CS 341: Algorithms

Douglas R. Stinson

David R. Cheriton School of Computer Science
University of Waterloo

January 30, 2019
1 Course Information
2 Introduction
3 Divide-and-Conquer Algorithms
4 Greedy Algorithms
5 Dynamic Programming Algorithms
6 Graph Algorithms
7 Intractability and Undecidability
Table of Contents

4 Greedy Algorithms
  - Optimization Problems
  - Design Strategy
  - Interval Selection
  - Interval Colouring
  - Knapsack
  - Coin Changing
  - Stable Matching Problem
Optimization Problems

**Problem:** Given a problem instance, find a feasible solution that maximizes (or minimizes) a certain objective function.

**Problem Instance:** Input for the specified problem.

**Problem Constraints:** Requirements that must be satisfied by any feasible solution.

**Feasible Solution:** For any problem instance $I$, $\text{feasible}(I)$ is the set of all outputs (i.e., solutions) for the instance $I$ that satisfy the given constraints.

**Objective Function:** A function $f : \text{feasible}(I) \rightarrow \mathbb{R}^+ \cup \{0\}$. We often think of $f$ as being a profit or a cost function.

**Optimal Solution:** A feasible solution $X \in \text{feasible}(I)$ such that the profit $f(X)$ is maximized (or the cost $f(X)$ is minimized).
The Greedy Method

partial solutions

Given a problem instance $I$, it should be possible to write a feasible solution $X$ as a tuple $[x_1, x_2, \ldots, x_n]$ for some integer $n$, where $x_i \in \mathcal{X}$ for all $i$. A tuple $[x_1, \ldots, x_i]$ where $i < n$ is a partial solution if no constraints are violated. Note: it may be the case that a partial solution cannot be extended to a feasible solution.

choice set

For a partial solution $X = [x_1, \ldots, x_i]$ where $i < n$, we define the choice set

$$\text{choice}(X) = \{y \in \mathcal{X} : [x_1, \ldots, x_i, y] \text{ is a partial solution}\}.$$
The Greedy Method (cont.)

**local evaluation criterion**

For any \( y \in X \), \( g(y) \) is a **local evaluation criterion** that measures the cost or profit of including \( y \) in a (partial) solution.

**extension**

Given a partial solution \( X = [x_1, \ldots, x_i] \) where \( i < n \), choose \( y \in \text{choice}(X) \) so that \( g(y) \) is as small (or large) as possible. Update \( X \) to be the \((i + 1)\)-tuple \([x_1, \ldots, x_i, y]\).

**greedy algorithm**

Starting with the “empty” partial solution, repeatedly extend it until a feasible solution \( X \) is constructed. This feasible solution may or may not be optimal.
Features of the Greedy Method

Greedy algorithms do no **looking ahead** and no **backtracking**.

Greedy algorithms can usually be implemented efficiently. Often they consist of a **preprocessing step** based on the function \( g \), followed by a **single pass** through the data.

In a greedy algorithm, only **one feasible solution** is constructed.

The execution of a greedy algorithm is based on **local criteria** (i.e., the values of the function \( g \)).

**Correctness:** For certain greedy algorithms, it is possible to prove that they always yield optimal solutions. However, these proofs can be tricky and complicated!
Interval Selection

Problem 4.1

Interval Selection

Instance: A set \( A = \{A_1, \ldots, A_n\} \) of intervals.

For \( 1 \leq i \leq n \), \( A_i = [s_i, f_i) \), where \( s_i \) is the start time of interval \( A_i \) and \( f_i \) is the finish time of \( A_i \).

Feasible solution: A subset \( B \subseteq A \) of pairwise disjoint intervals.

Find: A feasible solution of maximum size (i.e., one that maximizes \( |B| \)).
Possible Greedy Strategies for Interval Selection

1. Sort the intervals in increasing order of starting times. At any stage, choose the **earliest starting** interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $s_i$).

