CS 341: ALGORITHMS

Lecture 12: dynamic programming

Slides by Trevor Brown (some material from Doug Stinson)

trevor.brown@uwaterloo.ca
(http://tbrown.pro)
Designing Dynamic Programming Algorithms for Optimization Problems

Optimal Structure
Examine the structure of an optimal solution to a problem instance $I$, and determine if an optimal solution for $I$ can be expressed in terms of optimal solutions to certain subproblems of $I$.

Define Subproblems
Define a set of subproblems $S(I)$ of the instance $I$, the solution of which enables the optimal solution of $I$ to be computed. $I$ will be the last or largest instance in the set $S(I)$. 
Recurrence Relation

Derive a recurrence relation on the optimal solutions to the instances in $S(I)$. This recurrence relation should be completely specified in terms of optimal solutions to (smaller) instances in $S(I)$ and/or base cases.

Compute Optimal Solutions

Compute the optimal solutions to all the instances in $S(I)$. Compute these solutions using the recurrence relation in a bottom-up fashion, filling in a table of values containing these optimal solutions. Whenever a particular table entry is filled in using the recurrence relation, the optimal solutions of relevant subproblems can be looked up in the table (they have been computed already). The final table entry is the solution to $I$. 
Combining solutions to subproblems is easy in this case: just add (+) 

Express (optimal) solution to problem $i$ in terms of (optimal) solutions to problems $i - 1$ and $i - 2$. 

Base cases are critical: they can completely change the final solution. 

Recurrence relation that leads to this DP alg: 

$$f(n) = \begin{cases} 
    f(n - 1) + f(n - 2) : i \geq 2 \\
    1 : i = 1 \\
    0 : i = 0 
\end{cases}$$

Note that this recurrence fundamentally does not have anything to do with DP. Recurrences can be implemented in many ways. Building a table of results bottom-up is what makes this DP.
DP SOLUTION TO
0-1 KNAPSACK
Suppose the optimal solution \( O \) does not include this item. Then with the \( O \) must achieve the best possible value using only items 1-3.

Subproblem: output max value for \( \leq 7\)kg out of these three items

Problem: output maximum value one can get from taking \( \leq 7\)kg, out of these four items.

This is a smaller subproblem: reduced # of items

Goal: create recurrence relation to describe optimal solution in terms of subproblems

Let \( P[i, m] = \) maximum profit using any subset of the items \( 1..i \), with weight limit \( m \)

Note: \( P[4, 7] \) is the optimal profit

If \( O \) does not include the camera, then \( P[4, 7] = \) best we can do with the first three items and weight limit 7kg

That is, \( P[4, 7] = P[3, 7] \)
Suppose the optimal solution $O$ includes this subproblem:

**Subproblem:** output max value for $\leq 6\text{kg}$ out of these three items.

Problem: output maximum value one can get from taking $\leq 7\text{kg}$, out of these four items.

This is a smaller subproblem: reduced weight and # of items.

Recall: $P[i, m] = \text{maximum profit using any subset of the items 1..i, with weight limit } m$.

If $O$ includes the camera, then $P[4, 7] = p_4 + \text{best we can do with the first three items and weight limit } 7\text{kg} - w_4 = 6\text{kg}$.


How to evaluate both possibilities: in & not in $O$?

Then with the remaining $7\text{kg} - w_4 = 6\text{kg}$, and items 1-3, $O$ must achieve the best possible value.
Recall: \( P[i, m] = \text{maximum profit using any subset of the items } 1..i, \text{ with weight limit } m \)

<table>
<thead>
<tr>
<th>If O does not include the camera, then</th>
<th>In general:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P[4, 7] = \text{best we can do with the first three items and weight limit } 7\text{kg} )</td>
<td>( P[4, 7] = P[3, 7] )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>If O includes the camera, then</th>
<th>In general:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P[4, 7] = p_4 + \text{best we can do with the first three items and weight limit } 7\text{kg} - w_4 = 6\text{kg} )</td>
<td>( P[4, 7] = p_4 + P[3, 7-w_4] )</td>
</tr>
</tbody>
</table>

Try both and take the better result! (How?)

