CS 341: ALGORITHMS

Lecture 11: dynamic programming (after finishing greedy)

Slides by Trevor Brown (some material from Doug Stinson)

trevor.brown@uwaterloo.ca
(http://tbrown.pro)
DATING WEBSITE FOR CHEATERS HACKED
Hackers threaten to expose customers if website isn’t shut down

PROBLEM: STABLE MARRIAGE
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.

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: \textit{Gale-Shapley}(X, Y, \textit{pref})

\begin{align*}
\text{Match} & \leftarrow \emptyset \\
\text{while} \quad & \text{there exists an unmatched } x_i \\
\quad & \text{let } y_j \text{ be the next element in } x_i \text{'s preference list} \\
& \quad \text{if } y_j \text{ is not matched} \\
& \quad \quad \text{then } \text{Match} \leftarrow \text{Match} \cup \{x_i, y_j\} \\
& \quad \quad \text{do} \\
& \quad \quad \quad \quad \text{suppose } \{x_k, y_j\} \in \text{Match} \\
& \quad \quad \quad \quad \text{if } y_j \text{ prefers } x_i \text{ to } x_k \\
& \quad \quad \quad \quad \quad \text{then } \text{Match} \leftarrow \text{Match}\backslash\{x_k, y_j\} \cup \{x_i, y_j\} \\
& \quad \quad \text{else} \\
& \quad \quad \quad \quad \text{if } y_j \text{ prefers } x_i \text{ to } x_k \\
& \quad \quad \quad \quad \quad \text{then } \text{Match} \leftarrow \text{Match}\backslash\{x_k, y_j\} \cup \{x_i, y_j\} \\
& \quad \quad \quad \quad \quad \text{comment: } x_k \text{ is now unmatched} \\
\text{end do} \\
\text{return } (\text{Match})
\end{align*}

Keeps track of \textbf{current} matches

Termination is not so obvious…

Propose to \textbf{most desired} \textit{y}

Unmatched \textit{y}_j \text{ accepts} \textbf{any} \textit{proposal}

Matched \textit{y}_j \text{ considers} \textbf{upgrading}
Suppose we have the following preference lists:

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

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>
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 (i.e., optimal).

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 $y_{\ell}$, $x_i$ prefers $y_{\ell}$ to $y_j$ and $y_{\ell}$ prefers $x_i$ to $x_k$.

Observe: $x_i$ proposes to $y_{\ell}$ before proposing to $y_j$

There three cases to consider:

1. $y_{\ell}$ rejected $x_i$’s proposal.
2. $y_{\ell}$ accepted $x_i$’s proposal, but later accepted another proposal.
3. $y_{\ell}$ accepted $x_i$’s proposal, and did not accept any subsequent proposal.

Then $y_{\ell}$ should end up matched with $x_i$. Contradiction!

Other proposal must be to someone better. Contradiction!

Contradicts our assumption that this instability exists!

All three cases are impossible, so assumption is wrong. There cannot be an instability!
It is obvious that the number of iterations is at most $n^2$ since every $x_i$ proposes at most once to every $y_j$.

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

But how much time does it take per iteration?
Algorithm: Gale-Shapley\((X, Y, \text{pref})\)

\[ \text{Match} \leftarrow \emptyset \]

\[ \text{while there exists an unmatched} \ x_i \]

\[ \left\{ \begin{array}{l} \text{let} \ y_j \ \text{be the next element in} \ x_i \text{'s preference list} \\
\text{if} \ y_j \ \text{is not matched} \\
\text{then} \ \text{Match} \leftarrow \text{Match} \cup \{x_i, y_j\} \\
\text{do} \left\{ \begin{array}{l} \text{suppose} \ \{x_k, y_j\} \in \text{Match} \\
\text{if} \ y_j \text{prefers} \ x_i \ \text{to} \ x_k \\
\text{then} \ \text{Match} \leftarrow \text{Match} \setminus \{x_k, y_j\} \cup \{x_i, y_j\} \\
\text{comment:} \ x_k \ \text{is now unmatched} \\
\end{array} \right. \\
\text{else} \left\{ \begin{array}{l} \text{suppose} \ \{x_k, y_j\} \in \text{Match} \\
\text{if} \ y_j \text{prefers} \ x_i \ \text{to} \ x_k \\
\text{then} \ \text{Match} \leftarrow \text{Match} \setminus \{x_k, y_j\} \cup \{x_i, y_j\} \\
\text{comment:} \ x_k \ \text{is now unmatched} \\
\end{array} \right. \\
\end{array} \right. \\
\end{array} \]

\[ \text{return} \ (\text{Match}) \]

