CS 341: ALGORITHMS

Trevor Brown

trevor.brown@uwaterloo.ca

DC 2338, Office hour M3-4pm
THIS TIME

• More greedy algorithms
  • Coin changing
  • Stable marriage
PROBLEM: COIN CHANGING

Problem 4.5

Coin Changing

Instance: A list of coin denominations, \(d_1, d_2, \ldots, d_n\), and a positive integer \(T\), which is called the target sum.

Find: An \(n\)-tuple of non-negative integers, say \(A = [a_1, \ldots, a_n]\), such that \(T = \sum_{i=1}^{n} a_i d_i\) and such that \(N = \sum_{i=1}^{n} a_i\) is minimized.

In the **Coin Changing** problem, \(a_i\) denotes the number of coins of denomination \(d_i\) that are used, for \(i = 1, \ldots, n\).

The total value of all the chosen coins must be exactly equal to \(T\). We want to **minimize** the number of coins used, which is denoted by \(N\).
EXAMPLE: (OLD) CANADIAN COINS

- Penny (1 cent)
- Nickel (5 cents)
- Dime (10 cents)
- Quarter (25 cents)
- Toonie (2 dollars / 200 cents)
- 50 Cent Piece (50 cents)
- Loonie (1 dollar / 100 cents)
EXAMPLE: (OLD) CANADIAN COINS

• Input: coin denominations = 200, 100, 25, 10, 5, 1 (R.I.P.)
  target sum $T = 155$

• Output: **minimum number** of coins to pay $T$
  (and list of coins)

• Solution: $1 \times 100 + 2 \times 25 + 1 \times 5$ ; 4 coins

• Suggestion for an algorithm?
  • Sort coin denominations from largest to smallest value
  • Greedily use the largest possible coin at all times
Algorithm: GreedyCoinChanging \((D : \text{array}; T : \text{integer})\)

comment: \(D = [d_1, \ldots, d_n]\)

sort the coins so that \(d_1 > \cdots > d_n\)

\(N \leftarrow 0\)

\(\text{for } i \leftarrow 1 \text{ to } n\)

\[
\begin{align*}
    a_i & \leftarrow \left\lfloor \frac{T}{d_i} \right\rfloor \\
    T & \leftarrow T - a_i d_i \\
    N & \leftarrow N + a_i
\end{align*}
\]

\(\text{if } T > 0\)

\(\text{then return } (\text{fail})\)

\(\text{else return } ([a_1, \ldots, a_n], N)\)
OPTIMALITY

• Is this algorithm optimal?
• Trying to build a correctness argument:
  • Fix part of the input:
    • Canadian coin system (including pennies)
  • Try to prove optimality for all target sums T
• Reasoning about **one class of inputs** at a time can make an algorithm easier to understand
We will prove that the greedy algorithm always finds an optimal solution for coin denominations $D = [100, 25, 10, 5, 1]$.

We will make use of the following properties of any optimal solution:

(1) the number of pennies is at most 4 (replace five pennies by a nickel)

(2) the number of nickels is at most 1 (replace two nickels by a dime)

(3) the number of quarters is at most 3 (replace four quarters by a loonie), and

(4) the number of nickels + the number of dimes is at most 2 (replace three dimes by a quarter and a nickel; replace two dimes and a nickel by a quarter; the number of nickels is at most one).

The proof is by induction on $T$. As (trivial) base cases, we can take $T = 1, 2, 3, 4$. 
Inductive step ($T>4$): assume greedy makes optimal change for target values less than $T$. Show it makes optimal change for $T$.

Suppose $5 \leq T < 10$. First, assume there is no nickel in the optimal solution. Then the optimal solution contains only of pennies, so $T \leq 4$ (property (1)); contradiction. Therefore the optimal solution contains at least one nickel. Clearly the greedy solution contains at least one nickel. By induction, the greedy solution for $T - 5$ is optimal. Therefore the greedy solution for $T$ is also optimal.

