Warm up

Given access to unlimited quantities of pennies, nickels dimes, and quarters, (worth value 1, 5, 10, 25 respectively), provide an algorithm which gives change for a given value \( x \) using the fewest number of coins.
Change Making

43 cents
Change Making Algorithm

• Given: target value $x$, list of coins $C = [c_1, \ldots, c_n]$
  (in this case $C = [1,5,10,25]$)

• Repeatedly select the largest coin less than the remaining target value:

  
  
  \[
  \text{while}(x > 0)
  
  \begin{align*}
  &\text{let } c = \max(c_i \in \{c_1, \ldots, c_n\} \mid c_i \leq x) \\
  &\text{print } c \\
  &x = x - c
  \end{align*}
  
  \]
Why does this always work?

• If \( x < 5 \), then pennies only
  – 5 pennies can be exchanged for a nickel
    Only case Greedy uses pennies!

• If \( 5 \leq x < 10 \) we must have a nickel
  – 2 nickels can be exchanged for a dime
    Only case Greedy uses nickels!

• If \( 10 \leq x < 25 \) we must have at least 1 dime
  – 3 dimes can be exchanged for a quarter and a nickel
    Only case Greedy uses dimes!

• If \( x \geq 25 \) we must have at least 1 quarter
  Only case Greedy uses quarters!
Today’s Keywords

• Dynamic Programming
• Gerrymandering
• Greedy Algorithms
• Choice Function
• Change Making
CLRS Readings

• Chapter 15
• Chapter 16
Homeworks

• Homework 5 due Wednesday March 27 at 11pm
  – Seam Carving!
  – Dynamic Programming (implementation)
  – Java or Python

• Homework 6 out tonight, due Wednesday April 3 at 11pm
  – Dynamic Programming and Greedy Algorithms
  – Written (using Latex!)
Dynamic Programming

• Requires **Optimal Substructure**
  – Solution to larger problem contains the solutions to smaller ones

• Idea:
  1. Identify recursive structure of the problem
     • What is the “last thing” done?
  2. Select a good order for solving subproblems
     • “Top Down”: Solve each recursively
     • “Bottom Up”: Iteratively solve smallest to largest
  3. Save solution to each subproblem in memory
Generic Top-Down Dynamic Programming Soln

```python
mem = {}
def myDPalgo(problem):
    if mem[problem] not blank:
        return mem[problem]
    if baseCase(problem):
        solution = solve(problem)
        mem[problem] = solution
        return solution
    for subproblem of problem:
        subsolutions.append(myDPalgo(subproblem))
    solution = OptimalSubstructure(subsolutions)
    mem[problem] = solution
    return solution
```
DP Algorithms so far

- $2 \times n$ domino tiling (Fibonacci)
- Log cutting
- Matrix Chaining
- Longest Common Subsequence
- Seam Carving
Domino Tiling

Tile(n):

- Initialize Memory $M$
- $M[0] = 0$
- $M[1] = 0$
- for $i = 0$ to $n$:
  - $M[i] = M[i-1] + M[i-2]$
- return $M[n]$
Log Cutting

Solve Smallest subproblem first

$$Cut(4) = \max \left\{ Cut(3) + P[1], \right.$$

$$\left. \begin{array}{l}
Cut(2) + P[2], \\
Cut(1) + P[3], \\
Cut(0) + P[4]
\end{array} \right\}$$

Cut(i):

<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>
</tr>
</thead>
<tbody>
<tr>
<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>

Length:

<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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Matrix Chaining

\[ \text{Best}(i, j) = \min_{k=1}^{j-1} \left( \text{Best}(i, k) + \text{Best}(k+1, j) + r_i r_{k+1} c_j \right) \]

\[ \text{Best}(i, i) = 0 \]

\[ \text{Best}(1, 6) = \min \left\{ \begin{array}{l}
\text{Best}(1,1) + \text{Best}(2, 6) + r_1 r_2 c_6 \\
\text{Best}(1,2) + \text{Best}(3, 6) + r_1 r_3 c_6 \\
\text{Best}(1,3) + \text{Best}(4, 6) + r_1 r_4 c_6 \\
\text{Best}(1,4) + \text{Best}(5, 6) + r_1 r_5 c_6 \\
\text{Best}(1,5) + \text{Best}(6, 6) + r_1 r_6 c_6 \end{array} \right\} \]
Longest Common Subsequence

\[
LCS(i, j) = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
LCS(i - 1, j - 1) + 1 & \text{if } X[i] = Y[j] \\
\max(LCS(i, j - 1), LCS(i - 1, j)) & \text{otherwise}
\end{cases}
\]

To fill in cell \((i, j)\) we need cells \((i - 1, j - 1)\), \((i - 1, j)\), \((i, j - 1)\)

Fill from Top->Bottom, Left->Right (with any preference)
Supreme Court Associate Justice Anthony Kennedy gave no sign that he has abandoned his view that extreme partisan gerrymandering might violate the Constitution. I Eric Thayer/Getty Images

**Supreme Court eyes partisan gerrymandering**

Anthony Kennedy is seen as the swing vote that could blunt GOP’s map-drawing successes.
Gerrymandering

- Manipulating electoral district boundaries to favor one political party over others
- Coined in an 1812 Political cartoon
- Governor Gerry signed a bill that redistricted Massachusetts to benefit his Democratic-Republican Party
According to the Supreme Court

• Gerrymandering cannot be used to:
  – Disadvantage racial/ethnic/religious groups

• It can be used to:
  – Disadvantage political parties
VA 5th District
Gerrymandering Today

- Computers make it really effective
Gerrymandering Today

• Computers make it really effective
Gerrymandering Today

- Computers make it really effective
How does it work?

