
• **Definition**: For a string \( X = (x_1, x_2, \ldots, x_n) \), we denote by \( X_i \) the prefix \((x_1, x_2, \ldots, x_i)\).

• \( X_0 \) is the empty subsequence \( \emptyset \).

• Do not confuse \( x \) with \( X \).

• \( \text{LCS}(X_i, Y_j) = \text{longest subsequence of } X_i \text{ and } Y_j \)

\[
\text{LCS}(X_i, Y_j) = \begin{cases} 
\emptyset & \text{if } i = 0 \text{ or } j = 0 \\
(LCS(X_{i-1}, Y_{j-1}), x_i) & \text{if } x_i = y_j \\
\text{longest}(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) & \text{if } x_i \neq y_j
\end{cases}
\]

function LCSLength(X[1..m], Y[1..n])
    C = array(0..m, 0..n)
    for i := 0..m
        C[i,0] = 0
    for j := 0..n
        C[0,j] = 0
    for i := 1..m
        for j := 1..n
            if X[i] = Y[j]
                C[i,j] := C[i-1,j-1] + 1
            else
                C[i,j] := max(C[i,j-1], C[i-1,j])
    return C[m,n]

T(m,n) = O(m n)
• As before, if we want to output the sequence, we record which rule was used at each point

\[
LCS(X_i, Y_j) = \begin{cases} 
\emptyset & \text{if } i = 0 \text{ or } j = 0 \\
(LCS(X_{i-1}, Y_{j-1}), x_i) & \text{if } x_i = y_j \\
\text{longest}(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) & \text{if } x_i \neq y_j
\end{cases}
\]

What arrows?

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>C</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>G</td>
<td>Ø</td>
<td>(G)</td>
<td>Ø</td>
<td>(G)</td>
<td>(G)</td>
</tr>
<tr>
<td>A</td>
<td>(A)</td>
<td>(A)</td>
<td>(A)</td>
<td>(GA)</td>
<td>(GA)</td>
</tr>
<tr>
<td>C</td>
<td>(A)</td>
<td>(A)</td>
<td>(AC)</td>
<td>(AC) &amp; (AC) &amp; (AC) &amp; (GA) &amp; (AC) &amp; (AC) &amp; (GC) &amp; (GC) &amp; (GA) &amp; (GA)</td>
<td></td>
</tr>
</tbody>
</table>

• Then we can reconstruct the sequence backwards.
As before, if we want to output the sequence, we record which rule was used at each point:

\[
\text{LCS} (X_i, Y_j) = \begin{cases} 
\emptyset & \text{if } i = 0 \text{ or } j = 0 \\
(LCS (X_{i-1}, Y_{j-1}), x_i) & \text{if } x_i = y_j \\
\text{longest} (LCS (X_i, Y_{j-1}), LCS (X_{i-1}, Y_j)) & \text{if } x_i \neq y_j 
\end{cases}
\]

Then we can reconstruct the sequence backwards.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>C</th>
<th></th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>G</td>
<td>Ø</td>
<td>(G)</td>
<td>(G)</td>
<td>(G)</td>
<td>(G)</td>
<td>(G)</td>
</tr>
<tr>
<td>A</td>
<td>(A)</td>
<td>(A)</td>
<td>(A) &amp; (G)</td>
<td>(GA)</td>
<td>(GA)</td>
<td>(GA)</td>
</tr>
<tr>
<td>C</td>
<td>(A)</td>
<td>(A)</td>
<td>(AC) &amp; (GC)</td>
<td>(AC) &amp; (GC) &amp; (GA)</td>
<td>(AC) &amp; (GC) &amp; (GA)</td>
<td>(AC) &amp; (GC) &amp; (GA)</td>
</tr>
</tbody>
</table>
• We have described dynamic programming in an iterative “bottom-up” fashion, i.e., solve all the problems from the smallest to the biggest.

