CS 341: Algorithms

Douglas R. Stinson

David R. Cheriton School of Computer Science
University of Waterloo

Winter, 2017
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

1 Course Information
Course mechanics

- My Sections:
  - Section 2, T Th 10:00–11:20, RCH 207
  - Section 4, T Th 8:30–9:50, RCH 207

- My Scheduled Office Hours:
  - Th, 1:30–2:30
Course mechanics

- **Come to class!** Not all the material will be on the slides or in the text.

- **You will need an account in the student.cs environment**

- **The course website can be found at**
  
  https://www.student.cs.uwaterloo.ca/~cs341/

  - Syllabus, calendar, policies, etc. can be found there (or on Learn).
Learn and Piazza

- Slides and assignments will be available on the course website.
- Grades will be available on Learn.
- Discussion related to the course will take place on Piazza (piazza.com).
  - General course questions, announcements
  - Assignment-related questions
  - You will be getting an invitation via email to join Piazza in the first week of classes.
- Keep up with the information posted on the course website, Learn and Piazza.
### Courtesy

- Please silence cell phones and other mobile devices before coming to class.
- Questions are encouraged, but please refrain from talking in class – it is distracting to your classmates who are trying to listen to the lectures and to your professor who is trying to think, talk and write at the same time.
- Carefully consider whether using your laptop, ipad, smartphone, etc., in class will help you learn the material and follow the lectures.
- Do not play games, tweet, watch youtube videos, update your facebook page or use a mobile device in any other way that will distract your classmates.
Course syllabus

- You are expected to be familiar with the contents of the course syllabus
- Available on the course home page
- If you haven’t read it, read it as soon as possible
Plagiarism and academic offenses

- We take academic offences very seriously
- There is a good discussion of plagiarism online:
- Read this and understand it
  - Ignorance is no excuse!
  - Questions should be brought to instructor
- Plagiarism applies to both text and code
- You are free (even encouraged) to exchange ideas, but no sharing code or text
Plagiarism (2)

- Common mistakes
  - Excess collaboration with other students
    - Share ideas, but no design or code!
  - Using solutions from other sources (like for previous offerings of this course, maybe written by yourself)

- Possible penalties
  - First offense (for assignments; exams are harsher)
    - 0% for that assignment, -5% on final grade
  - Second offense
    - Expulsion is possible

- More information linked to from course syllabus
Grading scheme for CS 341

- Midterm (25%)
  - Thursday, March 2, 2017, 7:00–8:50 PM

- Assignments (25%)
  - There will be five assignments.
  - Work alone
  - See syllabus for reappraisal policies, academic integrity policy, and other details

- Final (50%)

- For medical conditions, you need to submit a Verification of Illness form.
Assignments

- All sections will have the same assignments, midterm and final exam.
- Assignments will be due at Noon on the due date.
- No late submissions will be accepted.
- You need to notify your instructor well before the due date of a severe, long-lasting or ongoing problem that prevents you from doing an assignment.
Assignment due dates

- Assignment 1: due Friday Jan. 20
- Assignment 2: due Friday Feb. 3
- Assignment 3: due Friday Feb. 17
- Assignment 4: due Friday March 10
- Assignment 5: due Friday March 31
Required textbook


- You are expected to know
  - entire textbook sections, as listed on course website or Learn
  - all the material presented in class
2 Introduction

- Algorithm Design and Analysis
- The Maximum Problem
- The Max-Min Problem
- The 3SUM Problem
- Definitions and Terminology
- Order Terminology
- Formulae
- Algorithm Analysis Techniques
In this course, we study the **design** and **analysis** of algorithms. “Analysis” refers to mathematical techniques for establishing both the **correctness** and **efficiency** of algorithms.

**Correctness**: We often want a formal proof of correctness of an algorithm we design. This might be accomplished through the use of **loop invariants** and mathematical induction.
Analysis of algorithms (cont.)

Efficiency: Given an algorithm $A$, we want to know how efficient it is. This includes several possible criteria:

- What is the asymptotic complexity of algorithm $A$?
- What is the exact number of specified computations done by $A$?
- How does the average-case complexity of $A$ compare to the worst-case complexity?
- Is $A$ the most efficient algorithm to solve the given problem? (For example, can we find a lower bound on the complexity of any algorithm to solve the given problem?)
- Are there problems that cannot be solved efficiently? This topic is addressed in the theory of NP-completeness.
- Are there problems that cannot be solved by any algorithm? Such problems are termed undecidable.
“Design” refers to **general strategies** for creating new algorithms. If we have good design strategies, then it will be easier to end up with correct and efficient algorithms. Also, we want to avoid using **ad hoc** algorithms that are hard to analyze and understand.

Here are some useful design strategies, many of which we will study:

- **divide-and-conquer**
- **greedy**
- **dynamic programming**
- **depth-first and breadth-first search**
- **local search** (not studied in this course)
- **linear programming** (not studied in this course)
The “Maximum” problem

**Problem**

**Maximum**

**Instance:** an array $A$ of $n$ integers,

$$A = [A[1], \ldots, A[n]].$$

**Find:** the maximum element in $A$.

The Maximum problem has an obvious simple solution.

**Algorithm:** $FindMaximum(A = [A[1], \ldots, A[n]])$

1. $max \leftarrow A[1]$
2. for $i \leftarrow 2$ to $n$
   - do 
     - if $A[i] > max$
       - then $max \leftarrow A[i]$
   - end do
3. return ($max$)
Correctness of FindMaximum

How can we formally prove that $\textit{FindMaximum}$ is correct?

**Claim:** At the end of iteration $i$ ($i = 2, \ldots, n$), the current value of $\textit{max}$ is the maximum element in $[A[1], \ldots, A[i]]$.

