Dynamic Programming

Chapters 6 of Dasgupta et al.

Outline

• Intro
• Counting combinations
• 0-1 Knapsack (section 6.4)
• Longest common subsequence
• Later
  – Bellman-Ford (single source shortest path)
  – Floyd-Warshall (all pairs shortest path) (section 8.2)
Two key ingredients

- Two key ingredients for an optimization problem to be suitable for a dynamic programming solution

1. optimal substructures
2. overlapping subproblems

Each substructure is optimal (principle of optimality)
Subproblems are dependent
(otherwise, a divide-and-conquer approach is the choice)
Three basic components

- The development of a dynamic programming algorithm has three basic components
  - a recurrence relation (for defining the value/cost of an optimal solution)
  - a tabular computation (for computing the value of an optimal solution)
  - a trace-back procedure (for delivering an optimal solution)

Counting combinations
Counting combinations

To choose \( r \) things out of \( n \), either

- Choose the first item. Then we must choose the remaining \( r-1 \) items from the other \( n-1 \) items. Or
- Don’t choose the first item. Then we must choose the \( r \) items from the other \( n-1 \) items.

Therefore,

\[
\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}
\]

Counting combinations: D&C

- A simple divide & conquer algorithm for finding the number of combinations of \( n \) things chosen \( r \) at a time

```python
def choose(n, r):
    if r == 0 or n == r:
        return 1
    else:
        return choose(n-1, r-1) + choose(n-1, r)
```
Counting combinations: D&C

**Correctness Proof:** A simple induction on $n$.

**Analysis:** Let $T(n)$ be the worst case running time of choose($n$, $r$) over all possible values of $r$.

Then,

$$T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  2T(n-1) + d & \text{otherwise}
\end{cases}$$

for some constants $c, d$.

---

Counting combinations: D&C

\[
T(n) = 2T(n-1) + d \\
= 2(2T(n-2) + d) + d \\
= 4T(n-2) + 2d + d \\
= 4(2T(n-3) + 2d) + 2 + d \\
= 8T(n-3) + 4d + 2d + d \\
= 2^iT(n - i) + d \sum_{j=0}^{i-1} 2^j \\
= 2^{n-1}T(1) + d \sum_{j=0}^{n-2} 2^j \\
= (c + d)2^{n-1} - d
\]

Hence, $T(n) = \Theta(2^n)$. 

---

Stefano Lonardi
Counting combinations: Example

The problem is, the algorithm solves the same subproblems over and over again!
Counting Combinations

- Generate the Pascal’s triangle $T[0..n, 0..r]$ where $T[i,j]$ holds $\binom{i}{j}$

```python
def choose(n, r):
    T = {}
    for i in range(n-r+1):
        T[i,0] = 1
    for i in range(r+1):
        T[i,i] = 1
    for j in range(1,r+1):
        for i in range(j+1,n-r+j+1):
            T[i,j] = T[i-1,j-1] + T[i-1,j]
    return T[n,r]
```

Initialization

![Diagram of Pascal's triangle initialization](image-url)
General Rule

To fill in $T[i, j]$, we need $T[i - 1, j - 1]$ and $T[i - 1, j]$ to be already filled in.

Filling in the Table

Fill in the columns from left to right. Fill each of the columns from top to bottom.

Numbers show the order in which the entries are filled in.
Example

Analysis

How many table entries are filled in?

\[(n-r+1)(r+1) = nr + n - r^2 + 1 \leq n(r+1) + 1\]

Each entry takes time \(O(1)\), so total time required is \(O(n^2)\).

This is much better than \(O(2^n)\).

Space: naive, \(O(nr)\). Smart, \(O(r)\).
Dynamic Programming

When divide and conquer generates a large number of identical subproblems, recursion is too expensive.

Instead, store solutions to subproblems in a table.

This technique is called dynamic programming.