<table>
<thead>
<tr>
<th>( P[4, 7] = \max{ )</th>
<th>( P[4, 7] = P[3, 7] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P[3, 7], )</td>
<td>( P[3, 7] )</td>
</tr>
<tr>
<td>( p_4 + P[3, 7-w_4] } )</td>
<td>( p_i + P[i-1, m-w_i] )</td>
</tr>
</tbody>
</table>

\[ P[i, m] = \max\{ P[i-1, m], \]
\[ p_i + P[i-1, m-w_i] \} \]

Note that \( \max\{ P[i-1, m], p_i + P[i-1, m-w_i] \} \) is only valid if \( i \geq 2 \text{ and } m \geq w_i \)

What to do when \( i = 1 \text{ or } m < w_i \)? These are special cases.
Special case 3: \( i = 1 \) and \( m < w_i \)

Since \( m < w_i \), we cannot carry item \( i \).

So, \( P[i, m] = P[i-1, m] \).

Special case 2: \( i = 1 \) and \( m \geq w_i \)

Since \( i \leq 1 \), we can only use item 1.
Since \( m \geq w_i \), we can carry item 1.
So, \( P[i, m] = p_i \).

Special case 1: \( i \geq 2 \) and \( m < w_i \)

Since \( m < w_i \), we cannot carry item \( i \).

So, \( P[i, m] = 0 \).

General case: \( i \geq 2 \) and \( m \geq w_i \)

Since \( m \geq w_i \), we can carry item \( i \).
\[
P[i, m] = \max\{P[i-1, m], p_i + P[i-1, m-w_i]\}
\]

Recurrence Relation:

\[
P[i, m] = \begin{cases} 
\max\{P[i-1, m], p_i + P[i-1, m-w_i]\} & \text{if } i \geq 2, m \geq w_i \\
p_1 & \text{if } i = 1, m \geq w_1 \\
0 & \text{if } i = 1, m < w_1.
\end{cases}
\]
FILLING THE ARRAY:

No data dependencies on any other array cells.

\( i \)-axis (can use items in 1..i)

\( m \)-axis (remaining weight limit)

\[
P[i, m] = \begin{cases} 
\max\{P[i-1, m], p_i + P[i-1, m-w_i]\} & \text{if } i \geq 2, \ m \geq w_i \\
\max\{P[i-1, m], P[i-1, m-w_i]\} & \text{if } i \geq 2, \ m < w_i \\
p_i & \text{if } i = 1, \ m \geq w_1 \\
0 & \text{if } i = 1, \ m < w_1. 
\end{cases}
\]

Suppose item 1 does not fit until \textbf{this} \( m \) value.
**FILLING THE ARRAY:**

**Data dependency:**
- Need cell above to be computed already.

- **Suppose** $m < w_2$ from here

- **$i$-axis** (can use items in $1..i$)

**$i$-axis**
- 

**$m$-axis** (remaining weight limit)

- **$m$-axis** (remaining weight limit)
FILLING THE ARRAY:

\[ P[i, m] = \begin{cases} \max\{P[i - 1, m], p_i + P[i - 1, m - w_i]\} & \text{if } i \geq 2, m \geq w_i \\ P[i - 1, m] & \text{if } i \geq 2, m < w_i \\ p_1 & \text{if } i = 1, m \geq w_1 \\ 0 & \text{if } i = 1, m < w_1. \end{cases} \]

**i-axis** (can use items in 1..i)

**m-axis** (remaining weight limit)

Where is slot \([i - 1, m - w_i]\)?

Consider this entry. Suppose \(m \geq w_2\).