The claim can be proven by induction. The base case, when $i = 2$, is obvious.

Now we make an induction assumption that the claim is true for $i = j$, where $2 \leq j \leq n - 1$, and we prove that the claim is true for $i = j + 1$ (fill in the details!).

When $j = n$ we are done and the correctness of $\textit{FindMaximum}$ is proven.
Analysis of FindMaximum

It is obvious that the complexity of FindMaximum is $\Theta(n)$. More precisely, we can observe that the number of comparisons of array elements done by FindMaximum is exactly $n - 1$.

It turns out that FindMaximum is optimal with respect to the number of comparisons of array elements.

That is, any algorithm that correctly solves the Maximum problem for an array of $n$ elements requires at least $n - 1$ comparisons of array elements. How can we prove this assertion?
The “Max-Min” problem

Problem

Max-Min

Instance: an array \( A \) of \( n \) integers, \( A = [A[1], \ldots, A[n]] \).

Find: the maximum and the minimum element in \( A \).

The Max-Min problem also has an obvious simple solution.

Algorithm: \( \text{FindMaximumAndMinimum}(A = [A[1], \ldots, A[n]]) \)

\[
\begin{align*}
\text{max} & \leftarrow A[1] \\
\text{min} & \leftarrow A[1] \\
\text{for } i & \leftarrow 2 \text{ to } n \\
\quad \text{do } & \begin{cases} \\
\quad \text{if } A[i] > \text{max} \quad & \text{then } \text{max} \leftarrow A[i] \\
\quad \text{if } A[i] < \text{min} \quad & \text{then } \text{min} \leftarrow A[i] \\
\quad \end{cases} \\
\text{return } (\text{max}, \text{min})
\end{align*}
\]
Analysis of FindMaximumAndMinimum

Exercise: Give a formal proof by induction that FindMaximumAndMinimum is correct.

The complexity of FindMaximumAndMinimum is $\Theta(n)$

More precisely, FindMaximumAndMinimum requires $2n - 2$ comparisons of array elements given an array of size $n$.

The complexity is optimal (why?), but there are algorithms to solve the Max-Min problem which require fewer comparisons of array elements than FindMaximumAndMinimum.

Note: An algorithm requiring fewer comparisons of array elements may or may not be faster than FindMaximumAndMinimum.
A more significant improvement

With some ingenuity, we can actually reduce the number of comparisons of array elements by (roughly) 25%.

Suppose \( n \) is even and we consider the elements two at a time. Initially, we compare the first two elements and initialize maximum and minimum values. (One comparison is required here.)

Then, each time we compare a new pair of elements, we subsequently compare the larger of the two elements to the current maximum and the smaller of the two to the current minimum. (Three comparisons are done here to process two array elements.)

This yields an algorithm requiring a total of \( 3n/2 - 2 \) comparisons.
An improved algorithm

Algorithm: \textit{ImprovedFindMaximumAndMinimum}(A)

comment: assume $n$ is even

\begin{verbatim}
\{ \begin{align*}
max & \leftarrow A[1] \\
min & \leftarrow A[2]
\end{align*} \}
else \\
\{ \begin{align*}
max & \leftarrow A[2] \\
min & \leftarrow A[1]
\end{align*} \}
\end{verbatim}

for $i \leftarrow 2$ to $n/2$

\begin{verbatim}
\{ \begin{align*}
\text{ then } \{ \begin{align*}
\text{if } A[2i - 1] > max \text{ then } max & \leftarrow A[2i - 1] \\
\text{if } A[2i] < min \text{ then } min & \leftarrow A[2i]
\end{align*} \}
\text{else } \{ \begin{align*}
\text{if } A[2i] > max \text{ then } max & \leftarrow A[2i] \\
\text{if } A[2i - 1] < min \text{ then } min & \leftarrow A[2i - 1]
\end{align*} \}
\end{align*} \}
\end{verbatim}

return $(max, min)$
Optimality of the previous algorithm

It is possible to prove that any algorithm that solves the Max-Min problem requires at least $\frac{3n}{2} - 2$ comparisons of array elements in the worst case.

Therefore the algorithm $\text{ImprovedFindMaximumAndMinimum}$ is in fact optimal with respect to the number of comparisons of array elements required.
The “3SUM” problem

**Problem**

**3SUM**

**Instance:** an array $A$ of $n$ distinct integers, $A = [A[1], \ldots, A[n]]$.

**Question:** do there exist three elements in $A$ that sum to 0?

The 3SUM problem also has an obvious algorithm to solve it.

**Algorithm:** *Trivial3SUM*($A = [A[1], \ldots, A[n]]$)

for $i \leftarrow 1$ to $n - 2$

\[
\begin{array}{l}
\text{for } j \leftarrow i + 1 \text{ to } n - 1 \\
\quad \text{do } \left\{ \\
\quad \quad \text{for } k \leftarrow j + 1 \text{ to } n \\
\quad \quad \quad \text{do } \left\{ \\
\quad \quad \quad \quad \text{if } A[i] + A[j] + A[k] = 0 \\
\quad \quad \quad \quad \text{then output } (i,j,k) \\
\quad \quad \quad \right. \\
\quad \right. \\
\end{array}
\]

The complexity of *Trivial3SUM* is $O(n^3)$. 
A possible improvement

Instead of having three nested loops, suppose we have two nested loops (with indices $i$ and $j$, say) and then we search for an $A[k]$ for which $A[i] + A[j] + A[k] = 0$.

If we try all possible $k$-values, then we basically have the previous algorithm.

What can we do to make the search more efficient?