• Alternatively, dynamic programming may be implemented in a “top-down” recursive fashion. You keep a list of the subproblems solved, and at the beginning you check if the current subproblem was already solved, if so you just read off the solution and return.

• This is called Memoization

• Recall even divide-and-conquer may be implemented either in a recursive “top-down” fashion, or in an iterative “bottom-up” fashion.
• Dynamic programming requires solving all subproblems, leads to algorithms with running time usually $n^2$ or $n^3$

• Sometimes, greedy is faster.
Greedy Algorithms

A greedy algorithm always makes the choice that looks best at the moment.

That is, it keeps making locally optimal decision in the hope that this will lead to a globally optimal solution.
Activity Selection problem

Input: Set of $n$ activities that need the same resource.

$A := \{ a_1, a_2, \ldots, a_n \}, \ \forall \ i \in [n] \ \ a_i = [s_i, f_i].$

Activity $a_i$ takes time $[s_i, f_i]$. Activities $a_i$, $a_j$ are compatible if $s_j \geq f_i$.

Output:

Maximum-size subset of mutually compatible activities.
Example:

<table>
<thead>
<tr>
<th>a_i</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>s_i</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>f_i</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

A set of compatible activities = ?

![Diagram showing overlapping activities]
Example:

<table>
<thead>
<tr>
<th>a_i</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>s_i</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>f_i</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

A set of compatible activities = \((a_3, a_9, a_{11})\).
Example:

<table>
<thead>
<tr>
<th>(a_i)</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>(s_i)</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>(f_i)</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

A set of compatible activities = \((a_3, a_9, a_{11})\).

A maximal set of compatible activities = ?
Example:

<table>
<thead>
<tr>
<th>$a_i$</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>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

A set of compatible activities = $(a_3, a_9, a_{11})$.

A maximal set of compatible activities = $(a_1, a_4, a_8, a_{11})$. 
Example:

<table>
<thead>
<tr>
<th>$a_i$</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>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

A set of compatible activities = $(a_3, a_9, a_{11})$.

A maximal set of compatible activities = $(a_1, a_4, a_8, a_{11})$.

Is there another maximal set?
Example:

\[
\begin{array}{cccccccccccc}
 a_i & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
 s_i & 1 & 3 & 0 & 5 & 3 & 5 & 6 & 8 & 8 & 2 & 12 \\
 f_i & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\end{array}
\]

A set of compatible activities = \((a_3, a_9, a_{11})\).

A maximal set of compatible activities = \((a_1, a_4, a_8, a_{11})\).

Is there another maximal set? Yes. \((a_2, a_4, a_9, a_{11})\)
• **Claim**: some optimal solution contains activity with earliest finish time

• **Proof**:  
  Let \([s^*, f^*]\) be activity with earliest finish time \(f^*\)
  
  Let \(S\) be an optimal solution
  
  Write \(S = S' \cup [s, f)\) where \([s, f)\) has earliest finish time among activities in \(S\)

• Then \(S' \cup [s^*, f^*]\) is also an optimal solution, because every activity in \(S'\) has start time \(> f > f^*\).
• **Greedy Algorithm:**
  Pick activity with earliest finish time,
  that does not overlap with activities already picked
  Repeat

• **Claim:** The algorithm is correct
• **Proof:** Follows from applying previous claim iteratively.

• Let us see the algorithm in more detail
Greedy activity selection algorithm

activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := {a[i]};
    i := 1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i] ) {
            Add a[i] to S;
            i := m;
        }
    return S
}
activity-selection(A) {
    sort A increasingly according to f[i];
    n:= length[A];
    S:={a[i]};
    i:=1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i] ) {
            Add a[i] to S;
            i :=m;}
    return S;
}
activity-selection(A) {
    sort A increasingly according to f[i];
    n:= length[A];
    S:={a[i]};
    i:=1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i] ) {
            Add a[i] to S;
            i :=m;}
    return S;
}
```plaintext
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := {a[i]};
    i := 1;
    for (m = 2; m ≤ n; m++)
        if (s[m] ≥ f[i]) {
            Add a[i] to S;
            i := m;
        }
    return S;
}

Example:

<table>
<thead>
<tr>
<th>a_i</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>s_i</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>f_i</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>
```

activity-selection(A) {
    sort A increasingly according to $f[i]$;
    n:= length[A];
    S:={a[i]};
    i:=1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i] ) {
            Add a[i] to S;
            i :=m;}
    return S;
}