Suppose $10 \leq T < 25$. First, assume there is no dime in the optimal solution. Then the optimal solution contains only nickels and pennies, so $T \leq 5 + 4 = 9$ (property (2)); contradiction. Therefore the optimal solution contains at least one dime. Clearly the greedy solution contains at least one dime. By induction, the greedy solution for $T - 10$ is optimal. Therefore the greedy solution for $T$ is also optimal.
**EXERCISE: 25 ≤ T < 100**

Suppose $10 \leq T < 25$. First, assume there is no dime in the optimal solution. Then the optimal solution contains only nickels and pennies, so $T \leq 5 + 4 = 9$ (property (2)); contradiction. Therefore the optimal solution contains at least one dime. Clearly the greedy solution contains at least one dime. By induction, the greedy solution for $T - 10$ is optimal. Therefore the greedy solution for $T$ is also optimal.

1. the number of pennies is at most 4 (replace five pennies by a nickel)
2. the number of nickels is at most 1 (replace two nickels by a dime)
3. the number of quarters is at most 3 (replace four quarters by a loonie), and
4. the number of nickels + the number of dimes is at most 2 (replace three dimes by a quarter and a nickel; replace two dimes and a nickel by a quarter; the number of nickels is at most one).
• Exercise: suppose $25 \leq T < 100$
  • Find one coin that must be in optimal & greedy to reduce this case to making change for less than $T$
• Assume no quarters in optimal solution
  • Then by properties 1&4, the optimal solution uses at most: (4 pennies) and (2 nickels or dimes)
  • Max value is therefore 24 cents, so cannot make $T$ change!
• So optimal contains a quarter. (And so does greedy.)
• By inductive hypothesis, greedy is optimal for $T - 25$.
• So, greedy is optimal for $T$. 
• Exercise: suppose $100 \leq T < 200$
  • Find one coin that must be in optimal & greedy to reduce this case to making change for \textbf{less than T}
• Assume no loonies in optimal solution
  • Then by properties 1, 3, 4, the optimal solution uses at most:
    (4 pennies) and (2 nickels or dimes) and (3 quarters)
  • Max value is therefore 99 cents, so \textbf{cannot} make T change!
• So optimal contains a loonie. (And so does greedy.)
• By inductive hypothesis, greedy is optimal for $T - 100$.
• So, greedy is optimal for T.
• Exercise for home: $200 \leq T$
OPTIMALITY CONTINUED...

- Optimal for old Canadian coin system
- How about new Canadian coin system?
  - Denominations: 200, 100, 25, 10, 5
  - Some values can’t be created at all!
- How about the old British coin system
  - Denominations: 30, 24, 12, 6, 3, 1
  - Counter-example: T=48. Greedy=30,12,6 ; Opt=24,24
- What makes a coin system optimal / non-optimal?
MORE CHALLENGING HOME EXERCISE:

• Show greedy is optimal for any coin system satisfying:
  • $d_j \mid d_{j-1}$ for all $j, 2 \leq j \leq n$
  • Hints (tiny font, so no spoilers):
    • Is greedy non-optimal for any coin system that does not satisfy this property?
    • No, it’s optimal for old Canadian coins even though 10 does not divide 25
Problem 4.6
Stable Matching

Instance: Two sets of size \( n \) say \( X = [x_1, \ldots, x_n] \) and \( Y = [y_1, \ldots, y_n] \). Each \( x_i \) has a preference ranking of the elements in \( Y \), and each \( y_i \) has a preference ranking of the elements in \( X \). \( \text{pref}(x_i, j) = y_k \) if \( y_k \) is the \( j \)-th favourite element of \( Y \) of \( x_i \); and \( \text{pref}(y_i, j) = x_k \) if \( x_k \) is the \( j \)-th favourite element of \( X \) of \( y_i \).