What effect does this have on the complexity of the resulting algorithm?
An improved algorithm for the "3SUM" problem

Algorithm: \textit{Improved3SUM}(A = [A[1], \ldots, A[n]])

sort $A$ in increasing order

\begin{verbatim}
for $i \leftarrow 1$ to $n - 2$
  for $j \leftarrow i + 1$ to $n - 1$
          if the search is successful, output $(i, j, k)$ \}
\end{verbatim}

The complexity of \textit{Improved3SUM} is $O(n \log n + n^2 \log n) = O(n^2 \log n)$. 
A further improvement

In Improved3SUM, we pre-sorted the array $A$, which enabled us to do binary searches.

There is a better way to make use of the sorted array, however . . .


We start with $j = i + 1$ and $k = n$.

At any stage of the algorithm, we either increment $j$ or decrement $k$ (or both, if $A[i] + A[j] + A[k] = 0$).

Does this remind you of a familiar algorithm you have seen in CS 240?

The resulting algorithm will have complexity $O(n \log n + n^2) = O(n^2)$. 
A quadratic time algorithm for the “3SUM” problem

Algorithm: Quadratic3SUM(\(A = [A[1], \ldots, A[n]]\))

sort \(A\) in increasing order

for \(i \leftarrow 1\) to \(n - 2\)

\[
\begin{align*}
&j \leftarrow i + 1 \\
&k \leftarrow n \\
&\text{while } j < k \\
&\quad \text{do} \begin{cases} \\
&\quad \text{if } S < 0 \quad \text{then } j \leftarrow j + 1 \\
&\quad \text{else if } S > 0 \quad \text{then } k \leftarrow k - 1 \\
&\quad \text{else} \begin{cases} \\
&\quad \quad j \leftarrow j + 1 \\
&\quad \quad k \leftarrow k - 1 \\
&\quad \end{cases} \\
&\end{cases}
\end{align*}
\]
Problems

**Problem**: Given a problem instance $I$ for a problem $P$, carry out a particular computational task.

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

**Problem Solution**: Output (correct answer) for the specified problem.

**Size of a problem instance**: $\text{Size}(I)$ is a positive integer which is a measure of the size of the instance $I$. 
**Algorithms and Programs**

**Algorithm:** An algorithm is a step-by-step process (e.g., described in *pseudocode*) for carrying out a series of computations, given some appropriate input.

**Algorithm solving a problem:** An Algorithm $A$ solves a problem $P$ if, for every instance $I$ of $P$, $A$ finds a valid solution for the instance $I$ in finite time.

**Program:** A program is an *implementation* of an algorithm using a specified computer language.
Running Time

Running Time of a Program: $T_M(I)$ denotes the running time (in seconds) of a program $M$ on a problem instance $I$.

Worst-case Running Time as a Function of Input Size: $T_M(n)$ denotes the maximum running time of program $M$ on instances of size $n$:

$$T_M(n) = \max\{T_M(I) : \text{Size}(I) = n\}.$$  

Average-case Running Time as a Function of Input Size: $T_M^{\text{avg}}(n)$ denotes the average running time of program $M$ over all instances of size $n$:

$$T_M^{\text{avg}}(n) = \frac{1}{|\{I : \text{Size}(I) = n\}|} \sum_{\{I : \text{Size}(I) = n\}} T_M(I).$$
Complexity

**Worst-case complexity of an algorithm:** Let $f : \mathbb{Z}^+ \rightarrow \mathbb{R}$. An algorithm $A$ has **worst-case complexity** $f(n)$ if there exists a program $M$ implementing the algorithm $A$ such that $T_M(n) \in \Theta(f(n))$.

**Average-case complexity of an algorithm:** Let $f : \mathbb{Z}^+ \rightarrow \mathbb{R}$. An algorithm $A$ has **average-case complexity** $f(n)$ if there exists a program $M$ implementing the algorithm $A$ such that $T_{M}^{avg}(n) \in \Theta(f(n))$. 
Running Time vs Complexity

**Running time** can only be determined by implementing a program and running it on a specific computer.

Running time is influenced by many factors, including the programming language, processor, operating system, etc.

**Complexity** (AKA growth rate) can be analyzed by high-level mathematical analysis. It is independent of the above-mentioned factors affecting running time.

Complexity is a less precise measure than running time since it is asymptotic and it incorporates unspecified constant factors and unspecified lower order terms.

However, if algorithm $A$ has lower complexity than algorithm $B$, then a program implementing algorithm $A$ will be faster than a program implementing algorithm $B$ for sufficiently large inputs.
Order Notation

\( O \)-notation:

\[ f(n) \in O(g(n)) \text{ if there exist constants } c > 0 \text{ and } n_0 > 0 \text{ such that } 0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0. \]

Here the complexity of \( f \) is **not higher** than the complexity of \( g \).

\( \Omega \)-notation:

\[ f(n) \in \Omega(g(n)) \text{ if there exist constants } c > 0 \text{ and } n_0 > 0 \text{ such that } 0 \leq c g(n) \leq f(n) \text{ for all } n \geq n_0. \]

Here the complexity of \( f \) is **not lower** than the complexity of \( g \).

\( \Theta \)-notation:

\[ f(n) \in \Theta(g(n)) \text{ if there exist constants } c_1, c_2 > 0 \text{ and } n_0 > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0. \]

Here \( f \) and \( g \) have the **same complexity**.
Order Notation (cont.)

$o$-notation:

$f(n) \in o(g(n))$ if for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) \leq c g(n)$ for all $n \geq n_0$.

Here $f$ has **lower complexity** than $g$.

$\omega$-notation:

$f(n) \in \omega(g(n))$ if for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq c g(n) \leq f(n)$ for all $n \geq n_0$.