2. Sort the intervals in increasing order of duration. At any stage, choose the interval of **minimum duration** that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $f_i - s_i$).

3. Sort the intervals in increasing order of finishing times. At any stage, choose the **earliest finishing** interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $f_i$).

Does one of these strategies yield a correct greedy algorithm?
A Greedy Algorithm for Interval Selection

Algorithm: $\text{GreedyIntervalSelection}(A)$
rename the intervals, by sorting if necessary, so that $f_1 \leq \cdots \leq f_n$
$B \leftarrow \{A_1\}$
$prev \leftarrow 1$

$\text{comment: } prev$ is the index of the last selected interval
for $i \leftarrow 2$ to $n$
    do $\begin{cases} 
        \text{if } s_i \geq f_{prev} \\
        \text{then } \begin{cases} 
            B \leftarrow B \cup \{A_i\} \\
            prev \leftarrow i 
        \end{cases} 
    \end{cases}$
return $(B)$
Correctness Proof

We give an induction proof.

Let $B$ be the greedy solution,

$$B = (A_{i_1}, \ldots, A_{i_k}),$$

where $i_1 < \cdots < i_k$.

Let $O$ be any optimal solution,

$$O = (A_{j_1}, \ldots, A_{j_\ell}),$$

where $j_1 < \cdots < j_\ell$.

Observe that $\ell \geq k$ since $O$ is optimal.

We want to prove that $\ell = k$. 
Correctness Proof (cont.)

**Lemma 4.2 (Greedy stays ahead)**

\[ f_{i_m} \leq f_{j_m} \text{ for } m = 1, 2, \ldots \]

**Proof.**

Initial case \( m = 1 \). We have \( f_{i_1} \leq f_{j_1} \) since the greedy algorithm begins by choosing \( i_1 = 1 \). (\( A_1 \) has the earliest finishing time.)

Induction assumption: \( f_{i_{m-1}} \leq f_{j_{m-1}} \). Consider \( A_{i_m} \) and \( A_{j_m} \). We have

\[ s_{j_m} \geq f_{j_{m-1}} \geq f_{i_{m-1}}. \]

\( A_{i_m} \) has the earliest finishing time of any interval that starts after \( f_{j_{m-1}} \) finishes. Therefore \( f_{i_m} \leq f_{j_m} \). \( \square \)
Correctness Proof (cont.)

Now we complete the proof.

From the Lemma, we have $f_{i_k} \leq f_{j_k}$.

Suppose that $\ell > k$.

$A_{j_k+1}$ starts after $A_{j_k}$ finishes, and $f_{i_k} \leq f_{j_k}$.

So $A_{j_k+1}$ is feasible WRT the greedy solution, and therefore the greedy solution would not have terminated with $A_{i_k}$.

This contradiction shows that $\ell = k$. 
A Slick Proof

Induction is a standard way to prove correctness of greedy algorithms; however, sometimes shorter “slick” proofs are possible.

Let $F = \{f_{i_1}, \ldots, f_{i_k}\}$ be the finishing times of the intervals in $B$.

There is no interval in $O$ that is “between” $f_{i_m-1}$ and $f_{i_m}$ for any $m \geq 2$, since $A_{i_m}$ would not be chosen by the greedy algorithm in this case.

As well, there is no interval in $O$ that finishes before $f_{i_1}$, or one that starts after $f_{i_k}$.

Therefore every interval in $O$ contains a point in $F$ (or has a point in $F$ as a finishing time).

No two intervals in $O$ contain the same point in $F$ because the intervals are disjoint.

Hence, there is an injective mapping from $O$ to $F$ and therefore $|O| \leq |F|$. Then we have

$$\ell = |O| \leq |F| = |B| = k.$$
Interval Colouring

Problem 4.3

Interval Colouring

Instance: A set $A = \{A_1, \ldots, A_n\}$ of intervals.