Find: A matching of the sets \( X \) and \( Y \) such that there does not exist a pair \((x_i, y_j)\) which is not in the matching, but where \( x_i \) and \( y_j \) prefer each other to their existing matches. A matching with this this property is called a stable matching.
This problem is also known as the **Stable Marriage Problem**.

Real-world examples (1950s):

- Matching medical interns to hospitals.
- Matching organs to patients requiring transplants

The 2012 Nobel Prize in economics was awarded to Roth and Shapley for their work in the “theory of stable allocation and the practice of market design”.

An example of an instability: Suppose $x_i$ is matched with $y_j$, $x_k$ is matched with $y_\ell$, $x_i$ prefers $y_\ell$ to $y_j$, and $y_\ell$ prefers $x_i$ to $x_k$.
Overview of the Gale-Shapley Algorithm

Elements of $X$ propose to elements of $Y$.
If $y_j$ accepts a proposal from $x_i$, then the pair $\{x_i, y_j\}$ is matched.
An unmatched $y_j$ must accept a proposal from any $x_i$.
If $\{x_i, y_j\}$ is a matched pair, and $y_j$ subsequently receives a proposal from $x_k$, where $y_j$ prefers $x_k$ to $x_i$, then $y_j$ accepts and the pair $\{x_i, y_j\}$ is replaced by $\{x_k, y_j\}$.
If $\{x_i, y_j\}$ is a matched pair, and $y_j$ subsequently receives a proposal from $x_k$, where $y_j$ prefers $x_i$ to $x_k$, then $y_j$ rejects and nothing changes.
A matched $y_j$ never becomes unmatched.
An $x_i$ might make a number of proposals (up to $n$); the order of the proposals is determined by $x_i$’s preference list.
Algorithm: Gale-Shapley ($X, Y, \text{pref}$)

$\text{Match} \leftarrow \emptyset$

while there exists an unmatched $x_i$

let $y_j$ be the next element in $x_i$’s preference list

if $y_j$ is not matched

then $\text{Match} \leftarrow \text{Match} \cup \{x_i, y_j\}$

suppose $\{x_k, y_j\} \in \text{Match}$

if $y_j$ prefers $x_i$ to $x_k$

then $\{\text{Match} \leftarrow \text{Match}\backslash\{x_k, y_j\} \cup \{x_i, y_j\}\}$

comment: $x_k$ is now unmatched

else

Propose to most desired $y$

Unmatched $y_j$ accepts any proposal

Termination means...?

return $(\text{Match})$
Suppose we have the following preference lists:

\[
\begin{align*}
    x_1 &: y_2 > y_3 > y_1 \\
    x_2 &: y_1 > y_3 > y_2 \\
    x_3 &: y_1 > y_2 > y_3 \\
    y_1 &: x_1 > x_2 > x_3 \\
    y_2 &: x_2 > x_3 > x_1 \\
    y_3 &: x_3 > x_2 > x_1
\end{align*}
\]

The *Gale-Shapley algorithm* could be executed as follows:

<table>
<thead>
<tr>
<th>proposal</th>
<th>result</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$ proposes to $y_2$</td>
<td>$y_2$ accepts</td>
<td>${x_1, y_2}$</td>
</tr>
<tr>
<td>$x_2$ proposes to $y_1$</td>
<td>$y_1$ accepts</td>
<td>${x_1, y_2}, {x_2, y_1}$</td>
</tr>
<tr>
<td>$x_3$ proposes to $y_1$</td>
<td>$y_1$ rejects</td>
<td></td>
</tr>
<tr>
<td>$x_3$ proposes to $y_2$</td>
<td>$y_2$ accepts</td>
<td>${x_3, y_2}, {x_2, y_1}$</td>
</tr>
<tr>
<td>$x_1$ proposes to $y_3$</td>
<td>$y_3$ accepts</td>
<td>${x_3, y_2}, {x_2, y_1}, {x_1, y_3}$</td>
</tr>
</tbody>
</table>
AN IN-CLASS EXERCISE