Here $f$ has **higher complexity** than $g$. 
Suppose that \( f(n) > 0 \) and \( g(n) > 0 \) for all \( n \geq n_0 \). Suppose that
\[
L = \lim_{n \to \infty} \frac{f(n)}{g(n)}.
\]
Then
\[
f(n) \in \begin{cases} 
  o(g(n)) & \text{if } L = 0 \\
  \Theta(g(n)) & \text{if } 0 < L < \infty \\
  \omega(g(n)) & \text{if } L = \infty.
\end{cases}
\]
Relationships between Order Notations

\[ f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n)) \]
\[ f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n)) \]
\[ f(n) \in o(g(n)) \iff g(n) \in \omega(f(n)) \]

\[ f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n)) \]
\[ f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n)) \]
\[ f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n)) \]
Algebra of Order Notations

“Maximum” rules: Suppose that \( f(n) > 0 \) and \( g(n) > 0 \) for all \( n \geq n_0 \). Then:

\[
O(f(n) + g(n)) = O(\max\{f(n), g(n)\})
\]

\[
\Theta(f(n) + g(n)) = \Theta(\max\{f(n), g(n)\})
\]

\[
\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})
\]

“Summation” rules:

\[
O\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} O(f(i))
\]

\[
\Theta\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} \Theta(f(i))
\]

\[
\Omega\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} \Omega(f(i))
\]
Sequences

Arithmetic sequence:

\[ \sum_{i=0}^{n-1} (a + di) = na + \frac{d n (n - 1)}{2} \in \Theta(n^2). \]

Geometric sequence:

\[ \sum_{i=0}^{n-1} a r^i = \begin{cases} a \frac{r^n - 1}{r - 1} \in \Theta(r^n) & \text{if } r > 1 \\ na \in \Theta(n) & \text{if } r = 1 \\ a \frac{1 - r^n}{1 - r} \in \Theta(1) & \text{if } 0 < r < 1. \end{cases} \]

Arithmetic-geometric sequence:

\[ \sum_{i=0}^{n-1} (a + di) r^i = \frac{a}{1 - r} - \frac{(a + (n - 1)d) r^n}{1 - r} + \frac{dr (1 - r^{n-1})}{(1 - r)^2} \]

provided that \( r \neq 1. \)
Harmonic sequence:

\[ H_n = \sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n) \]

More precisely, it is possible to prove that

\[ \lim_{n \to \infty} (H_n - \ln n) = \gamma, \]

where \( \gamma \approx 0.57721 \) is Euler's constant.
Miscellaneous Formulae

\[ \log_b xy = \log_b x + \log_b y \]
\[ \log_b x/y = \log_b x - \log_b y \]
\[ \log_b 1/x = -\log_b x \]
\[ \log_b x^y = y \log_b x \]
\[ \log_b a = \frac{1}{\log_a b} \]
\[ \log_b a = \frac{\log_c a}{\log_c b} \]
\[ a^{\log_b c} = c^{\log_b a} \]
\[ n! \in \Theta\left(n^{n+1/2}e^{-n}\right) \]
\[ \log n! \in \Theta(n \log n) \]
Techniques for Algorithm Analysis

Two general strategies are as follows:

- Use $\Theta$-bounds throughout the analysis and thereby obtain a $\Theta$-bound for the complexity of the algorithm.

- Prove a $O$-bound and a matching $\Omega$-bound separately to get a $\Theta$-bound. Sometimes this technique is easier because arguments for $O$-bounds may use simpler upper bounds (and arguments for $\Omega$-bounds may use simpler lower bounds) than arguments for $\Theta$-bounds do.
Techniques for Loop Analysis

Identify elementary operations that require constant time (denoted $\Theta(1)$ time).

The complexity of a loop is expressed as the sum of the complexities of each iteration of the loop.

Analyze independent loops separately, and then add the results: use “maximum rules” and simplify whenever possible.

If loops are nested, start with the innermost loop and proceed outwards. In general, this kind of analysis requires evaluation of nested summations.
Example of Loop Analysis

Algorithm: \textit{LoopAnalysis1}(n : integer)

\begin{enumerate}
  \item \texttt{sum} $\leftarrow$ 0
  \item \texttt{for} \texttt{i} $\leftarrow$ 1 \texttt{to} \texttt{n}
    \begin{enumerate}
      \item \texttt{for} \texttt{j} $\leftarrow$ 1 \texttt{to} \texttt{i}
        \begin{enumerate}
          \item \texttt{do} \texttt{sum} $\leftarrow$ \texttt{sum} + \texttt{(i - j)}^2
          \item \texttt{do} \texttt{sum} $\leftarrow$ \texttt{sum}/\texttt{i}
        \end{enumerate}
    \end{enumerate}
  \item \texttt{return} \texttt{(sum)}
\end{enumerate}

$\Theta$-bound analysis

\begin{enumerate}
  \item $\Theta(1)$
  \item Complexity of inner \texttt{for} loop: $\Theta(i)$
    Complexity of outer \texttt{for} loop: $\sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$
    Note: $\sum_{i=1}^{n} i = n(n + 1)/2$
  \item $\Theta(1)$
\end{enumerate}

Total $\Theta(n^2)$
Example of Loop Analysis (cont.)

Proving separate $O$- and $\Omega$-bounds

We focus on the two nested $\textbf{for}$ loops (i.e., (2)).

The total number of iterations is $\sum_{i=1}^{n} i$, with $\Theta(1)$ time per iteration.

