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

Soren Kierkegaard
Dynamic programming

An interesting question is, "Where did the name, dynamic programming, come from?"
The 1950's were not good years for mathematical research. We had a very
interesting gentleman in Washington named Wilson. He was Secretary of Defence,
and he actually had a pathological fear and hatred of the word, research. I'm not
using the term lightly; I'm using it precisely. His face would suffuse, he would turn red,
and he would get violent if people used the term, research, in his presence. You can
imagine how he felt, then, about the term, mathematical. The RAND Corporation was
employed by the Air Force, and the Air Force had Wilson as its boss, essentially.
Hence, I felt I had to do something to shield Wilson and the Air Force from the fact
that I was really doing mathematics inside the RAND Corporation. What title, what
name, could I choose? In the first place, I was interested in planning, in decision-
making, in thinking. But planning, is not a good word for various reasons. I decided
therefore to use the word, "programming". I wanted to get across the idea that this
was dynamic, this was multistage, this was time-varying- I thought, let's kill two birds
with one stone. Let's take a word which has an absolutely precise meaning, namely
dynamic, in the classical physical sense. It also has a very interesting property as an
adjective, and that is it's impossible to use the word, dynamic, in the pejorative sense.
Try thinking of some combination which will possibly give it a pejorative meaning. It's
impossible. Thus, I thought dynamic programming was a good name. It was
something not even a Congressman could object to. So I used it as an umbrella for
my activities.

Richard Bellman, Eye of the Hurricane an autobiography, p. 159
The coin change problem

- You are a cashier and have k infinite piles of coins with values $d_1, \ldots, d_k$
  You have to give change for $t$
  You want to use the minimum number of coins

- **Definition:** $\text{Cost}[t] := \text{minimum number of coins to obtain } t$

- **Example:**
  $k = 3$, $d = (5,4,1)$, $t = 8$

  One solution has cost 4: $t = 5+1+1+1$
  A better solution has cost 2: $t = 4+4$, which is optimal

  $\text{Cost}[t] = 2.$
The coin change problem

- You are a cashier and have $k$ infinite piles of coins with values $d_1, \ldots, d_k$
- You have to give change for $t$
- You want to use the minimum number of coins

- **Definition**: $\text{Cost}[t] := \text{minimum number of coins to obtain } t$

- **Try to obtain a recursion:**
The coin change problem

- You are a cashier and have $k$ infinite piles of coins with values $d_1, \ldots, d_k$
  You have to give change for $t$
  You want to use the minimum number of coins

- **Definition:** $\text{Cost}[t] := \text{minimum number of coins to obtain } t$

- **Try to obtain a recursion:** To give change for $t$ you can:
The coin change problem

• You are a cashier and have k infinite piles of coins with values $d_1, \ldots, d_k$
You have to give change for t
You want to use the minimum number of coins

• Definition: $\text{Cost}[t] := \text{minimum number of coins to obtain } t$

• Try to obtain a recursion: To give change for $t$ you can:
use coin $d_1$, then need change for $t - d_1 \Rightarrow \text{Cost}[t] \leq 1 + \text{Cost}[t - d_1]$

The coin change problem

- You are a cashier and have $k$ infinite piles of coins with values $d_1, \ldots, d_k$.
  You have to give change for $t$.
  You want to use the minimum number of coins.

- **Definition:** $\text{Cost}[t] := \text{minimum number of coins to obtain } t$.

- **Try to obtain a recursion:** To give change for $t$ you can:
  - use coin $d_1$, then need change for $t - d_1 \Rightarrow \text{Cost}[t] \leq 1 + \text{Cost}[t - d_1]$.
  - or use coin $d_2$, then need change for $t - d_2 \Rightarrow \text{Cost}[t] \leq 1 + \text{Cost}[t - d_2]$. 

The coin change problem

- You are a cashier and have $k$ infinite piles of coins with values $d_1, \ldots, d_k$. You have to give change for $t$.
  You want to use the minimum number of coins.