Suppose we have the following preference lists:

\[ x_1 : y_1 > y_2 > y_3 > y_4 \]
\[ x_2 : y_2 > y_3 > y_1 > y_4 \]
\[ x_3 : y_3 > y_1 > y_2 > y_4 \]
\[ x_4 : y_1 > y_2 > y_3 > y_4 \]
\[ y_1 : x_2 > x_3 > x_4 > x_1 \]
\[ y_2 : x_3 > x_4 > x_1 > x_2 \]
\[ y_3 : x_4 > x_1 > x_2 > x_3 \]
\[ y_4 : x_1 > x_2 > x_3 > x_4 \]

Exercise: Show the execution of the Gale-Shapley algorithm.

<table>
<thead>
<tr>
<th>proposal</th>
<th>result</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 ) proposes to ( y_1 )</td>
<td>( y_1 ) accepts</td>
<td>( {x_1, y_1} )</td>
</tr>
</tbody>
</table>

You figure this out ... answer on next slide ... in a few minutes
<table>
<thead>
<tr>
<th>proposal</th>
<th>result</th>
<th>\textbf{Match}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$ proposes to $y_1$</td>
<td>$y_1$ accepts</td>
<td>${x_1, y_1}$</td>
</tr>
<tr>
<td>$x_2$ proposes to $y_2$</td>
<td>$y_2$ accepts</td>
<td>${x_1, y_1}, {x_2, y_2}$</td>
</tr>
<tr>
<td>$x_3$ proposes to $y_3$</td>
<td>$y_3$ accepts</td>
<td>${x_1, y_1}, {x_2, y_2}, {x_3, y_3}$</td>
</tr>
<tr>
<td>$x_4$ proposes to $y_1$</td>
<td>$y_1$ accepts</td>
<td>${x_4, y_1}, {x_2, y_2}, {x_3, y_3}$</td>
</tr>
<tr>
<td>$x_1$ proposes to $y_2$</td>
<td>$y_2$ accepts</td>
<td>${x_4, y_1}, {x_2, y_2}, {x_3, y_3}$</td>
</tr>
<tr>
<td>$x_2$ proposes to $y_3$</td>
<td>$y_3$ accepts</td>
<td>${x_4, y_1}, {x_2, y_2}, {x_3, y_3}$</td>
</tr>
<tr>
<td>$x_3$ proposes to $y_1$</td>
<td>$y_1$ accepts</td>
<td>${x_3, y_1}, {x_2, y_2}, {x_3, y_3}$</td>
</tr>
<tr>
<td>$x_4$ proposes to $y_2$</td>
<td>$y_2$ accepts</td>
<td>${x_3, y_1}, {x_4, y_2}, {x_3, y_3}$</td>
</tr>
<tr>
<td>$x_1$ proposes to $y_3$</td>
<td>$y_3$ accepts</td>
<td>${x_3, y_1}, {x_4, y_2}, {x_1, y_3}$</td>
</tr>
<tr>
<td>$x_2$ proposes to $y_1$</td>
<td>$y_1$ accepts</td>
<td>${x_2, y_1}, {x_4, y_2}, {x_1, y_3}$</td>
</tr>
<tr>
<td>$x_3$ proposes to $y_2$</td>
<td>$y_2$ accepts</td>
<td>${x_2, y_1}, {x_3, y_2}, {x_1, y_3}$</td>
</tr>
<tr>
<td>$x_4$ proposes to $y_3$</td>
<td>$y_3$ accepts</td>
<td>${x_2, y_1}, {x_3, y_2}, {x_4, y_3}$</td>
</tr>
<tr>
<td>$x_1$ proposes to $y_4$</td>
<td>$y_4$ accepts</td>
<td>${x_2, y_1}, {x_3, y_2}, {x_4, y_3}, {x_1, y_4}$</td>
</tr>
</tbody>
</table>
Proof of Correctness

First we need to show that the algorithm always terminates, i.e., it is impossible that an unmatched $x_i$ has proposed to every $y_j$.