Upper bound:

$$\sum_{i=1}^{n} O(i) \leq \sum_{i=1}^{n} O(n) = O(n^2).$$

Lower bound:

$$\sum_{i=1}^{n} \Omega(i) \geq \sum_{i=n/2}^{n} \Omega(i) \geq \sum_{i=n/2}^{n} \Omega(n/2) = \Omega(n^2/4) = \Omega(n^2).$$

Since the upper and lower bounds match, the complexity is $\Theta(n^2)$. 
Another Example of Loop Analysis

**Algorithm:** LoopAnalysis2 \((A: array; n: integer)\)

\[
\text{max} \leftarrow 0 \\
\text{for } i \leftarrow 1 \text{ to } n \\
\quad \begin{cases} 
\quad \text{for } j \leftarrow i \text{ to } n \\
\quad \quad \text{do } \begin{cases} 
\quad \quad \quad \text{sum} \leftarrow 0 \\
\quad \quad \quad \text{for } k \leftarrow i \text{ to } j \\
\quad \quad \quad \quad \text{do } \begin{cases} 
\quad \quad \quad \quad \text{if } \text{sum} > \text{max} \\
\quad \quad \quad \quad \quad \text{then } \text{max} \leftarrow \text{sum} 
\quad \quad \quad \end{cases} 
\quad \quad \quad \end{cases} 
\quad \end{cases} 
\]

return \((\text{max})\)
Yet Another Example of Loop Analysis

Algorithm: \textit{LoopAnalysis3}(n : integer)

\begin{verbatim}
sum ← 0
for i ← 1 to n
    j ← i
    while j ≥ 1
        do { sum ← sum + i/j
              j ← ⌊j/2⌋
        }
return (sum)
\end{verbatim}
# Table of Contents

3 **Divide-and-Conquer Algorithms**
   - Recurrence Relations
   - Master Theorem
   - Divide-and-Conquer Design Strategy
   - Mergesort
   - Divide-and-Conquer Recurrence Relations
   - Max-Min Problem
   - Non-dominated Points
   - Closest Pair
   - Multiprecision Multiplication
   - Matrix Multiplication
   - Selection and Median
Recurrence Relations

Suppose \( a_1, a_2, \ldots, \) is an infinite sequence of real numbers.

A recurrence relation is a formula that expresses a general term \( a_n \) in terms of one or more previous terms \( a_1, \ldots, a_{n-1} \).

A recurrence relation will also specify one or more initial values starting at \( a_1 \).

Solving a recurrence relation means finding a formula for \( a_n \) that does not involve any previous terms \( a_1, \ldots, a_{n-1} \).

There are many methods of solving recurrence relations. Two important methods are guess-and-check and the recursion tree method.

We will make extensive use of the recursion tree method. However, we first take a quick look at the guess-and-check method.
Guess-and-check Method

**step 1**  Tabulate some values $a_1, a_2, \ldots$ using the recurrence relation.

**step 2**  Guess that the solution $a_n$ has a specific form, involving undetermined constants.

**step 3**  Use $a_1, a_2, \ldots$ to determine specific values for the unspecified constants.

**step 4**  Use induction to prove your guess for $a_n$ is correct.
Example of the Guess-and-check Method

Suppose we have the recurrence \( T(n) = T(n - 1) + 6n - 5, \ T(0) = 4 \).

We compute a few values: \( T(1) = 5, \ T(2) = 12, \ T(3) = 25, \ T(4) = 44 \).

If we are sufficiently perspicacious, we might guess that \( T(n) \) is a quadratic function, e.g., \( T(n) = an^2 + bn + c \).

Next, we use \( T(0) = 4, \ T(1) = 5, \ T(2) = 12 \) to compute \( a, b \) and \( c \) by solving three equations in three unknowns.

We get \( a = 3, \ b = -2, \ c = 4 \).

Now we can use induction to prove that \( T(n) = 3n^2 - 2n + 4 \) for all \( n \geq 0 \).
Recursion Tree Method

The following recurrence relation arises in the analysis of Mergesort:

\[
T(n) = \begin{cases} 
2T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \text{ is a power of } 2 \\
 d & \text{if } n = 1,
\end{cases}
\]

where \( c \) and \( d \) are constants.

We can solve this recurrence relation when \( n \) is a power of two, by constructing a recursion tree, as follows:

**step 1** Start with a one-node tree, say \( N \), having the value \( T(n) \).

**step 2** Grow two children of \( N \). These children, say \( N_1 \) and \( N_2 \), have the value \( T(n/2) \), and the value of \( N \) is replaced by \( cn \).

**step 3** Repeat this process recursively, terminating when a node receives the value \( T(1) = d \).

**step 4** Sum the values on each level of the tree, and then compute the sum of all these sums; the result is \( T(n) \).
Master Theorem

The Master Theorem provides a formula for the solution of many recurrence relations typically encountered in the analysis of algorithms. The following is a simplified version of the Master Theorem:

**Theorem**

Suppose that $a \geq 1$ and $b > 1$. Consider the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^y),$$

(1)

where $n$ is a power of $b$. Denote $x = \log_b a$. Then

$$T(n) \in \begin{cases} 
\Theta(n^x) & \text{if } y < x \\
\Theta(n^x \log n) & \text{if } y = x \\
\Theta(n^y) & \text{if } y > x.
\end{cases}$$
Proof of the Master Theorem (simplified version)

Suppose that $a \geq 1$ and $b \geq 2$ are integers and

$$T(n) = aT\left(\frac{n}{b}\right) + cn^y, \quad T(1) = d.$$ 

Let $n = b^j$.