Example:
S:={a_1}
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := {a[i]};
    i := 1;
    for (m = 2; m ≤ n; m++)
        if (s[m] ≥ f[i]) {
            Add a[i] to S;
            i := m;
        }
    return S;
}

Example:
S := {a_1}
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := {a[i]};
    i := 1;
    for (m = 2; m ≤ n; m++)
        if (s[m] ≥ f[i]) {
            Add a[i] to S;
            i := m;
        }
    return S;
}

Example:
S := {a₁}

<table>
<thead>
<tr>
<th>aᵢ</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>sᵢ</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>fᵢ</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Example:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

1 2 3 5 6 7 8 9 10 11 12 13 14
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := {a[i]};
    i := 1;
    for (m = 2; m ≤ n; m++)
        if (s[m] ≥ f[i]) {
            Add a[i] to S;
            i := m;
        }
    return S;
}

Example:
S := {a_1}
s[2] ≥ f[1]?

\[\begin{array}{c|cccccccccccc}
  i & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
  \hline
  s_i & 1 & 3 & 0 & 5 & 3 & 5 & 6 & 8 & 8 & 2 & 12 \\
  f_i & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14
\end{array}\]
activity-selection(A) {
    sort A increasingly according to f[i];
    n:= length[A];
    S:={a[i]};
    i:=1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i]) {
            Add a[i] to S;
            i := m;
        }
}
activity-selection(A) {
  sort A increasingly according to f[i];
  n:= length[A];
  S:=\{a[i]\};
  i:=1;
  for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i]) {
      Add a[i] to S;
      i := m;
    }
  return S;
}

Example:
S:=\{a_1\}
s[3] ≥ f[1]? n:=11

\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c}
  a_i & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
  s_i & 1 & 3 & 0 & 5 & 3 & 5 & 6 & 8 & 8 & 2 & 12 \\
  f_i & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\end{array}
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := {a[i]};
    i := 1;
    for (m = 2; m ≤ n; m++)
        if (s[m] ≥ f[i])
            Add a[i] to S;
            i := m;
} return S;

Example:
S := {a₁}

<table>
<thead>
<tr>
<th>aᵢ</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>fᵢ</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>sᵢ</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
</tbody>
</table>

Example Table:

<table>
<thead>
<tr>
<th>i</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

Example Diagram:
activity-selection(A) {
  sort A increasingly according to f[i];
  n:= length[A];
  S:=\{a[i]\};
  i:=1;
  for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i]) {
      Add a[i] to S;
      i := m;
    }
  return S;
}
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := \{a[i]\};
    i := 1;
    for (m = 2; m ≤ n; m++)
        if (s[m] ≥ f[i])
            Add a[i] to S;
        i := m;
    return S;
}
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := {a[i]};
    i := 1;
    for (m = 2; m ≤ n; m++)
        if (s[m] ≥ f[i]) {
            Add a[i] to S;
            i := m;
        }
    return S;
}
activity-selection(A) {
    sort A increasingly according to f[i];
    n:= length[A];
    S:={a[i]};
    i:=1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i] ) {
            Add a[i] to S;
            i :=m;}
    return S;
}
activity-selection(A) {
    sort A increasingly according to f [i];
    n:= length[A];
    S:={a[i]};
    i:=1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i] ) {
            Add a[i] to S;
            i := m;
        }
    return S;
}

Example:
S:={a_1, a_4}

<table>
<thead>
<tr>
<th>a_i</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>s_i</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>f_i</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Example graph:
activity-selection(A) {
    sort A increasingly according to f[i];
    n:= length[A];
    S:=\{a[i]\};
    i:=1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i] ) {
            Add a[i] to S;
            i :=m;}
    return S;
}