Entry \([i - 1, m]\)

Data dependency: need this to be computed already

So, what value should be stored in this entry?

\[ \max\{p_1, p_2 + 0\} \]
FILLING THE ARRAY:

\[ P[i, m] = \begin{cases} 
\max\{P[i-1, m], p_i + P[i-1, m-w_i]\} & \text{if } i \geq 2, \ m \geq w_i \\
\max\{p_i, p_2 + 0\} & \text{if } i = 2, \ m < w_i \\
p_1 & \text{if } i = 1, \ m \geq w_1 \\
0 & \text{if } i = 1, \ m < w_1. 
\end{cases} \]

- **i-axis** (can use items in 1..i)
- **m-axis** (remaining weight limit)

We only ever look at the previous row!

To satisfy data dependencies, we can fill entries in the order: for \( (i = 1..n) \), for \( (m = 1..M) \)

Depending how many zeros we have in the top row, and how far back we’re looking, might start to get cells containing \( \max\{p_1, p_2 + p_1\} \)

Would the following fill-order work? for \( (i = 1..n) \), for \( (m = M..1) \)
Suppose we have profits 1, 2, 3, 5, 7, 10, weights 2, 3, 5, 8, 13, 16, and capacity 30.

The following table is computed:

\[
P[i, m] = \begin{cases} 
\max\{P[i-1, m], p_i + P[i-1, m-w_i]\} & \text{if } i \geq 2, \ m \geq w_i \\
P[i-1, m] & \text{if } i \geq 2, \ m < w_i \\
0 & \text{if } i = 1, \ m \geq w_i 
\end{cases}
\]

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 |   |   | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 2 |   | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 3 |   | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 4 |   |   | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 5 |   |   |   |   | 1 | 2 | 3 | 3 | 3 | 3 | 3 | 3 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 6 |   |   |   |   |   |   |   |   |   |   | 1 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

\[P[3, 16] = \] ? What do you think?
Recall: To satisfy data dependencies, we can fill entries in the order:
for \((i = 1 \ldots n)\), for \((m = 1 \ldots M)\)

\[
P[i, m] = \begin{cases} 
\max\{P[i-1,m], p_i + P[i-1,m-w_i]\} & \text{if } i \geq 2, \ m \geq w_i \\
P[i-1,m] & \text{if } i \geq 2, \ m < w_i \\
p_1 & \text{if } i = 1, \ m \geq w_1 \\
0 & \text{if } i = 1, \ m < w_1 
\end{cases}
\]

**Algorithm:** 0-1 Knapsack\((p_1, \ldots, p_n, w_1, \ldots, w_n, M)\)

```python
for m ← 0 to M
    if m ≥ w_1
        then P[1, m] ← p_1
    else P[1, m] ← 0
for i ← 2 to n
    for m ← 0 to M
        if m < w_i
            do { then P[i, m] ← P[i-1, m]
                else P[i, m] ← \max\{P[i-1, m-w_i] + p_i, P[i-1, m]\}
return (P[n, M]);
```

Fill first row \((i=1)\)

Fill remaining rows in our specific order

Read & return optimal profit

How about the optimal items?
The optimal solution is computed by tracing back through the table.

For the previous example, consisting of profits 1, 2, 3, 5, 7, 10, weights 2, 3, 5, 8, 13, 16, and capacity 30, the optimal solution is

<table>
<thead>
<tr>
<th>Items you can take</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

weight limit remaining

<table>
<thead>
<tr>
<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>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
<th>28</th>
<th>29</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8 > 6 so O must take item 4

Same profit using items 1..4 or 1..5. So, there exists an optimal solution O that does not use item 5! Consider O.