<table>
<thead>
<tr>
<th>level</th>
<th># nodes</th>
<th>value at each node</th>
<th>value of the level</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j$</td>
<td>1</td>
<td>$cn^y$</td>
<td>$cn^y$</td>
</tr>
<tr>
<td>$j-1$</td>
<td>$a$</td>
<td>$c(n/b)^y$</td>
<td>$ca(n/b)^y$</td>
</tr>
<tr>
<td>$j-2$</td>
<td>$a^2$</td>
<td>$c(n/b^2)^y$</td>
<td>$ca^2(n/b^2)^y$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>1</td>
<td>$a^{j-1}$</td>
<td>$c(n/b^{j-1})^y$</td>
<td>$ca^{j-1}(n/b^{j-1})^y$</td>
</tr>
<tr>
<td>0</td>
<td>$a^j$</td>
<td>$d$</td>
<td>$da^j$</td>
</tr>
</tbody>
</table>
Computing $T(n)$

Summing the values at all levels of the recursion tree, we have that

$$T(n) = d a^j + c n^y \sum_{i=0}^{j-1} \left( \frac{a}{b^y} \right)^i.$$

Recall that $b^x = a$ and $n = b^j$. Hence $a^j = (b^x)^j = (b^j)^x = n^x$.

The formula for $T(n)$ is a geometric sequence with ratio $r = a/b^y = b^{x-y}$:

$$T(n) = d n^x + c n^y \sum_{i=0}^{j-1} r^i.$$

There are three cases, depending on whether $r > 1$, $r = 1$ or $r < 1$. 
## Complexity of $T(n)$

<table>
<thead>
<tr>
<th>case</th>
<th>$r$</th>
<th>$y, x$</th>
<th>complexity of $T(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>heavy leaves</td>
<td>$r &gt; 1$</td>
<td>$y &lt; x$</td>
<td>$T(n) \in \Theta(n^x)$</td>
</tr>
<tr>
<td>balanced</td>
<td>$r = 1$</td>
<td>$y = x$</td>
<td>$T(n) \in \Theta(n^x \log n)$</td>
</tr>
<tr>
<td>heavy top</td>
<td>$r &lt; 1$</td>
<td>$y &gt; x$</td>
<td>$T(n) \in \Theta(n^y)$</td>
</tr>
</tbody>
</table>

**heavy leaves** means that the value of the recursion tree is dominated by the values of the leaf nodes.

**balanced** means that the values of the levels of the recursion tree are constant (except for the last level).

**heavy top** means that the value of the recursion tree is dominated by the value of the root node.
Master Theorem (modified general version)

**Theorem**

Suppose that $a \geq 1$ and $b > 1$. Consider the recurrence

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n),$$

where $n$ is a power of $b$. Denote $x = \log_b a$. Then

$$T(n) \in \begin{cases} 
\Theta(n^x) & \text{if } f(n) \in O(n^{x-\epsilon}) \text{ for some } \epsilon > 0 \\
\Theta(n^x \log n) & \text{if } f(n) \in \Theta(n^x) \\
\Theta(f(n)) & \text{if } f(n)/n^{x+\epsilon} \text{ is an increasing function of } n \\
\Theta(f(n)) & \text{for some } \epsilon > 0.
\end{cases}$$
The Divide-and-Conquer Design Strategy

divide: Given a problem instance $I$, construct one or more smaller problem instances, denoted $I_1, \ldots, I_a$ (these are called subproblems). Usually, we want the size of these subproblems to be small compared to the size of $I$, e.g., half the size.

conquer: For $1 \leq j \leq a$, solve instance $I_j$ recursively, obtaining solutions $S_1, \ldots, S_a$.

combine: Given $S_1, \ldots, S_a$, use an appropriate combining function to find the solution $S$ to the problem instance $I$, i.e., $S \leftarrow \text{Combine}(S_1, \ldots, S_a)$. 
Example: Design of Mergesort

Here, a problem instance consists of an array $A$ of $n$ integers, which we want to sort in increasing order. The size of the problem instance is $n$.

divide: Split $A$ into two subarrays: $A_L$ consists of the first $\lceil \frac{n}{2} \rceil$ elements in $A$ and $A_R$ consists of the last $\lfloor \frac{n}{2} \rfloor$ elements in $A$.

conquer: Run $Mergesort$ on $A_L$ and $A_R$.

combine: After $A_L$ and $A_R$ have been sorted, use a function $Merge$ to merge $A_L$ and $A_R$ into a single sorted array. Recall that this can be done in time $\Theta(n)$ with a single pass through $A_L$ and $A_R$. We simply keep track of the “current” element of $A_L$ and $A_R$, always copying the smaller one into the sorted array.
Mergesort

Algorithm: \textit{Mergesort}(A : array; n : integer)

if \( n = 1 \)
then \( S \leftarrow A \)
\begin{align*}
  n_L &\leftarrow \left\lfloor \frac{n}{2} \right\rfloor \\
  n_R &\left\lceil \frac{n}{2} \right\rceil \\
  A_L &\left[ A[1], \ldots, A[n_L] \right] \\
  A_R &\left[ A[n_L + 1], \ldots, A[n] \right] \\
  S_L &\leftarrow \textit{Mergesort}(A_L, n_L) \\
  S_R &\leftarrow \textit{Mergesort}(A_R, n_R) \\
  S &\leftarrow \textit{Merge}(S_L, n_L, S_R, n_R) \\
\end{align*}

return \((S, n)\)
Analysis of Mergesort

Let $T(n)$ denote the time to run *Mergesort* on an array of length $n$.

- **divide** takes time $\Theta(n)$
- **conquer** takes time $T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$
- **combine** takes time $\Theta(n)$

Recurrence relation:

$$T(n) = \begin{cases} 
T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{if } n > 1 \\
\Theta(1) & \text{if } n = 1.
\end{cases}$$
Sloppy and Exact Recurrence Relations

It is simpler to replace the $\Theta(n)$ term by $cn$, where $c$ is an unspecified constant. The resulting recurrence relation is called the exact recurrence.

\[
T(n) = \begin{cases} 
T\left(\lceil \frac{n}{2} \rceil \right) + T\left(\lfloor \frac{n}{2} \rfloor \right) + cn & \text{if } n > 1 \\
d & \text{if } n = 1
\end{cases}
\]

If we then remove the floors and ceilings, we obtain the so-called sloppy recurrence:

\[
T(n) = \begin{cases} 
2T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \\
d & \text{if } n = 1
\end{cases}
\]

The exact and sloppy recurrences are identical when $n$ is a power of two. Further, the sloppy recurrence makes sense only when $n$ is a power of two.
Complexity of the Solution to the Exact Recurrence

The Master Theorem provides the \textbf{exact} solution of the recurrence when $n = 2^j$ (it is in fact a \textbf{proof} for these values of $n$).