Example:
S:=\{a_1 ,a_4\}
Example:
S:=\{a_1, a_4\}

s[6] ≥ f[4]?

```
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := \{a[i]\};
    i := 1;
    for (m = 2; m ≤ n; m++)
        if (s[m] ≥ f[i]) {
            Add a[i] to S;
            i := m;
        }
    return S;
}
```
activity-selection(A) {
    sort A increasingly according to f[i];
    n:= length[A];
    S:={a[i]};
    i:=1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i]) {
            Add a[i] to S;
            i :=m;}
    return S;
}

Example:
S:={a_1, a_4}
activity-selection(A) {
    n:= length[A];
    S:={a[i]};
    i:=1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i] ) {
            Add a[i] to S;
            i :=m;
        }
        return S;
}
activity-selection(A) {
    sort A increasingly according to f[i];
    n:= length[A];
    S:={a[i]};
    i:=1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i] ) {
            Add a[i] to S;
            i :=m;}
}
activity-selection(A) {
    sort A increasingly according to f[i];
    n:= length[A];
    S:={a[i]};
    i:=1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i]) {
            Add a[i] to S;
            i := m;
        }
    return S;
}
activity-selection(A) {
  sort A increasingly according to f[i];
  n:= length[A];
  S:={a[i]};
  i:=1;
  for (m=2; m ≤ n; m++)
    if (s[m] ≥ f[i] ) {
      Add a[i] to S;
      i :=m;
    }
  return S;
}

Example:
S:={a_1 ,a_4 ,a_8}
activity-selection(A) {
    sort A increasingly according to \( f[i] \);
    \( n := \text{length}[A] \);
    \( S := \{a[i]\} \);
    \( i := 1 \);
    for (\( m = 2; m < n; m++ \)) {
        if (\( s[m] \geq f[i] \)) {
            Add \( a[i] \) to \( S \);
            \( i := m \);
        }
    }
    return \( S \);
}
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := {a[i]};
    i := 1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i]) {
            Add a[i] to S;
            i := m;
        }
    return S;
}

Example:
S := {a_1, a_4, a_8}

<table>
<thead>
<tr>
<th>i</th>
<th>m</th>
<th>n := 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_i</td>
<td>1 2 3 4 5 6 7 8 9 10 11</td>
<td></td>
</tr>
<tr>
<td>s_i</td>
<td>1 3 0 5 3 5 6 8 8 2 12</td>
<td></td>
</tr>
<tr>
<td>f_i</td>
<td>4 5 6 7 8 9 10 11 12 13 14</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := {a[i]};
    i := 1;
    for (m = 2; m ≤ n; m++)
        if (s[m] ≥ f[i]) {
            Add a[i] to S;
            i := m;
        }
} return S;

Example:
S := \{a_1, a_4, a_8\}
s[9] ≥ f[8]?

<table>
<thead>
<tr>
<th>a_i</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>s_i</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>f_i</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Example:
S := \{a_1, a_4, a_8\}
s[9] ≥ f[8]?
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := {a[i]};
    i := 1;
    for (m = 2; m ≤ n; m++)
        if (s[m] ≥ f[i]) {
            Add a[i] to S;
            i := m;
        }
    return S;
}
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := {a[i]};
    i := 1;
    for (m = 2; m ≤ n; m++)
        if (s[m] ≥ f[i]) {
          Add a[i] to S;
          i := m;
        }
    return S;
}

Example:
S := {a_1, a_4, a_8}
s[10] ≥ f[8]?