For $1 \leq i \leq n$, $A_i = [s_i, f_i)$, where $s_i$ is the start time of interval $A_i$ and $f_i$ is the finish time of $A_i$.

Feasible solution: A $c$-colouring is a mapping $col : A \rightarrow \{1, \ldots, c\}$ that assigns each interval a colour such that two intervals receiving the same colour are always disjoint.

Find: A $c$-colouring of $A$ with the minimum number of colours.
Greedy Strategies for Interval Colouring

As usual, we consider the intervals one at a time.
At a given point in time, suppose we have coloured the first \( i < n \) intervals using \( d \) colours.

We will colour the \((i+1)\)st interval with any permissible colour. If it cannot be coloured using any of the existing \( d \) colours, then we introduce a new colour and \( d \) is increased by 1.

**Question:** In what order should we consider the intervals?

Consider the following example:

\[
A_1 = [0, 3) \quad A_2 = [8, 11) \quad A_3 = [14, 20) \quad A_4 = [4, 9) \\
A_5 = [16, 20) \quad A_6 = [6, 13) \quad A_7 = [10, 15) \quad A_8 = [0, 7) \\
A_9 = [12, 20) \quad A_{10} = [0, 5)
\]
A Greedy Algorithm for Interval Colouring

Algorithm: \textit{GreedyIntervalColouring}(A)

sort the intervals so that \( s_1 \leq \cdots \leq s_n \)
\( d \leftarrow 1 \)
\( \text{colour}[1] \leftarrow 1 \)
\( \text{finish}[1] \leftarrow f_1 \)
\For {\( i \leftarrow 2 \) to \( n \)}
\begin{align*}
\text{flag} & \leftarrow \text{false} \\
\text{c} & \leftarrow 1 \\
\text{while } c \leq d \text{ and ( not flag) do}
\end{align*}
\begin{align*}
\text{if } \text{finish}[c] \leq s_i \text{ then}
\end{align*}
\begin{align*}
\text{colour}[i] & \leftarrow c \\
\text{finish}[c] & \leftarrow f_i \\
\text{flag} & \leftarrow \text{true}
\end{align*}
\begin{align*}
\text{else } c & \leftarrow c + 1 \\
\text{if not flag then}
\end{align*}
\begin{align*}
\quad d & \leftarrow d + 1 \\
\quad \text{colour}[i] & \leftarrow d \\
\quad \text{finish}[d] & \leftarrow f_i
\end{align*}
\Return (\( d, \text{colour} \))
Correctness of the Algorithm

The correctness of this greedy algorithm can be proven inductively as well as by a “slick” method—we give the “slick” proof:

Let $D$ denote the number of colours used by the algorithm.

Suppose $A_i = [s_i, f_i]$ is the first interval to receive the last colour, $D$.

For every colour $c < D$, there is an interval $A_c = [s_c, f_c]$ such that $s_c \leq s_i < f_c$ (i.e., $A_c$ overlaps $A_i$).

Therefore we have $D$ intervals, all of which contain the point $s_i$.

These $D$ intervals must all receive different colours, so there is no colouring with fewer than $D$ colours.
The complexity of the algorithm is $O(nD)$, where $D$ is the value of $d$ returned by the algorithm.

We don’t know the value of $D$ ahead of time; all we know is that $1 \leq D \leq n$.

If it turns out that $D \in \Omega(n)$, then the best we can say is that the complexity is $O(n^2)$.

What inefficiencies exist in this algorithm?

What data structure would allow a more efficient algorithm to be designed?

What would be the complexity of an algorithm making use of an appropriate data structure?
Implementation Details

For each interval, suppose we searched the $d$ existing colours to find if one of them is suitable. This is a linear search.

A modification is to use the colour of the interval having the earliest finishing time among the most recently chosen intervals of each colour.

We can use a priority queue to keep track of these finishing times.

Whenever we colour interval $A_i$ with colour $c$, we insert $(f_i, c)$ into the priority queue (here $f_i$ is the “key”).

When we want to colour the next interval $A_i$, we look at the minimum key $f$ in the priority queue. If $f \leq s_i$, then we do a deletemin operation, yielding the pair $(f, c)$ and we use colour $c$ for interval $A_i$. If $f > s_i$, we introduce a new colour.

Note that each interval is inserted once and deleted once from the priority queue. Therefore, the complexity of this approach is $O(n \log D)$. Since $D \leq n$, it is $O(n \log n)$. 
Knapsack Problems

Problem 4.4

Knapsack

Instance: Profits $P = [p_1, \ldots, p_n]$; weights $W = [w_1, \ldots, w_n]$; and a capacity, $M$. These are all positive integers.

Feasible solution: An $n$-tuple $X = [x_1, \ldots, x_n]$ where $\sum_{i=1}^{n} w_i x_i \leq M$.

In the 0-1 Knapsack problem (often denoted just as Knapsack), we require that $x_i \in \{0, 1\}$, $1 \leq i \leq n$.

In the Rational Knapsack problem, we require that $x_i \in \mathbb{Q}$ and $0 \leq x_i \leq 1$, $1 \leq i \leq n$.

Find: A feasible solution $X$ that maximizes $\sum_{i=1}^{n} p_i x_i$. 

D.R. Stinson (SCS)
Possible Greedy Strategies for Knapsack Problems

1. Consider the items in decreasing order of profit (i.e., the local evaluation criterion is $p_i$).
2. Consider the items in increasing order of weight (i.e., the local evaluation criterion is $w_i$).
3. Consider the items in decreasing order of profit divided by weight (i.e., the local evaluation criterion is $p_i/w_i$).

Does one of these strategies yield a correct greedy algorithm for the Rational Knapsack problem?
A Greedy Algorithm for Rational Knapsack

Algorithm: \textit{GreedyRationalKnapsack}(P, W : array; M : integer)

sort the items so that $p_1/w_1 \geq \cdots \geq p_n/w_n$

$X \leftarrow [0, \ldots, 0]$

$i \leftarrow 1$

$CurW \leftarrow 0$

\textbf{while} ($CurW < M$) \textbf{and} ($i \leq n$)

\quad \textbf{if} $CurW + w_i \leq M$

\quad \quad \textbf{then} \quad \begin{cases} 
  x_i \leftarrow 1 \\
  CurW \leftarrow CurW + w_i \\
  i \leftarrow i + 1 
\end{cases}

\quad \textbf{else} \quad \begin{cases} 
  x_i \leftarrow (M - CurW)/w_i \\
  CurW := M 
\end{cases}

\textbf{return} $(X)$
Correctness Proof

For simplicity, assume that the profit / weight ratios are all distinct, so

\[
\frac{p_1}{w_1} > \frac{p_2}{w_2} > \ldots > \frac{p_n}{w_n}.
\]

Suppose the greedy solution is \( X = (x_1, \ldots, x_n) \) and the optimal solution is \( Y = (y_1, \ldots, y_n) \).

We will prove that \( X = Y \), i.e., \( x_j = y_j \) for \( j = 1, \ldots, n \). Therefore there is a unique optimal solution and it is equal to the greedy solution.

Suppose \( X \neq Y \).

Pick the smallest integer \( j \) such that \( x_j \neq y_j \).

It is impossible that \( x_j < y_j \), so we have \( x_j > y_j \).

There exists an index \( k > j \) such that \( y_k > 0 \) (otherwise \( Y \) is not optimal).
Correctness Proof (cont.)

Let \( \delta = \min\{w_ky_k, w_j(x_j - y_j)\} \); note that \( \delta > 0 \).

Define
\[
y'_j = y_j + \frac{\delta}{w_j} \quad \text{and} \quad y'_k = y_k - \frac{\delta}{w_k}.
\]

Then let \( Y' \) be \( Y \) with \( y_j \) and \( y_k \) updated to \( y'_j \) and \( y'_k \), respectively.