We can express this solution (for powers of $2$) as a function of $n$, using $\Theta$-notation.

It can be shown that the resulting function of $n$ will in fact yield the \textbf{complexity} of the solution of the exact recurrence for \textbf{all values} of $n$.

This derivation of the complexity of $T(n)$ is \textbf{not a proof}, however. If a rigorous mathematical proof is required, then it is necessary to use \textbf{induction} along with the \textbf{exact recurrence}. 
The Max-Min Problem

Let’s design a divide-and-conquer algorithm for the Max-Min problem.

**Divide:** Suppose we split $A$ into two equal-sized subarrays, $A_L$ and $A_R$.

**Conquer:** We find the maximum and minimum elements in each subarray recursively, obtaining $\text{max}_L$, $\text{min}_L$, $\text{max}_R$, and $\text{min}_R$.

**Combine:** Then we can easily “combine” the solutions to the two subproblems to solve the original problem instance:

$$\text{max} \leftarrow \max\{\text{max}_L, \text{max}_R\}$$

and

$$\text{min} \leftarrow \min\{\text{min}_L, \text{min}_R\}$$
The Max-Min Problem (cont.)

The recurrence relation describing the complexity of the running time is
\[ T(n) = 2T(n/2) + \Theta(1). \]

The Master Theorem shows that the \( T(n) \in \Theta(n) \).

However, we can also count the exact number of comparisons done by
the algorithm, obtaining the (sloppy) recurrence
\[ C(n) = 2C(n/2) + 2, \quad C(2) = 1. \]

For \( n \) a power of 2, the solution to this recurrence relation is
\[ C(n) = 3n/2 - 2, \] so the divide-and-conquer algorithm is optimal for
these values of \( n \) (see slide 26).
Non-dominated Points

Given two points \((x_1, y_1), (x_2, y_2)\) in the Euclidean plane, we say that \((x_1, y_1)\) dominates \((x_2, y_2)\) if \(x_1 \geq x_2\) and \(y_1 \geq y_2\).

**Problem**

**Non-dominated Points**

**Instance:** A set \(S\) of \(n\) points in the Euclidean plane, say \(S = \{S[1], \ldots, S[n]\}\).

**Question:** Find all the non-dominated points in \(S\), i.e., all the points that are not dominated by any other point in \(S\).

Non-dominated Points has a trivial \(\Theta(n^2)\) algorithm to solve it, based on comparing all pairs of points in \(S\). Can we do better?
Problem Decomposition

Observe that the non-dominated points form a staircase such that all the other points are “under” the staircase.

Suppose we pre-sort the points in $S$ with respect to their $x$-co-ordinates. This takes time $\Theta(n \log n)$.

Divide: Let the first $n/2$ points be denoted $S_1$ and let the last $n/2$ points be denoted $S_2$.

Conquer: Recursively solve the subproblems defined by the two instances $S_1$ and $S_2$.

Combine: Given the non-dominated points in $S_1$ and the non-dominated points in $S_2$, how do we find the non-dominated points in $S$?

Observe that no point in $S_1$ dominates a point in $S_2$.

Therefore we only need to eliminate the points in $S_1$ that are dominated by a point in $S_2$. This can be done in time $O(n)$. 
Non-dominated Points

Algorithm: \textit{Non-dominated}(S)

\textbf{comment:} the \(n\) points in \(S\) are pre-sorted WRT their \(x\)-co-ordinates

\textbf{if} \(n = 1\) \textbf{then return} \((\{S[1]\})\)

\textbf{else}
\[
\begin{align*}
\{Q[1], \ldots, Q[\ell]\} & \leftarrow \text{Non-dominated}(\{S[1], \ldots, S[\lfloor n/2 \rfloor]\}) \\
\{(R[1], \ldots, R[m]\} & \leftarrow \text{Non-dominated}(\{S[\lfloor n/2 \rfloor + 1], \ldots, S[n]\})
\end{align*}
\]

\textbf{else}

\(i \leftarrow 1\)

\textbf{while} \(i \leq \ell\ \text{and} \ Q[i].y > R[1].y\)

\textbf{do} \(i \leftarrow i + 1\)

\textbf{return} \((\{Q[1], \ldots, Q[i - 1], R[1], \ldots, R[m]\})\)
Closest Pair

Problem

Closest Pair

Instance: a set $Q$ of $n$ distinct points in the Euclidean plane,

$$Q = \{Q[1], \ldots, Q[n]\}.$$ 

Find: Two distinct points $Q[i] = (x, y), Q[j] = (x', y')$ such that the Euclidean distance

$$\sqrt{(x' - x)^2 + (y' - y)^2}$$

is minimized.
Closest Pair: Problem Decomposition

Suppose we presort the points in $Q$ with respect to their $x$-coordinates (this takes time $\Theta(n \log n)$).

Then we can easily find the vertical line that partitions the set of points $Q$ into two sets of size $n/2$: this line has equation $x = Q[m].x$, where $m = n/2$.

The set $Q$ is global with respect to the recursive procedure $ClosestPair1$.

At any given point in the recursion, we are examining a subarray $(Q[\ell], \ldots, Q[r])$, and $m = \lfloor (\ell + r)/2 \rfloor$.