<table>
<thead>
<tr>
<th>a_i</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>s_i</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>f_i</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>
activity-selection(A) {
    sort A increasingly according to f[i];
    n:= length[A];
    S:={a[i]};
    i:=1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i] ) {
            Add a[i] to S;
            i :=m;}
    return S;
}

Example:
S:={a_1 ,a_4 ,a_8}
s[10] < f[8]

<table>
<thead>
<tr>
<th></th>
<th>a_i</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>s_i</td>
<td></td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>f_i</td>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>
Example:
S := \{a_1, a_4, a_8\}

```
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := \{a[i]\};
    i := 1;
    for (m = 2; m ≤ n; m++)
        if (s[m] ≥ f[i])
            Add a[i] to S;
            i := m;
} return S;
```
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := {a[i]};
    i := 1;
    for (m = 2; m ≤ n; m++)
        if (s[m] ≥ f[i])
            Add a[i] to S;
        i := m;
    return S;
}
Example:

\[ S := \{a_1, a_4, a_8, a_{11}\} \]

activity-selection(A) {
    sort A increasingly according to f[i];
    n:= length[A];
    S:={a[i]};
    i:=1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i] ) {
            Add a[i] to S;
            i :=m;
        }
    return S;
}

Example:
S:={a_1 ,a_4 ,a_8 ,a_{11}}  \quad \quad \quad m = 12, \quad n = 11
activity-selection(A) {
    sort A increasingly according to f[i];
    n:= length[A];
    S:={a[i]};
    i:=1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i] ) {
            Add a[i] to S;
            i :=m;}
    return S;
}
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := {a[i]};
    i := 1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i] ) {
            Add a[i] to S;
            i := m;
        } return S;
}
Knapsack

Input:
- $S := \{(v_1, w_1), (v_2, w_2), \ldots (v_n, w_n)\}$.
  - $(v_i, w_i)$ means item $i$ is worth $v_i$ and weighs $w_i$.
- $W$, weight-capacity of knapsack.

Output:
- Items that maximize value in knapsack.

Can we take a fraction of an item?
Fractional Knapsack: Yes
0-1 Knapsack: No

Fractional Knapsack

- Compute $v_i / w_i$ for each item.
- Sort $S$ according to $v_i / w_i$ decreasingly.
- Take as much as possible of the item with the most $v_i / w_i$
Fractional knapsack \((W, S)\)

Sort \(S\), decreasingly according to \(v_i / w_i\);

\(x[1..n] = 0; \ //x[i] = \text{amount of } i \text{ to be taken}\)

\(\text{weight} = 0; i = 1;\)

while (weight < \(W\) and \(i \leq n\))

\(\text{if weight} + w[i] \leq W \{\)

\(x[i] = 1;\)

\(\text{weight} += w[i];\)

\(i++\)

\}\) else {

\(x[i] = (W - \text{weight}) / w[i];\)

\(\text{weight} = W;\)

\}

return \(x\)
Fractional knapsack \((W, S)\)

Sort \(S\), decreasingly according to \(v_i / w_i\);

\(x[1..n] = 0; \quad // x[i] = \text{amount of i to be taken}\)

\(\text{weight} = 0; \quad i = 1;\)

\[
\text{while (weight < W and i ≤ n)} \\
\quad \text{if weight + w[i] ≤ W} \\
\quad \quad x[i] = 1; \\
\quad \quad \text{weight += w[i];} \\
\quad \quad i++ \\
\quad \text{else} \\
\quad \quad x[i] = (W - weight) / w[i]; \\
\quad \quad \text{weight = W;} \\
\]

\(\text{return} \ x\)
Fractional knapsack \((W, S)\)

Sort \(S\), decreasingly according to \(v_i / w_i\);

for \((i = 1; i \leq n; i++)\)

\(x[i] = 0;\)

\(\text{Weight} = 0; i = 1;\)

while (weight < \(W\) and \(i \leq n\))

if \(\text{weight} + w[i] \leq W\)

then \(x[i] = 1;\)

\{weight = weight + w[i];

\(i = i + 1;\}\}

else

\{\(x[i] = (w - \text{weight}) / w[i];\)

weight = \(W;\}\}

return \(x\)

Running time:

\(T(n) = O(n \log n).\)