Identification:
- devise divide-and-conquer algorithm
- analyze — running time is exponential
- same subproblems solved many times
Dynamic Programming Construction

- take part of divide-and-conquer algorithm that does the “conquer” part and replace recursive calls with table lookups
- instead of returning a value, record it in a table entry
- use base of divide-and-conquer to fill in start of table
- devise “look-up template”
- devise for-loops that fill the table using “look-up template”
0-1 knapsack

The Knapsack Problem

- A thief robbing a store finds \( n \) items
- The \( i \)th item is worth \( b_i \) and weighs \( w_i \) pounds
- Thief’s knapsack can carry at most \( W \) pounds
- Variables \( b_i \), \( w_i \) and \( W \) are integers
- **Problem**: What items to select to maximize profit?
The 0-1 Knapsack Problem

- Each item must be either taken or left behind (a binary choice of 0 or 1)
- Exhibits *optimal substructure* property (for the same reason as for the fractional)
- 0-1 knapsack problem however *cannot* be solved by a greedy strategy
- Can be solved (less) efficiently by *dynamic programming*

---

0-1 Knapsack - Greedy Strategy

- The greedy choice property does *not* hold
0-1 Knapsack Problem

• Let $x_i=1$ denote item $i$ is in the knapsack and $x_i=0$ denote it is not in the knapsack
• Problem stated formally as follows

$$\text{maximize } \sum_{i=1}^{n} b_i x_i \quad \text{(total profit)}$$
$$\text{subject to } \sum_{i=1}^{n} w_i x_i \leq W \quad \text{(weight constraint)}$$

Define the problem recursively ...

• Consider the first item $i=1$
  1. If it is selected (put in the knapsack)
     $$\text{maximize } \sum_{i=2}^{n} b_i x_i \text{ subject to } \sum_{i=2}^{n} w_i x_i \leq W - w_i$$
  2. If it is not selected
     $$\text{maximize } \sum_{i=2}^{n} b_i x_i \text{ subject to } \sum_{i=2}^{n} w_i x_i \leq W$$
• Compute both cases, select the better one
Recursive Solution

- Let us define $P[i,k]$ as the maximum profit possible using items \{i, i+1, ..., n\} and residual (knapsack) capacity $k$

- We can define $P[i,k]$ recursively as follows

$$P[i,k] = \begin{cases} 
0 & i = n \land w_n > k \\
b_i & i = n \land w_n \leq k \\
\max \{P[i+1,k], \ b_i + P[i+1,k-w_i]\} & i < n \land w_i > k \\
P[i+1,k] & i < n \land w_i \leq k 
\end{cases}$$
0-1 knapsack (recursive) in Python

```python
def knapsack(items, i, k):
    n = len(items)
    if i == n:
        return b(items[n-1]) if w(items[n-1])<=k else 0
    if w(items[i-1])>k:
        return knapsack(items, i+1, k)
    else:
        return max(knapsack(items, i+1, k),
                    b(items[i-1])+knapsack(items, i+1, k-w(items[i-1])))
```

Remark: i < n

### Recursive Solution

- We can write an algorithm for the recursive solution based on the four cases
- Recursive algorithm will take \(O(2^n)\) time
- Inefficient because \(P[i,k]\) for the same \(i\) and \(k\) will be computed many times
- Example
  - \(n=5, W=10, w=[2, 2, 6, 5, 4], b=[6, 3, 5, 4, 6]\)
Dynamic Programming Solution

- The inefficiency can be eliminated by computing each $P[i,k]$ once and storing the result in a table for future use.
- The table is filled for $i=n,n-1, \ldots,2,1$ in that order for $1 \leq k \leq W$.
- First row (initialization)

<table>
<thead>
<tr>
<th>$k$</th>
<th>0</th>
<th>1</th>
<th>\ldots</th>
<th>$w_{n-1}$</th>
<th>$w_n$</th>
<th>$w_{n+1}$</th>
<th>\ldots</th>
<th>$W$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[n,k]$</td>
<td>0</td>
<td>0</td>
<td>\ldots</td>
<td>0</td>
<td>$b_n$</td>
<td>$b_n$</td>
<td>\ldots</td>
<td>$b_n$</td>
</tr>
</tbody>
</table>
Example

\[ n=5, \ W=10, \ w = [2, 2, 6, 5, \ 4], \ b = [2, 3, 5, 4, \ 6] \]

<table>
<thead>
<tr>
<th>(i)</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>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>4</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>3</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>2</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</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>

\[ P[i,k] = \max \{P[i+1,k], \ b_i + P[i+1,k-w] \} \]
Example

$n=5$, $W=10$, $w = [2, 2, 6, 5, 4]$, $b = [2, 3, 5, 4, 6]$

\[
P[i,k] = \max\{P[i+1,k], \ b_i + P[i+1,k-w_i]\}
\]

Example

$n=5$, $W=10$, $w = [2, 2, 6, 5, 4]$, $b = [2, 3, 5, 4, 6]$

\[
P[i,k] = \max\{P[i+1,k], \ b_i + P[i+1,k-w_i]\}
\]
Example

\( n=5, \ W=10, \ w = [2, 2, 6, 5, 4], \ b = [2, 3, 5, 4, 6] \)

\[
\begin{array}{cccccccccc}
\hline
i/k & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
5 & 0 & 0 & 0 & 0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\
4 & 0 & 0 & 0 & 0 & 6 & 6 & 6 & 6 & 6 & 10 & 10 \\
3 & 0 & 0 & 0 & 0 & 6 & 6 & 6 & 6 & 6 & 10 & 11 \\
2 & 0 & 0 & 3 & 3 & 6 & 6 & 9 & 2 & 9 & 2 & 10 & 11 \\
1 & 0 & 0 & 3 & 3 & 6 & 6 & 9 & 9 & 9 & 11 & 11 & 11 \\
\hline
\end{array}
\]

\[ P[i,k] = \max\{P[i+1,k], \ b_i + P[i+1,k-w_i]\} \]

Example

\( n=5, \ W=10, \ w = [2, 2, 6, 5, 4], \ b = [2, 3, 5, 4, 6] \)

\[
\begin{array}{cccccccccc}
\hline
i/k & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
5 & 0 & 0 & 0 & 0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 \\
4 & 0 & 0 & 0 & 0 & 6 & 6 & 6 & 6 & 6 & 10 & 10 \\
3 & 0 & 0 & 0 & 0 & 6 & 6 & 6 & 6 & 6 & 10 & 11 \\
2 & 0 & 0 & 3 & 3 & 6 & 6 & 9 & 9 & 9 & 10 & 11 \\
1 & 0 & 0 & 3 & 3 & 6 & 6 & 9 & 9 & 9 & 11 & 11 \\
\hline
\end{array}
\]

\[ x = [0,0,1,0,1] \quad x = [1,1,0,0,1] \]
```python
def knapsack(items, w):
    P, n = {k: b(k) for k in items}
    for j in range(w + 1):
        P[n][j] = b(items[n-1]) if w(items[n-1]) <= j else 0
    for i in range(len(items) - 1, -1, -1):
        for j in range(w + 1):
            if w(items[i-1]) > j:
                P[i][j] = P[i + 1][j]
            else:
                P[i][j] = max(P[i + 1][j],
                               b(items[i-1]) + P[i + 1][j - w(items[i-1])])
    return P
```

**Time complexity**

- **Running time**: $O(nW)$

- Technically, this is not a poly-time algorithm

- This class of algorithms is called *pseudo-polynomial*
Longest common subsequence

Longest Common Subsequence

A sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a subsequence of a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$ if $Z$ can be generated by striking out some (or none) elements from $X$.

For example, $\langle b, c, d, b \rangle$ is a subsequence of $\langle a, b, c, a, d, c, a, b \rangle$. 
Longest Common Subsequence

The **longest common subsequence problem** is the problem of finding, for given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, a maximum-length common subsequence of $X$ and $Y$.

For example, given

$X = \text{B D C A B A}$

$Y = \text{A B C B D A B}$

$Z = \text{LCS}(X, Y) = \text{BCBA}$

$X = \text{[B D C A B A]}$

$Y = \text{[A B C B D A B]}$
Longest Common Subsequence

Brute-force search for LCS requires exponentially many steps because if \( m < n \), there are \( \sum_{i=1}^{m} \binom{n}{i} \) candidate subsequences.

Solve this problem by dynamic programming.

The optimal-substructure of LCS

For a sequence \( Z = \langle z_1, z_2, \ldots, z_k \rangle \) and \( i, 1 \leq i \leq k \), let \( Z_i \) denote the length-\( i \) prefix of \( Z \), namely, \( Z_i = \langle z_1, z_2, \ldots, z_i \rangle \).