We call $ClosestPair1(1, n)$ to solve the given problem instance.
Closest Pair: Solution 1

Algorithm: \textit{ClosestPair1}(\ell, r)

\begin{enumerate}
  \item \textbf{if} \( \ell = r \) \textbf{then}\quad \delta \leftarrow \infty
  \begin{cases}
    m \leftarrow \lfloor (\ell + r)/2 \rfloor \\
    \delta_L \leftarrow \text{ClosestPair1}(\ell, m) \\
    \delta_R \leftarrow \text{ClosestPair1}(m + 1, r)
  \end{cases}
  \item \textbf{else}\quad \delta \leftarrow \min\{\delta_L, \delta_R\}
    \begin{cases}
      R \leftarrow \text{SelectCandidates}(\ell, r, \delta, Q[m].x) \\
      R \leftarrow \text{SortY}(R) \\
      \delta \leftarrow \text{CheckStrip}(R, \delta)
    \end{cases}
\end{enumerate}

return \((\delta)\)
Selecting Candidates from the Vertical Strip

Algorithm: \textit{SelectCandidates}(\ell, r, \delta, xmid)

\begin{align*}
    j &\leftarrow 0 \\
    \text{for } i &\leftarrow \ell \text{ to } r \\
    &\quad \text{if } |Q[i].x - xmid| \leq \delta \\
    &\quad \quad \text{do } \\
    &\quad \quad \quad \text{then } \\
    &\quad \quad \quad \quad j \leftarrow j + 1 \\
    &\quad \quad \quad R[j] \leftarrow Q[i] \\
    \text{return } (R)
\end{align*}
Checking the Vertical Strip

Algorithm: \textit{CheckStrip}(R, \delta)

\begin{align*}
  t & \leftarrow \text{size}(R) \\
  \delta' & \leftarrow \delta \\
  \text{for } j & \leftarrow 1 \text{ to } t - 1 \\
  \text{for } k & \leftarrow j + 1 \text{ to } \min\{t, j + 7\} \\
  \text{do } \{ \\
  x & \leftarrow R[j].x \\
  x' & \leftarrow R[k].x \\
  y & \leftarrow R[j].y \\
  y' & \leftarrow R[k].y \\
  \delta' & \leftarrow \min \left\{ \delta', \sqrt{(x' - x)^2 + (y' - y)^2} \right\} \\
  \text{do } \}
\end{align*}
Closest Pair: Solution 2

Algorithm: $ClosestPair2(\ell, r)$

if $\ell = r$ then $\delta \leftarrow \infty$

else

\[
\begin{cases}
    m \leftarrow \lfloor (\ell + r)/2 \rfloor \\
    X_{mid} \leftarrow Q[m].x \\
    \delta_L \leftarrow ClosestPair2(\ell, m) \\
    \text{comment: } Q[\ell], \ldots, Q[m] \text{ is sorted WRT } y\text{-coordinates}
    \\
    \delta_R \leftarrow ClosestPair2(m + 1, r) \\
    \text{comment: } Q[m + 1], \ldots, Q[r] \text{ is sorted WRT } y\text{-coordinates}
    \\
    \delta \leftarrow \min\{\delta_L, \delta_R\} \\
    \text{Merge}(\ell, m, r) \\
    R \leftarrow SelectCandidates(\ell, r, \delta, X_{mid}) \\
    \delta \leftarrow CheckStrip(R, \delta)
\end{cases}
\]

return $(\delta)$
Multiprecision Multiplication

Problem

Multiprecision Multiplication

Instance: Two $k$-bit positive integers, $X$ and $Y$, having binary representations

\[ X = [X[k-1], \ldots, X[0]] \]

and

\[ Y = [Y[k-1], \ldots, Y[0]]. \]

Question: Compute the $2k$-bit positive integer $Z = XY$, where

\[ Z = (Z[2k-1], \ldots, Z[0]). \]

We are interested in the bit complexity of algorithms that solve Multiprecision Multiplication, which means that the complexity is expressed as a function of $k$ (the size of the problem instance is $2k$ bits).
Not-So-Fast D&C Multiprecision Multiplication

Algorithm: \textit{NotSoFastMultiply}(X, Y, k)

if \(k = 1\)
then \(Z \leftarrow X[0] \times Y[0]\)

\[
\begin{align*}
Z_1 & \leftarrow \text{NotSoFastMultiply}(X_L, Y_L, k/2) \\
Z_2 & \leftarrow \text{NotSoFastMultiply}(X_R, Y_R, k/2)
\end{align*}
\]

else
\[
\begin{align*}
Z_3 & \leftarrow \text{NotSoFastMultiply}(X_L, Y_R, k/2) \\
Z_4 & \leftarrow \text{NotSoFastMultiply}(X_R, Y_L, k/2)
\end{align*}
\]
\(Z \leftarrow \text{LeftShift}(Z_1, k) + Z_2 + \text{LeftShift}(Z_3 + Z_4, k/2)\)

return \((Z)\)
Fast D&C Multiprecision Multiplication

Algorithm: \( \text{FastMultiply}(X, Y, k) \)

if \( k = 1 \)

then \( Z \leftarrow X[0] \times Y[0] \)

\[
\begin{align*}
X_T & \leftarrow X_L + X_R \\
Y_T & \leftarrow Y_L + Y_R
\end{align*}
\]

else

\[
\begin{align*}
Z_1 & \leftarrow \text{FastMultiply}(X_L, Y_L, k/2) \\
Z_2 & \leftarrow \text{FastMultiply}(X_R, Y_R, k/2) \\
Z_3 & \leftarrow \text{FastMultiply}(X_T, Y_T, k/2),
\end{align*}
\]

\[
Z \leftarrow \text{LeftShift}(Z_1, k) + Z_2 + \text{LeftShift}(Z_3 - