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 $C = [c_1, ..., c_n]$ (in this case $C = [1, 5, 10, 25]$)
• Repeatedly select the largest coin less than the remaining target value:

```plaintext
while($x > 0$)
  let $c = \max(c_j \in [c_1, ..., c_n] | c_j \leq x)$
  print $c$
  $x = x - c$
```
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 Chain ing
- Longest Common Subsequence
- Seam Carving

---

**Domino Tiling**

Tile(n):
1. Initialize Memory M
2. $M[0] = 0$
3. $M[1] = 0$
4. for $i = 0$ to $n$:
   - $M[i] = M[i-1] + M[i-2]$
5. return $M[n]$

---

**Log Cutting**

Solve Smallest subproblem first

$$Cut(P) = \max(\begin{array}{c}
Cut(3) + P[1] \\
Cut(2) + P[2] \\
Cut(1) + P[3] \\
Cut(0) + P[4]
\end{array})$$
Matrix Chaining

\[
\begin{array}{cccccccc}
30 & 5 & 35 & 10 & & 30 & 5 & 25 \\
15 & 5 & 15 & 5 & & 15 & 5 & 20 \\
10 & 6 & 20 & 5 & & 10 & 6 & 20 \\
20 & 5 & 20 & 5 & & 20 & 5 & 20 \\
\end{array}
\]

Best(0, 0) = min\left\{ \begin{array}{l}
\text{Best}(0, 0) = 0 \\
\text{Best}(0, 1) = 30^2 \\
\text{Best}(1, 0) = 5 \\
\text{Best}(1, 1) = 30 \\
\text{Best}(2, 0) = 15^2 \\
\text{Best}(2, 1) = 30 \\
\text{Best}(3, 0) = 15 \\
\text{Best}(3, 1) = 30 \\
\text{Best}(4, 0) = 15 \\
\text{Best}(4, 1) = 30 \\
\text{Best}(5, 0) = 20^2 \\
\text{Best}(5, 1) = 20 \\
\text{Best}(6, 0) = 25^2 \\
\text{Best}(6, 1) = 20 \\
\end{array} \right. 
\]

\[
\begin{array}{cccccccc}
0 & 30^2 & 5 & 30 & 15 & 30 & 15 & 20 \\
0 & 25^2 & 15 & 30 & 20 & 20 & 15 & 20 \\
0 & 15^2 & 15 & 30 & 20 & 20 & 15 & 20 \\
0 & 15^2 & 15 & 30 & 20 & 20 & 15 & 20 \\
\end{array}
\]

Longest Common Subsequence

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

\[
\begin{array}{cccccccc}
\text{X} & A & T & C & T & C & A & T \\
\hline
\text{Y} & 0 & 0 & 3 & 5 & 0 & 0 & 0 \\
A & 0 & 0 & 1 & 4 & 0 & 0 & 0 \\
G & 0 & 0 & 1 & 4 & 0 & 0 & 0 \\
C & 0 & 0 & 1 & 4 & 0 & 0 & 0 \\
A & 0 & 0 & 1 & 4 & 0 & 0 & 0 \\
\end{array}
\]

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

Fill from Top-Right, Bottom-Left (with any preference)

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
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

Gerrymandering Problem Statement

• Given:
  – A list of precincts: $P_1, P_2, ..., P_n$ – precincts, $n$ of them
  – Each containing $m$ voters
• Output:
  – Districts $D_1, D_2, ...$ $\subset \{P_1, P_2, ..., P_n\}$ – Where $|D_i| = |D_j|
  – $R(D_i), R(D_j) > \frac{m}{2}$
  – $R(D_i) > \frac{m}{2}$ 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

\[ D_1 \text{ \# precincts} \quad \text{\# voters for R} \]

\[ D_2 \text{ \# precincts} \quad \text{\# voters for R} \]

After assigning the first \( n - 1 \) precincts

If we assign \( p_k \) to \( D_1 \)

If we assign \( p_k \) to \( D_2 \)

Valid gerrymandering if:

\[ k + 1 \quad \text{\# voters for R} > \frac{m}{3} \]

Define Recursive Structure

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

4D Dynamic Programming!!!

Two ways to satisfy \( S(j, k, x, y) \):

\[ D_1 \text{ \# precincts} \quad \text{\# voters for R} \]

\[ D_2 \text{ \# precincts} \quad \text{\# voters for R} \]

\[ D_3 \text{ \# precincts} \quad \text{\# voters for R} \]

\[ D_4 \text{ \# precincts} \quad \text{\# voters for R} \]

Then assign \( p_j \) to \( D_1 \)

Then assign \( p_j \) to \( D_2 \)

Then assign \( p_j \) to \( D_3 \)

Then assign \( p_j \) to \( D_4 \)

\[ S(j, k, x, y) = \left\{ \begin{array}{ll}
S(j-1, k-1, x-R(p_j), y) & \text{OR} \\
S(j-1, k-1, x, y-R(p_j)) & 
\end{array} \right. \]
**Final Algorithm**

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

1. Initialize \( S(0,0,0,0) = True \)
2. for \( j = 1, \ldots, n \):
   1. for \( k = 1, \ldots, \min(j, \frac{m}{n^2}) \):
      1. for \( x = 0, \ldots, jm \):
         1. for \( y = 0, \ldots, jm \):
            \[
            S(j, k, x, y) = S(j-1, k-1, x-R(p_j), y) \vee S(j-1, k, x, y-R(p_j))
            \]
   3. Search for True entry at \( S(j) \) \( \Theta(n^4m^2) \)

**Run Time**

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

1. Initialize \( S(0,0,0,0) = True \)
2. for \( j = 1, \ldots, n \):
   1. for \( k = 1, \ldots, \min(j, \frac{m}{n^2}) \):
      1. for \( x = 0, \ldots, jm \):
         1. for \( y = 0, \ldots, jm \):
            \[
            S(j, k, x, y) = S(j-1, k-1, x-R(p_j), y) \vee S(j-1, k, x, y-R(p_j))
            \]
   3. Search for True entry at \( S(j) \) \( \Theta(n^4m^2) \)

**\( \Theta(n^4m^2) \)**

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