Termination of the algorithm: Once an element of $Y$ is matched, they are never unmatched. If $x_i$ has proposed to every $y_j$, then every $y_j$ is matched. But then every element of $X$ is matched, which is a contradiction.
Proof of Correctness

First we need to show that the algorithm always terminates, i.e., it is impossible that an unmatched $x_i$ has proposed to every $y_j$.

Termination of the algorithm: Once an element of $Y$ is matched, they are never unmatched. If $x_i$ has proposed to every $y_j$, then every $y_j$ is matched. But then every element of $X$ is matched, which is a contradiction.

So the algorithm terminates, and each $x_i$ is matched with some $y_j$

Need to argue the matching is stable

That is, no $x_i$ and $y_j$ prefer each other more than their current partners
To prove that the algorithm terminates with a stable matching: Suppose there is an instability: \( x_i \) is matched with \( y_j \), \( x_k \) is matched with \( y_l \), \( x_i \) prefers \( y_l \) to \( y_j \) and \( y_l \) prefers \( x_i \) to \( x_k \). Observe that \( x_i \) proposed to \( y_l \) before he proposed to \( y_j \).

There three cases to consider:

1. \( y_l \) rejected \( x_i \)'s proposal.
2. \( y_l \) accepted \( x_i \)'s proposal, but later accepted another proposal.
3. \( y_l \) accepted \( x_i \)'s proposal, and did not accept any subsequent proposal.

All three cases are impossible, and hence we conclude that the resulting matching is stable.
EXPLANATION OF CASES

(1) $y_\ell$ rejected $x_i$’s proposal. This could happen only if $y_\ell$ was already matched with someone she preferred to $x_i$. But $y_\ell$ ended up matched with someone she liked less than $x_i$. We conclude that $y_\ell$ did not reject a proposal by $x_i$.

(2) $y_\ell$ accepted $x_i$’s proposal, but later accepted another proposal. This could happen only if $y_\ell$ later received a proposal from someone when preferred to $x_i$. But $y_\ell$ ended up matched with someone she liked less than $x_i$, so this also did not happen.

(3) $y_\ell$ accepted $x_i$’s proposal, and did not accept any subsequent proposal. In this case, $y_\ell$ would have ended up matched to $x_i$, which did not happen.
COMPLEXITY

It is obvious that the number of iterations is at most $n^2$ since every $x_i$ proposes at most once to every $y_j$.

It is possible to prove the stronger result that the maximum number of iterations is $n^2 - n + 1$.

The average number of iterations is $\Theta(n \log n)$ (but we will not prove this).

Is there an efficient way to identify an unmatched $x_i$ at any point in the algorithm?

What data structure would be helpful in doing this?

What can we then say about the complexity of the algorithm?
IMPROVED IMPLEMENTATION

Keep track of matchings using a dynamic array indexed by the elements of X and Y: \( M[x_i] = y_j \) and \( M[y_j] = x_i \) if \( x_i \) and \( y_j \) are currently matched.

\( M[x_i] = 0 \) for if \( x_i \) is unmatched and \( M[y_j] = 0 \) if \( y_j \) is unmatched.

Keep track of the unmatched \( x_i \)'s using a dynamic queue. Initially the queue contains all elements of \( X \). Delete \( x_i \) from the queue when \( x_i \) is matched with a \( y_j \). \( x_i \) enters the queue if \( y_j \) later accepts a proposal from some \( x_k \neq x_i \).

For each \( x_i \), create a static linked list containing \( x_i \)'s preference list.

For each \( y_j \), create a static array \( R \) such that \( R[y_j, x_i] = k \) if \( x_i \) is \( y_j \)'s \( k \)th favourite element of \( X \).

Then each iteration requires \( O(1) \) time and the resulting algorithm has complexity \( O(n^2) \).
DYNAMIC PROGRAMMING
A powerful algorithmic design paradigm
Richard Bellman, the inventor of dynamic programming in 1950, related the following in his autobiography:

“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, lets kill two birds with one stone. Lets take a word that 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 its impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. Its 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.”
Computing Fibonacci Numbers Inefficiently

The Recursion Tree to Evaluate $f_5$:

**Algorithm: BadFib\(n\)**

if $n = 0$ then $f \leftarrow 0$
else if $n = 1$ then $f \leftarrow 1$
else
    \[
    \begin{align*}
    f_1 & \leftarrow \text{BadFib}(n - 1) \\
    f_2 & \leftarrow \text{BadFib}(n - 2) \\
    f & \leftarrow f_1 + f_2
    \end{align*}
    \]
return $(f)$;
Complexity of the Algorithm

The recurrence tree has $f_n$ leaf nodes with the value 1 and $f_{n-1}$ leaf nodes with the value 0. So there are a total of $f_{n+1}$ leaf nodes. The number of interior nodes is $f_{n+1} - 1$.

In the unit cost model, the complexity of computing $f_n$ is $\Theta(f_{n+1})$.

How quickly does $f_n$ grow? Let $\phi = (1 + \sqrt{5})/2$; then

$$f_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}} = \left[ \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right].$$

Therefore $f_n \in \Theta(\phi^n)$ and hence we also have $f_{n+1} \in \Theta(\phi^n)$.

The value $\phi \approx 1.6$ is the golden ratio.

The time to compute $f_n$ is exponential in $n$. 

The unit cost model understates the computation time because the numbers in the sequence are growing exponentially quickly.

This is an inefficient use of recursion because we have to solve subproblems $f_{n-1}$ and $f_{n-2}$ to solve the given problem instance $f_n$.

The recurrence tree ends up being of linear depth and exponential size (as a function of $n$).

In divide-and-conquer, we typically solve subproblems of size $n/2$ to solve the given instance of size $n$.

In these situations, the recurrence tree is of logarithmic depth and polynomial size.
Computing Fibonacci Numbers More Efficiently

**Algorithm:** BetterFib(n)

\[
\begin{align*}
f[0] &\leftarrow 0 \\
f[1] &\leftarrow 1 \\
\text{for } i &\leftarrow 2 \text{ to } n \\
\quad &\text{do } f[i] \leftarrow f[i - 1] + f[i - 2] \\
\text{return } (f[n])
\end{align*}
\]

Compute bottom-up (iteratively) instead of top-down (recursively)!

This **bottom-up pre-building of solutions to recursive subproblems** is called dynamic programming.

Note that \( f_n \) has \( \Theta(n) \) digits.

Complexity? (To start, assume \( O(1) \) cost for addition.)

Complexity for linear time addition?
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 \( S(I) \).
Designing Dynamic Programming Algorithms (cont.)

**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 $I$. 
Base cases are important: they totally determine the final solution.

Algorithm: BetterFib(n)
\[ f[0] \leftarrow 0 \]
\[ f[1] \leftarrow 1 \]
\[ \text{for } i \leftarrow 2 \text{ to } n \]
\[ \quad \text{do } f[i] \leftarrow f[i - 1] + f[i - 2] \]
\[ \text{return } (f[n]) \]

Express solution to problem size \(i\) in terms of problem sizes \(i-1\) and \(i-2\).

Combining solutions to subproblems is easy in this case: just add (+)

Recurrence relation that leads to this code:

\[
f(n) = \begin{cases} 
  f(n - 1) + f(n - 2) : i \geq 2 \\
  1 : i = 1 \\
  0 : i = 0 
\end{cases}
\]
NEXT TIME

• Dynamic programming algorithms
  • 0-1 Knapsack
  • Coin Changing (when greedy fails)