Best profit for remaining items + weight

18 > 17, so any optimal solution must take item 6 remaining weight = 14

Exercise: continue, and determine which other items are in O

Start at optimal profit
Algorithm: \text{ComputeOptimalKnapsack}(p_1, \ldots, p_n, w_1, \ldots, w_n, M, P)

\begin{align*}
m & \leftarrow M \\
p & \leftarrow P[n, M] \\
\text{for } i \leftarrow n \text{ downto } 2 \\
& \begin{cases}
\text{if } p = P[i - 1, m] \\
& \text{then } x_i \leftarrow 0 \\
& \text{do } \begin{cases}
& x_i \leftarrow 1 \\
& \text{else } \begin{cases}
& p \leftarrow p - p_i \\
& m \leftarrow m - w_i
\end{cases}
\end{cases}
\end{cases} \\
& \text{if } p = 0 \\
& \text{then } x_1 \leftarrow 0 \\
& \text{else } x_1 \leftarrow 1 \\
\text{return } (X);
\end{align*}
Complexity of the Algorithm

Suppose we assume the unit cost model, so additions / subtractions take time $O(1)$.

The complexity to construct the table is $\Theta(nM)$

Is this a polynomial-time algorithm, as a function of the size of the problem instance?

We have

$$\text{size}(I) = \log_2 M + \sum_{i=1}^{n} \log_2 w_i + \sum_{i=1}^{n} \log_2 p_i.$$ 

Note in particular that $M$ is exponentially large compared to $\log_2 M$. So constructing the table is not a polynomial-time algorithm, even in the unit cost model.

What would the complexity of a recursive algorithm be?

So the DP alg is faster when there are many item types, but small profit values.

Huge $n$ is fine, but $M$ should be in poly$(n)$ to get an asymptotic improvement.

DP takes $\Theta(nM)$ time, which could be $\Theta(n2^n)$ for huge $M$.

$n$ must be very small.

A recursive algorithm would take $\sim \Theta(2^n)$ time.
DP SOLUTION TO COIN CHANGING
There is a denomination with unit value!

In 0-1 knapsack, we only considered two subproblems in our recurrence: taking an item, or not.

Here we can do more than use a coin denomination or not.

What do you think?
Let $N[i, t]$ denote the optimal solution to the subproblem consisting of the first $i$ coin denominations $d_1, \ldots, d_i$ and target sum $t$.

Exploring: some sensible base case(s)?

<table>
<thead>
<tr>
<th>General case:</th>
</tr>
</thead>
<tbody>
<tr>
<td>What are the different ways we could use coin denomination $d_i$?</td>
</tr>
<tr>
<td>What subproblems / solutions should we use?</td>
</tr>
</tbody>
</table>

Final recurrence relation

Also $N[i, 0] = 0$ for all $i$.
FILLING THE ARRAY $N[1 \ldots n, 1 \ldots T]$:

No data dependencies on any other array cells.

$i$-axis (coin type)

(recall: $N[i, t]$ uses coin types $1 \ldots i$)

$$N[i, t] = \begin{cases} 
\min\{j + N[i-1, t-jd_i] : 0 < j < |t/d_i|\} & \text{if } i > 2 \\
\begin{array}{c}
\text{OR } t = 0 \\
t
\end{array} & \text{if } i = 1.
\end{cases}$$
FILLING THE ARRAY \(N[1 \ldots n, 1 \ldots T]\):

No data dependencies on any other array cells.

\(i\)-axis (coin type)

(recall: \(N[i, t]\) uses coin types 1..\(i\))

\(t\)-axis (target sum remaining)

\[
N[i, t] = \begin{cases} 
\min\{j + N[i-1, t - jd_i] : 0 \leq j \leq \lfloor t/d_i \rfloor\} & \text{if } i \geq 2 \\
 t & \text{if } i = 1, \ OR \ t = 0
\end{cases}
\]
FILLING THE ARRAY $N[1 \ldots n, 1 \ldots T]$: 

$$N[i, t] = \begin{cases} \min\{j + N[i-1, t - jd_i] : 0 \leq j \leq \lfloor t/d_i \rfloor\} & \text{if } i \geq 2 \\ t & \text{if } i = 1. \end{cases}$$

OR $t = 0$

$i$-axis (coin type)  
(recall: $N[i, t]$ uses coin types 1..$i$)

$t$-axis (target sum remaining)

We only look at the previous $i$-row!

It is sufficient to fill: row $i=1$ (base case), then for $(i = 2 \ldots n)$, for $(t = 0 \ldots T)$

Consider cell $N[i, t]$
Algorithm: Coin Changing\( (d_1, \ldots, d_n) \)

comment: \( d_1 = 1 \)

for \( t \leftarrow 0 \) to \( T \)
  do \( \{N[1, t] \leftarrow t; A[1, t] \leftarrow t\} \)

for \( i \leftarrow 2 \) to \( n \)
  do \( \{
  \text{for } t \leftarrow 0 \text{ to } T
  \text{ do } \{
  \text{do } \{ \text{for } j \leftarrow 1 \text{ to } \lfloor t/d_i \rfloor \}
  \text{ if } j + N[i-1, t - jd_i] < N[i, t] \text{ then } \{
  N[i, t] \leftarrow j + N[i-1, t - jd_i]
  A[i, t] \leftarrow j
  \}
  \}
  \}
  \}
  \}
  \}

return \((N[n, T])\)

Base case (row \( i=1 \)): using unit-valued coins

Additionally, we maintain \( A[i, t] = \# \text{ of coins of type } d_i \text{ used in } N[i, t] \)

General case \((i \geq 2)\): using other coin types

Compute \( \min\{\ldots\} \) over 
\[
\begin{align*}
j = 0 \ldots \left\lfloor \frac{t}{d_i} \right\rfloor
\end{align*}
\]

Complexity?
Recall: an algorithm is "polynomial time" only if its runtime is polynomial in the size (number of bits) in the input!

The input size $s$ is: \(\text{bits}(d_1) + \text{bits}(d_2) + ... + \text{bits}(d_n) + \text{bits}(T)\).

For simplicity, we assume unit cost operations on the $d_i$ inputs (read, write, add, multiply, compare, etc.). Otherwise, we will have lots of $\log_2 d_i$ terms in our final complexity.

Runtime: $O\left(\sum_{i=2}^{n} \frac{T}{d_i} \right) = O\left(\sum_{i=2}^{n} \frac{1}{d_i} \sum_{t=0}^{T} t \right) = O\left(\sum_{i=2}^{n} \frac{1}{d_i} \frac{T(T+1)}{2} \right) = O(DT^2)$ where $D = \sum_{i=2}^{n} \frac{1}{d_i} < n$.

It takes $\lfloor \log_2 d_i \rfloor$ bits to store each $d_i$. Assuming $d_i \leq T$ (otherwise $d_i$ cannot be used), we have $\lfloor \log_2 d_i \rfloor \in O(\log T)$.

And, it takes $\lfloor \log_2 T \rfloor$ bits to store $T$.

So, $s \in \sum_{i=1}^{n} O(\log T) + \lfloor \log_2 T \rfloor = \Theta(n \log T)$.

So, for example, if $n = 1$, then $s \in \Theta(\log T)$, which means $T \in \Theta(2^s)$. In this case, $T$ is exponential in the input size $s$, and hence so is the running time!

So, large $n$ and small $T$ is where this DP solution shines!
Computing the Optimal Set of Coins

We trace back through the table to compute the optimal set of coins. There are two possible approaches:

Recompute the relevant table entries $N[i, t]$ during the traceback
Store relevant extra information, while the table $N[i, t]$ is being constructed, in another table $A[i, t]$.

Suppose we follow the second approach.

The $A[i, t]$ values make it easy to determine number of coins of each denomination in the optimal solution $N[i, T]$.

This is kind of similar to 0-1 Knapsack.
**Algorithm:** \( \text{Trace}(d_1, \ldots, d_n, T, A) \)

\[
t \leftarrow T \\
\text{for } i \leftarrow n \text{ downto } 1 \\
\quad \text{do } \begin{cases} 
    a_i \leftarrow A[i, t] \\
    t \leftarrow t - a_i d_i
    \end{cases}
\]

\text{return } (a_1, \ldots, a_n)

Exercise for home: compute the correct output \textbf{without} using \( A[i,t] \)

Note that we only need the \( A[i, t] \) values to do the traceback.