- Maintain a **queue** of unmatched \(x\) elements
- Simple **queue** of preferences
- Want to know who \(y_j\) is matched with
- Maintain **arrays** of matches. If \(x_i\) and \(y_j\) are matched then \(M_x[i] = j\) and \(M_y[j] = i\) (Initially \(M_x[i], M_y[i] = 0\))
- Construct an **array** \(R[j, i]\) containing the rank of \(x_i\) in \(y_j\)'s preference list
- I.e., want \(R[j, i] = k\) if \(x_i\) is \(y_j\)'s \(k\)-th favourite partner

So, we get \(O(1)\) time per iteration, and \(O(n^2)\) time in total

Exercise: try writing pseudocode for this implementation.
DYNAMIC PROGRAMMING

Sort of like divide and conquer... but sometimes better
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.”

So, there isn’t a great reason for the name...

“Bottom-up recursion" might be a better name, as we’ll see...
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
\[
\begin{cases} 
  f_1 \leftarrow BadFib(n - 1) \\
  f_2 \leftarrow BadFib(n - 2) \\
  f \leftarrow f_1 + f_2
\end{cases}
\]

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.

The problem with recursion here:

Instead of dividing the problem into disjoint subproblems, we have tons of overlap in our subproblems!

This overlap suggests dynamic programming may be able to help!

In our analyses before this point, $n$ has always coincidentally been the size of the input.

So this is doubly exponential in the size of the input!

$$T(n) = T(n - 1) + T(n - 2) + O(1) \geq 2T(n - 2) + O(1) \ldots n/2 \text{ levels, and work doubles at each level!}$$

This is exponential in $n$. But, in this case, $n$ is not the size of the input.

The size of the input is $\log n$, since that is the number of bits needed to write the input down.

So this is doubly exponential in the size of the input!

$$2^n = 2^{\log n} = 2^{\text{size}}$$
**Computing Fibonacci Numbers More Efficiently**

**Algorithm: BetterFib(n)**

```plaintext
f[0] ← 0
f[1] ← 1
for i ← 2 to n
do f[i] ← f[i - 1] + f[i - 2]
return (f[n])
```

**Complexity?** (For simplicity, assume $O(1)$ cost for addition.)

Exercise: since $f_n$ can have $O(n)$ digits, we should really assume adding takes $O(n)$ time. How does this change the complexity? Is it still better than doubly exponential in $n$?

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

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

Takes $O(n)$ time. That’s linear in $n$, but not linear in the input size.

Input size $s$ is $\log n$, and $n = 2^{\log n} = 2^s$. So, runtime is exponential in the input size. But it’s not doubly exponential...

In this case, DP exponentially improves runtime vs the recursive algorithm!
HOME EXERCISES
Suppose we have the following preference lists:

\[
\begin{align*}
x_1 &: y_1 > y_2 > y_3 > y_4 \\
x_2 &: y_2 > y_3 > y_1 > y_4 \\
x_3 &: y_3 > y_1 > y_2 > y_4 \\
x_4 &: y_1 > y_2 > y_3 > y_4 \\
y_1 &: x_2 > x_3 > x_4 > x_1 \\
y_2 &: x_3 > x_4 > x_1 > x_2 \\
y_3 &: x_4 > x_1 > x_2 > x_3 \\
y_4 &: x_1 > x_2 > x_3 > x_4
\end{align*}
\]

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 ...
<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>
<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>${x_4, y_1}, {x_1, y_2}, {x_3, y_3}$</td>
</tr>
<tr>
<td>$x_2$ proposes to $y_3$</td>
<td>$y_3$ accepts</td>
<td>${x_4, y_1}, {x_1, y_2}, {x_2, y_3}$</td>
</tr>
<tr>
<td>$x_3$ proposes to $y_1$</td>
<td>$y_1$ accepts</td>
<td>${x_3, y_1}, {x_1, y_2}, {x_2, y_3}$</td>
</tr>
<tr>
<td>$x_4$ proposes to $y_2$</td>
<td>$y_2$ accepts</td>
<td>${x_3, y_1}, {x_4, y_2}, {x_2, y_3}$</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>