- **Definition:** $\text{Cost}[t] := \text{minimum number of coins to obtain } t$

- **Try to obtain a recursion:** To give change for $t$ you can:
  - use coin $d_1$, then need change for $t - d_1 \Rightarrow \text{Cost}[t] \leq 1 + \text{Cost}[t - d_1]$
  - or use coin $d_2$, then need change for $t - d_2 \Rightarrow \text{Cost}[t] \leq 1 + \text{Cost}[t - d_2]$
  - or ...

  Which one to pick?
The coin change problem

• You are a cashier and have k infinite piles of coins with values $d_1, \ldots, d_k$
  You have to give change for $t$
  You want to use the minimum number of coins

• Definition: $\text{Cost}[t] := \text{minimum number of coins to obtain } t$

• Try to obtain a recursion: To give change for $t$ you can:
  use coin $d_1$, then need change for $t - d_1$ ⇒ $\text{Cost}[t] \leq 1 + \text{Cost}[t - d_1]$
  or use coin $d_2$, then need change for $t - d_2$ ⇒ $\text{Cost}[t] \leq 1 + \text{Cost}[t - d_2]$
  or

Which one to pick? The one that gives the minimum:

$\text{Cost}[t] = 1 + \min_{i \leq k} \text{Cost}[t - d_i]$
The coin change problem

• You are a cashier and have \( k \) infinite piles of coins with values \( d_1, \ldots, d_k \).
  You have to give change for \( t \).
  You want to use the minimum number of coins.

• Definition: \( \text{Cost}[t] := \text{minimum number of coins to obtain } t \)

• Recursion: \[
\text{Cost}[t] = 1 + \min_{i \leq k} \text{Cost}[t - d_i]
\]
The coin change problem

- You are a cashier and have $k$ infinite piles of coins with values $d_1, \ldots, d_k$. You have to give change for $t$. You want to use the minimum number of coins.

- **Definition**: $\text{Cost}[t] :=$ minimum number of coins to obtain $t$

- **Recursion**: $\text{Cost}[t] = 1 + \min_{i \leq k} \text{Cost}[t - d_i]$

- **A false start: a naive recursive algorithm**

```
Alg(t) {
  return \min_{i \leq k} Alg(t - d_i)
}
```
The coin change problem

- You are a cashier and have $k$ infinite piles of coins with values $d_1, \ldots, d_k$
- You have to give change for $t$
- You want to use the minimum number of coins

- **Definition:** $\text{Cost}[t] := \text{minimum number of coins to obtain } t$

- **Recursion**
  
  $$\text{Cost}[t] = 1 + \min_{i \leq k} \text{Cost}[t - d_i]$$

- **A false start:** a naive recursive algorithm

  ```
  Alg(t) {
    return \min_{i \leq k} Alg(t - d_i)
  }
  ```

- **Running time of Alg**, even for $k = 2$, $d_1 = 1$, $d_2 = 2$

- **$T(t)$**
  
  $$T(t) \geq T(t-1) + T(t-2) \geq T(t-2) + T(t-3) + T(t-2) \geq 2T(t-2)$$
The coin change problem

- You are a cashier and have $k$ infinite piles of coins with values $d_1, \ldots, d_k$.
- You have to give change for $t$.
- You want to use the minimum number of coins.

**Definition:** $\text{Cost}[t] :=$ minimum number of coins to obtain $t$

**Recursion**

$$\text{Cost}[t] = 1 + \min_{i \leq k} \text{Cost}[t - d_i]$$

- A false start: a naive recursive algorithm

```plaintext
Alg(t) {
    return $\min_{i \leq k} \text{Alg}(t - d_i)$
}
```

- Running time of Alg, even for $k = 2$, $d_1 = 1$, $d_2 = 2$.

$$T(t) \geq T(t-1) + T(t-2) \geq T(t-2) + T(t-3) + T(t-2) \geq 2T(t-2) \Rightarrow T(t) \geq 2^{t/2}$$
The coin change problem

• You are a cashier and have k infinite piles of coins with values $d_1, \ldots, d_k$. You have to give change for t. You want to use the minimum number of coins.

• Definition: $\text{Cost}[t]$ is the minimum number of coins to obtain t.

• Recursion: $\text{Cost}[t] = 1 + \min_{i \le k} \text{Cost}[t - d_i]$.

• A false start: a naive recursive algorithm.

• Running time of Alg, even for $k = 2$, $d_1 = 1$, $d_2 = 2$.

For example, below you are recursing multiple times on problem $\text{Cost}[t-2]$. You should only compute this once!

$T(t) \ge T(t-1) + T(t-2) \ge T(t-2) + T(t-3) + T(t-2) \ge 2T(t-2) \Rightarrow T(t) \ge 2^{t/2}$
The coin change problem

• You are a cashier and have k infinite piles of coins with values \(d_1, \ldots, d_k\) You have to give change for \(t\) You want to use the minimum number of coins

• Definition: Cost\([t]\) := minimum number of coins to obtain \(t\)

• Recursion

\[
\text{Cost}[t] = 1 + \min_{i \leq k} \text{Cost}[t - d_i]
\]

• Alg\((t)\): { Auxiliary array \(C[0..t]\)

\[
C[0] = 0
\]

For \((s = 1..t)\) {

\[
m = \text{minimum of } C[s - d_i] \text{ over } i = 1..k \text{ such that } s - d_i \geq 0
\]

\[
C[S] = 1 + m
\]

}

• Running time: \(O(t \cdot k)\)
The coin change problem

- Example:
  \( k = 3, \ d = (5,4,1), \ t = 8 \)
The coin change problem

- Example:
  $k = 3$, $d = (5, 4, 1)$, $t = 8$

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
0 & 1 & & & & & & & \\
\end{array}
\]

Cost[1] = 1 + Cost[0]
The coin change problem

- Example:
  \[ k = 3, \ d = (5,4,1), \ t = 8 \]

<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>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The coin change problem

- Example:
  \( k = 3, \; d = (5,4,1), \; t = 8 \)

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
0 & 1 & 2 & 3 & & & & & \\
\end{array}
\]

\[
\text{Cost}[3] = 1 + \text{Cost}[2]
\]
The coin change problem

Example:
k = 3, d = (5,4,1), t = 8

Cost[4] = 1 + Minimum(Cost[3], Cost[0]) = 1 + Minimum(3,0) = 1
The coin change problem

- Example:
  \( k = 3, \ d = (5,4,1), \ t = 8 \)

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
0 & 1 & 2 & 3 & 1 & 1 & & & \\
\end{array}
\]

\[
\text{Cost}[5] = 1 + \text{Minimum}(\text{Cost}[4], \text{Cost}[1], \text{Cost}[0]) = 1 + \text{Minimum}(1, 1, 0) = 1
\]
The coin change problem

- Example:
  \( k = 3, \ d = (5,4,1), \ t = 8 \)

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
C & 0 & 1 & 2 & 3 & 1 & 1 & 2 & \text{\_} & \text{\_}
\end{array}
\]

\[
\text{Cost}[6] = 1 + \text{Minimum}(\text{Cost}[5], \text{Cost}[2], \text{Cost}[1]) = 1 + \text{Minimum}(1,2,1) = 2
\]
The coin change problem

- Example:
  \( k = 3 \), \( d = (5,4,1) \), \( t = 8 \)

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
0 & 1 & 2 & 3 & 1 & 1 & 2 & 3 & \text{\textbullet}\n\end{array}
\]

\[
\text{Cost}[7] = 1 + \text{Minimum}(\text{Cost}[6], \text{Cost}[3], \text{Cost}[2]) = 1 + \text{Minimum}(2,3,2) = 3
\]
The coin change problem

- Example:
  \( k = 3, \ d = (5, 4, 1), \ t = 8 \)

\[
\begin{array}{cccccccccc}
& 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
C & 0 & 1 & 2 & 3 & 1 & 1 & 2 & 3 & 2 \\
\end{array}
\]

\[
\text{Cost}[8] = 1 + \text{Minimum}(\text{Cost}[7], \text{Cost}[4], \text{Cost}[3]) = 1 + \text{Minimum}(3, 1, 3) = 2
\]
The coin change problem