- States are broken into precincts
- All precincts have the same size
- We know voting preferences of each precinct
- Group precincts into districts to maximize the number of districts won by my party

Overall: R:217 D:183

R:65 D:35
R:45 D:55
R:60 D:40
R:47 D:53

R:125
R:65 D:35
R:60 D:40
R:65 D:35
R:45 D:55
R:47 D:53

R:92
R:65 D:35
R:60 D:40
R:65 D:35
R:45 D:55
R:47 D:53

R:112
R:65 D:35
R:60 D:40
R:65 D:35
R:45 D:55
R:47 D:53

R:105
R:65 D:35
R:60 D:40
R:65 D:35
R:45 D:55
R:47 D:53
Gerrymandering Problem Statement

• Given:
  – A list of precincts: \( p_1, p_2, \ldots, p_n \) - precincts, \( n \) of them
  – Each containing \( m \) voters

• Output:
  – Districts \( D_1, D_2 \subset \{p_1, p_2, \ldots, p_n\} \)
  – Where \( |D_1| = |D_2| \)
  – \( R(D_1), R(D_2) > \frac{mn}{4} \), \( n \cdot \frac{1}{2} \cdot m \cdot \frac{1}{2} \)
    • \( R(D_i) \) gives number of “Regular Party” voters in \( D_i \)
    • \( R(D_i) > \frac{mn}{4} \) means \( D_i \) is majority “Regular Party”
  – “failure” if no such solution is possible
Dynamic Programming

• Requires **Optimal Substructure**
  – Solution to larger problem contains the solutions to smaller ones

• **Idea:**
  1. Identify recursive structure of the problem
     • What is the “last thing” done?
  2. Select a good order for solving subproblems
     • “Top Down”: Solve each recursively
     • “Bottom Up”: Iteratively solve smallest to largest
  3. Save solution to each subproblem in memory
Consider the last precinct

If we assign $p_n$ to $D_1$

Valid gerrymandering if:

$$k + 1 = \frac{n}{2},$$

$$x + R(p_n), y > \frac{mn}{4}$$

If we assign $p_n$ to $D_2$

Valid gerrymandering if:

$$n - k = \frac{n}{2},$$

$$x, y + R(p_n) > \frac{mn}{4}$$
Define Recursive Structure

\[ S(j, k, x, y) = \text{True} \] if from among the first \( j \) precincts:
- \( k \) are assigned to \( D_1 \)
- exactly \( x \) vote for R in \( D_1 \)
- exactly \( y \) vote for R in \( D_2 \)

\[ n \times n \times mn \times mn \]

4D Dynamic Programming!!!
Two ways to satisfy $S(j, k, x, y)$:

$S(j, k, x, y) = S(j - 1, k - 1, x - R(p_j), y) \lor S(j - 1, k, x, y - R(p_j))$

- **Case 1:**
  - $D_1$: $k - 1$ precincts
  - $D_2$: $x - R(p_j)$ voters for R

  Then assign $p_j$ to $D_1$

- **Case 2:**
  - $D_1$: $k$ precincts
  - $D_2$: $j - k$ precincts

  Then assign $p_j$ to $D_2$
Final Algorithm

\[ S(j, k, x, y) = S(j - 1, k - 1, x - R(p_j), y) \lor S(j - 1, k, x, y - R(p_j)) \]

Initialize \( S(0,0,0,0) = \text{True} \)
for \( j = 1, \ldots, n \):
    for \( k = 1, \ldots, \min(j, \frac{n}{2}) \):
        for \( x = 0, \ldots, jm \):
            for \( y = 0, \ldots, jm \):
                \[ S(j, k, x, y) = \]
                \[ S(j - 1, k - 1, x - R(p_j), y) \]
                \[ \lor S(j - 1, k, x, y - R(p_j)) \]

Search for True entry at \( S(n, \frac{n}{2}, > \frac{mn}{4}, > \frac{mn}{4}) \)

\( S(j, k, x, y) = \text{True if:} \)
\( \) from among the first \( j \) precincts
\( k \) are assigned to \( D_1 \)
exactly \( x \) vote for R in \( D_1 \)
exactly \( y \) vote for R in \( D_2 \)
Run Time

\[ S(j, k, x, y) = S(j - 1, k - 1, x - R(p_j), y) \lor S(j - 1, k, x, y - R(p_j)) \]

Initialize \( S(0,0,0,0) = \text{True} \)

\[ n \quad \text{for } j = 1, \ldots, n: \]
\[ \frac{n}{2} \quad \text{for } k = 1, \ldots, \min(j, \frac{n}{2}): \]
\[ nm \quad \text{for } x = 0, \ldots, jm: \]
\[ nm \quad \text{for } y = 0, \ldots, jm: \]
\[ S(j, k, x, y) = \]
\[ S(j - 1, k - 1, x - R(p_j), y) \]
\[ \lor S(j - 1, k, x, y - R(p_j)) \]

Search for True entry at \( S(n, \frac{n}{2}, > \frac{mn}{4}, > \frac{mn}{4}) \)
$\Theta(n^4m^2)$

- Runtime depends on the *value* of $m$, not *size* of $m$
- Run time is exponential in *size* of input
- Note: Gerrymandering is NP-Complete