The idea is to show that
1. \( Y' \) is feasible, and
2. \( \text{profit}(Y') > \text{profit}(Y) \).

This contradicts the optimality of \( Y \) and proves that \( X = Y \).
Correctness Proof (cont.)

To show $Y'$ is feasible, show that $y'_k \geq 0$, $y'_j \leq 1$ and weight($Y'$) $\leq M$.

First, we have

$$y'_k = y_k - \frac{\delta}{w_k} \geq y_k - \frac{w_k y_k}{w_k} = 0.$$  

Second,

$$y'_j = y_j + \frac{\delta}{w_j} \leq y_j + \frac{w_j (x_j - y_j)}{w_j} = x_j \leq 1.$$  

Third,

$$\text{weight}(Y') = \text{weight}(Y) + \frac{\delta}{w_j} w_j - \frac{\delta}{w_k} w_k = \text{weight}(Y) \leq M.$$  

Finally, we compute

$$\text{profit}(Y') = \text{profit}(Y) + \frac{\delta p_j}{w_j} - \frac{\delta p_k}{w_k} = \text{profit}(Y) + \delta \left( \frac{p_j}{w_j} - \frac{p_k}{w_k} \right) > \text{profit}(Y),$$  

since $\delta > 0$ and $\frac{p_j}{w_j} > \frac{p_k}{w_k}$.  

Greedy Algorithms

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 \(N\).
A Greedy Algorithm for Coin Changing

Algorithm: \textit{GreedyCoinChanging}(D : array; T : integer)

\begin{itemize}
  \item comment: \( D = [d_1, \ldots, d_n] \)
  \item sort the coins so that \( d_1 > \cdots > d_n \)
  \item \( N \leftarrow 0 \)
  \item \textbf{for} \( i \leftarrow 1 \text{ to } n \)
    \begin{itemize}
      \item \( a_i \leftarrow \left\lfloor \frac{T}{d_i} \right\rfloor \)
      \item \textbf{do} \begin{align*}
        \text{\( T \leftarrow T - a_i d_i \)} \\
        \text{\( N \leftarrow N + a_i \)}
      \end{align*}
    \end{itemize}
  \item \textbf{if} \( T > 0 \)
    \begin{itemize}
      \item \textbf{then return} \( (\text{fail}) \)
      \item \textbf{else return} \( ([a_1, \ldots, a_n], N) \)
    \end{itemize}
\end{itemize}
Proof of Optimality for $D = [100, 25, 10, 5, 1]$

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 $T = 1, 2, 3, 4$. 
Proof of Optimality (cont.)

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.

$25 \leq T < 100$. Exercise.

$100 \leq T$. Exercise.
The Stable Matching Problem

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 property is called a stable matching.
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.
**Gale-Shapley Algorithm**

**Algorithm:** *Gale-Shapley* (*X*, *Y*, *pref*)

1. Let *Match* ← ∅
2. While there exists an unmatched *x*:
   a. Let *y* be the next element in *x*’s preference list
   b. If *y* is not matched
      i. *Match* ← *Match* ∪ {*x*, *y*}
   c. Else
      i. Suppose {*x*, *y*} ∈ *Match*
       a. If *y* prefers *x* to *x*:
          i. *Match* ← *Match* \ {*x*, *y*} ∪ {*x*, *y*}
       b. Else
          i. *Match* ← *Match* \ {*x*, *y*} ∪ {*x*, *y*}
             Comment: *x* is now unmatched
3. Return (*Match*)
Example

Suppose we have the following preference lists:

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

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

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>
Another Example

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 \textit{Gale-Shapley algorithm}.
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 \( y_j \).

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 that \( x_i \) proposed to \( y_\ell \) before he proposed 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.

All three cases are impossible, and hence we conclude that the resulting matching is stable.
Proof of Correctness (cont.)

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.

(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 $y_\ell$ 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?