• So far we computed how many coins

• Now want to know which values, as in $8 = 4+4$

• Alg2(t):  
  Auxiliary arrays $C[0..t]$, $A[0..t]$
  $C[0] = 0$; $A[0] = 0$
  For $(s = 1..t)$ {
    $m = \text{minimum of } C[s - d_i]$ over $i = 1..k$ such that $s - d_i \geq 0$
    $i = \text{arg-minimum}$
    $C[s] = 1 + m$
    $A[s] = d_i$
  }

• Idea: values are: $A[t]$, $A[t-A[t]]$, ... until you get zero
The coin change problem

- Printing the coins used

- Print-Coins(t) {
  for(i = t; i > 0; i = i – A[i])
    Print(A[i])
}

- Time $O(t)$
The coin change problem

- Example:
  \( k = 3, \ d = (5,4,1), \ t = 8 \)

<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>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Print-coins(8) = 4, 4
The coin change problem

- Example:
k = 3, d = (5,4,1), t = 8

<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>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Print-coins(7) = 5, 1, 1
Steps for dynamic programming

• Identify subproblems (In coin-change example Cost[1..t])

• Obtain recursion \( (\text{Cost}[t] = 1 + \min_{i \leq k} \text{Cost}[t - d_i] ) \)
  (aka structure of solutions, optimal substructure property)

• Algorithm solves all the subproblems, once

• Running time = Number of subproblems (here t) 
  \( \times \) Time to compute recursion (here \( O(k) \) )
• Saw dynamic programming as iterative, “bottom-up”: solve all the problems from the smallest to the biggest.

• Can also be implemented in a “top-down” recursive fashion: 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.
Longest common subsequence

- Given two strings $X$ and $Y$ over some alphabet, want to find a longest subsequence $Z$.
  The symbols in $Z$ appear in $X$, $Y$ in the same order, but not necessarily consecutively.

- Example: Alphabet = \{A, C, G, T\}
  
  $$X = \text{A A G G A C A C T C T A G C G A T}$$
  $$Y = \text{T G G C A T T T A C G C G C A A}$$
Longest common subsequence

• Given two strings $X$ and $Y$ over some alphabet, want to find a longest subsequence $Z$.
  The symbols in $Z$ appear in $X$, $Y$ in the same order, but not necessarily consecutively

• Example: Alphabet = \{A, C, G, T\}
  \begin{align*}
  X &= A A G G A C A C T C T A G C G A T \\
  Y &= T G G C A T T T A C G C G C A A \\
  Z &= G A T T A C A
  \end{align*}
Arriving at subproblems and recursion

X = A A G G A C A C T C T A G C G A T
Y = T G G C A T T T A C G C G C A A

The strings X and Y end with different symbols. So either last T in X is not part of the solution, or last A in Y is not part of the solution.

In the first case I can remove last T from X. Now both strings end with A, which can be matched.

In the latter case I can remove the last A from Y.
Longest common subsequence

• On input $X[1..m]$, $Y[1..n]$, consider the prefixes $X[1..i]$, $Y[1..j]$ for any $i \leq m$, $j \leq n$.

• Subproblems:
  $L(i,j) =$ length longest subsequence of $X[1..i]$ and $Y[1..j]$

• Recursion:
  
  \[
  \begin{align*}
  \text{if } i &= 0 \text{ or } j = 0 & L(i,j) &= 0 \\
  \text{if } X[i] &= Y[j] & L(i,j) &= \? \\
  \text{if } X[i] &\neq Y[j] & L(i,j) &= \?
  \end{align*}
  \]
Longest common subsequence

- On input $X[1..m], Y[1..n]$, consider the prefixes $X[1..i], Y[1..j]$ for any $i \leq m, j \leq n$.

- Subproblems:
  $L(i,j) = \text{length longest subsequence of } X[1..i] \text{ and } Y[1..j]$

- Recursion:
  
  \[
  \begin{align*}
  &\text{if } i = 0 \text{ or } j = 0 \quad L(i,j) = 0 \\
  &\text{if } X[i] = Y[j] \quad L(i,j) = L(i-1,j-1) + 1 \\
  &\text{if } X[i] \neq Y[j] \quad L(i,j) = ?
  \end{align*}
  \]
Longest common subsequence

- On input \(X[1..m], Y[1..n]\),
  consider the prefixes \(X[1..i], Y[1..j]\) for any \(i \leq m, j \leq n\).

- Subproblems:
  \(L(i,j) = \text{length longest subsequence of } X[1..i] \text{ and } Y[1..j]\)

- Recursion:
  
  \[
  \begin{align*}
  &\text{if } i = 0 \text{ or } j = 0 \quad L(i,j) = 0 \\
  &\text{if } X[i] = Y[j] \quad L(i,j) = L(i-1,j-1) + 1 \\
  &\text{if } X[i] \neq Y[j] \quad L(i,j) = \max \{L(i-1,j), L(i,j-1)\}
  \end{align*}
  \]
• LCSLength(X[1..m], Y[1..n])
  \[
  L = \text{zero array}(0..m, 0..n)
  \]
  for i := 1..m
    for j := 1..n
      if X[i] = Y[j]
        \[ L[i,j] := L[i-1,j-1] + 1 \]
      else
        \[ L[i,j] := \max(L[i,j-1], L[i-1,j]) \]

  return L[m,n]

• Running time = O(mn)
Longest common subsequence

- If we want to output the sequence, we record which rule was used at each point:
  - `↖` if the last symbols match
  - `←` if we are dropping last symbol of X
  - `↑` if we are dropping last symbol of Y

Then we can reconstruct the sequence backwards.

<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>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</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>M</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>J</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>U</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
Let us plan Bob's next $L$ years.

He has $w$, and every year makes $w$

At the beginning of each year, he must decide how much to consume, rest is saved.

- Savings earn interest $(1+\rho)$ (round to integer)
- 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
Life can only be understood backwards; but it must be lived forwards.

Soren Kierkegaard
• Subproblems and recursion

• $U[k,i] := \text{utility for years } i, i+1, \ldots, L \text{ if at beginning of year } i$
  have $k$. Note $k$ integer $\leq M := wL (1 + \rho)^L$

• $U[k,L] := ?$

  How much should Bob consume in his last year of life?
Subproblems and recursion

- $U[k,i] := \text{utility for years } i, i+1, \ldots, L \text{ if at beginning of year } i$
  have $k$. Note $k$ integer $\leq M := wL (1 + \rho)^L$

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

- $U[k,i] := \text{What recursion for } i < L$?
• Subproblems and recursion

• $U[k,i] := \text{utility for years } i, i+1, \ldots, L \text{ if at beginning of year } i$
  have $k$. Note $k$ integer $\leq M := wL (1 + \rho)^L$

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

• $U[k,i] := \max_{0 \leq c \leq M} \log(c) + U[(k - c)(1+\rho) + w, i+1]$
  Consumption $= \arg\max$

$\Rightarrow$ Dynamic programming algorithm running in time $O(LM^2)$
• Slightly more realism

• With probability $q$ Bob earns interest rate $(1 + \rho)$

• With probability $1 - q$ Bob loses money rate $(1 - \rho)$

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

• $U[k,L] := \log(k)$

• $U[k,i] := \max_{0 \leq c \leq M} \log(c) + \$
• Slightly more realism

• With probability $q$ Bob earns interest rate $(1+\rho)$

• With probability $1-q$ Bob loses money rate $(1-\rho)$

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

• $U[k,L] := \log(k)$

• $U[k,i] := \max_{0 \leq c \leq M} \log (c) + q \ U[(k - c)(1+\rho) + w, i+1] + (1-q) \ U[(k - c)(1-\rho) + w, i+1]$
Subset sum problem

- Problem: Input integers \( w_1 \), \( w_2 \) \ldots , \( w_n \), \( t \)
- Output: Number of (subsets) \( x \in \{0,1\}^n : \sum_{i=1}^{n} w_i \cdot x_i = t \)

Example:

\( n = 3, \ t = 12 \)
\( w = \{2, 3, 5, 7, 10\} \)
\( t = 10+2, 7+5, 7+3+2 \)

Output = 3
Subset sum problem

- Problem: Input integers $w_1, w_2 \ldots, w_n, t$
- Output: Number of (subsets) $x \in \{0,1\}^n : \sum_{i=1}^{n} w_i \cdot x_i = t$

Arriving at subproblems and recursion

To get to $t$ we can either:

- use $w_n$ then need to get to $t - w_n$ using $w_1, w_2 \ldots, w_{n-1}$
- or not then need to get to $t$ using $w_1, w_2 \ldots, w_{n-1}$
Subset sum problem

Problem: Input integers $w_1, w_2, \ldots, w_n, t$

Output: Number of (subsets) $x \in \{0,1\}^n : \sum_{i=1}^{n} w_i \cdot x_i = t$

Subproblems and recursion:

- $S(i,s) :=$ number of $x \in \{0,1\}^i$ such that $\sum_{j=1}^{i} w_j \cdot x_j = s$
- Recursion: $S(i,s) = S(i-1,s) + S(i-1,s-w_i)$

There are only $tn$ different subproblems $S(i,s)$
(Don’t need to consider sums larger than $t$)

NOTE: Assuming weights are positive: $w_i \geq 0$ for all $i$
. Problem: Input integers \( w_1, w_2, \ldots, w_n, t \)

. Output: Number of (subsets) \( x \in \{0,1\}^n : \sum_{i=1}^n w_i \cdot x_i = t \)

<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>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Fill first column
- (for \( i = 2 \ldots n \))
- (for \( s = 0 \ldots t \))

Algorithm
Problem: Input integers \( w_1, w_2, \ldots, w_n, t \)
Output: Number of (subsets) \( x \in \{0,1\}^n : \sum_{i=1}^{n} w_i \cdot x_i = t \)

Algorithm

- Fill first column
- (for \( i = 2 \ldots n \))
  - (for \( s = 0 \ldots t \))
    - \( S(i,s) = S(i-1,s) + S(i-1,s-w_i) \)
- \( T(n) = \)
Problem: Input integers $w_1, w_2, \ldots, w_n$, $t$

Output: Number of (subsets) $x \in \{0,1\}^n : \sum_{i=1}^{n} w_i \cdot x_i = t$

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

$T(n) = O(tn)$


- Problem: Input integers \( w_1, w_2, \ldots, w_n, t \)
- Output: Number of (subsets) \( x \in \{0,1\}^n : \sum_{i=1}^{n} w_i \cdot x_i = t \)

\[
\begin{array}{cccc}
& & & \\
& 1 & & \\
& & & \\
1 & 2 & & \\
1 & 1 & & \\
\end{array}
\]

- Fill first column
- (for \( i = 2 \ldots n \))
  - (for \( s = 0 \ldots t \))
    \[
    S(i,s) = S(i-1,s) + S(i-1,s-w_i)
    \]
- Space: Trivial: \( O(tn) \) Better: ??
• Problem: Input integers $w_1, w_2 \ldots, w_n, t$
• Output: Number of (subsets) $x \in \{0,1\}^n : \sum_{i=1}^{n} w_i \cdot x_i = t$

$$S(i,s) = S(i-1,s) + S(i-1,s-w_i)$$

- Fill first column
- (for $i = 2 \ldots n$)
  (for $s = 0 \ldots t$)
  $$S(i,s) = S(i-1,s) + S(i-1,s-w_i)$$
- Space: $O(t)$, just keep two columns
Example:
n = 3, t = 12
w = \{2, 3, 5, 7, 10\}
t = 10+2, 7+5, 7+3+2

Output = 3
Greedy Algorithms
Dynamic programming requires solving all subproblems, leads to algorithms with running time usually $n^2$ or $n^3$

Sometimes, greedy is faster.

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.

\[ A := \{a_1, a_2, ..., a_n\} \]

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 = ?
Example:

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

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

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

A maximal set of compatible activities = ?
Example:

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:

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:

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

activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := a[1]
    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 $f[i]$;
    $n := \text{length}[A]$;
    $S := a[1]$;
    $i := 1$;
    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 \( f[i] \);
  \( n := \text{length}[A] \);
  \( S := a[1] \);
  \( i := 1 \);
  \( n := 11 \)
  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[1]
  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[1]
    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[1];
    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]?

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

Example:
\( S := \{a_1, a_4\} \)
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := a[1]
    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) {
  sort A increasingly according to f[i];
  n := length[A];
  S := a[1]
  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\}

s[5] ≥ f[4]?
activity-selection(A) {
    sort A increasingly according to f[i];
    n:= length[A];
    S:= a[1]
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[1]
    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[1]
  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[1];
    i := 1;
    for (m=2; m ≤ n; m++)
      if (s[m] ≥ f[i]) {
        Add a[i] to S;
        s[i] := m;
      }
    return S;
}
activity-selection(A) {
    sort A increasingly according to f[i];
    n:= length[A];
    S:= a[1]
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[1]
    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[1]
    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 $f[i]$;
    $n := \text{length}[A]$;
    $S := a[1]$
    $i := 1$;
    for ($m = 2$; $m \leq n$; $m++$)
        if ($s[m] \geq f[i]$) {
            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 := length[A];
    S := a[1]
    i := 1;
    for (m = 2; m ≤ n; m++)
        if (s[m] ≥ f[i]) {
            Add a[i] to S;
            i := m;
        }
    return S;
}
```
Example:

$S := \{a_1, a_4, a_8\}$

$s[9] \geq f[8]$ ?

\[\begin{array}{cccccccccccc}
\text{a}_i & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\text{s} & 1 & 3 & 0 & 5 & 3 & 5 & 6 & 8 & 8 & 2 & 12 \\
\text{f}_i & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\end{array}\]
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[1];
    i := 1;
    for (m=2; m ≤ n; m++)
        if \( s[m] \geq f[i] \) {
            Add \( a[i] \) to \( S \);
            i := m;
        }
    return S;
}
```

```
<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</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:
$S := \{a_1, a_4, a_8\}$

$s[10] \geq f[8]$ ?

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

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

\[ s[10] < f[8] \]
Example:

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


```plaintext
activity-selection(A) {
    sort A increasingly according to f[i];
    n := length[A];
    S := a[1]
    i := 1;
    for (m = 2; m \leq 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[1]
    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}\}

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[1]
  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}\}
```

\[ m = 12, \quad n = 11 \]

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

For activity selection, we have:
```
activity-selection(A) {
    sort A increasingly according to \( f[i] \);
    n:= length[A];
    S:= a[1]
    i:=1;
    for (m=2; m ≤ n; m++)
        if (s[m] ≥ f[i]) {
            Add a[i] to S;
            i :=m;
        }
    return S;
}
```
Deleted scenes
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 \( v_i / w_i \);
\( x[1..n] = 0; \) //\( x[i] \) = amount of \( i \) to be taken
weight = 0; \( i = 1; \)

while (weight < \( W \) and \( i \leq n \))
  if weight + \( w[i] \) ≤ \( W \) {
    \( x[i] = 1; \)
    weight += \( w[i] \);
    \( i++ \)
  } else {
    \( x[i] = (W - \text{weight}) / \text{w}[i]; \)
    weight = \( W \);
  }
return \( x \)
Fractional knapsack \((W, S)\)

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

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

weight = 0; \(i = 1\);

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

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

\[ x[i] = 1; \]

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

\[ i++ \]

\[
\} \text{ else } \{
\]

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

\[ \text{weight} = W; \]

\[
\}
\]

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

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

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

\(x[i] = 0;\)

Weight = 0; \(i = 1;\)

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

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

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

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

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

else 

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

weight = \(W;\);} 

return \(x\)

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