Optimal Substructure

**Theorem.** Let \( X = \langle x_1, \ldots, x_m \rangle \) and \( Y = \langle y_1, \ldots, y_n \rangle \) be two sequences, and let \( Z = \langle z_1, \ldots, z_k \rangle \) be any LCS of \( X \) and \( Y \).

1. If \( x_m = y_m \), then \( z_k = x_m = y_n \) and \( Z_{k-1} \) is an LCS of \( X_{m-1} \) and \( Y_{n-1} \).
2. If \( x_m \neq y_m \), then \( z_k \neq x_m \) implies that \( Z \) is an LCS of \( X_{m-1} \) and \( Y \).
3. If \( x_m \neq y_m \), then \( z_k \neq y_n \) implies that \( Z \) is an LCS of \( X \) and \( Y_{n-1} \).

**Proof:** omitted
Recursive Formulation

- Define \( c[i, j] = \) length of LCS of \( X_i \) and \( Y_j \)

\[
c[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0, \\
(c[i - 1, j - 1]) + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\
\max(c[i - 1, j], c[i, j - 1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j.
\end{cases}
\]

- We want \( c[m, n] \)
- This gives a recursive algorithm and solves the problem
- But is it efficient?

Example

\[
c[\alpha, \beta] = \begin{cases} 
0 & \text{if } \alpha \text{ empty or } \beta \text{ empty,} \\
(c[\text{prefix } \alpha, \text{prefix } \beta]) + 1 & \text{if end}(\alpha) = \text{end}(\beta), \\
\max(c[\text{prefix } \alpha, \beta], c[\alpha, \text{prefix } \beta]) & \text{if end}(\alpha) \neq \text{end}(\beta).
\end{cases}
\]

\[
c[\text{springtime, printing}]
\]

\[
c[\text{springt, printing}][\text{springtim, printin}]
\]

\[
\text{[springti, printing]}[\text{springtim, printin}][\text{springtime, printi}]
\]

\[
\text{[springt, printing]}[\text{springti, printin}][\text{springtime, printi}][\text{springtime, print}]
\]
\[ c[\alpha,\beta] = \begin{cases} 
0 & \text{if } \alpha \text{ empty or } \beta \text{ empty}, \\
c[\text{prefix } \alpha, \text{prefix } \beta] + 1 & \text{if } \text{end}(\alpha) = \text{end}(\beta), \\
\max(c[\text{prefix } \alpha, \beta], c[\alpha, \text{prefix } \beta]) & \text{if } \text{end}(\alpha) \neq \text{end}(\beta). 
\end{cases} \]

Keep track of \( c[\alpha,\beta] \) in a table of \( nm \) entries

---

**LCS in Python**

```python
def LCS(X, Y):
    c = {}
    for i in range(len(X)+1):
        for j in range(len(Y)+1):
            if i == 0 or j == 0:
                c[i,j] = 0
            elif X[i-1] == Y[j-1]:
                c[i,j] = c[i-1,j-1] + 1
            else:
                c[i,j] = max(c[i-1,j], c[i,j-1])
    #...continues
```

*Remark:* \( c[i,j] \) contains the length of an LCS of \( X[:i] \) and \( Y[:j] \)

*Time:* \( O(mn) \)
Reporting the LCS in Python

#...continued
i, j = len(X), len(Y)
LCS = []
while c[i, j]:
    while c[i, j] == c[i-1, j]:
        i -= 1
    while c[i, j] == c[i, j-1]:
        j -= 1
    i -= 1
    j -= 1
    LCS.append(X[i])
LCS.reverse()
return LCS

Time: \(O(m+n)\)

Longest Common Subsequence

![Diagram of LCS algorithm]
LCS algorithm

- Time complexity $O(nm)$
- Space complexity $O(nm)$
- Space can be reduced to linear by observing that we just need the previous row to compute the current row
- The length of the LCS can be computed easily in linear space

Reading Assignment

- Counting combinations
- 0-1 Knapsack (6.4)
- Longest common subsequence
- Later in the course
  - Bellman-Ford (single source shortest path)
  - Floyd-Warshall (all pairs shortest path)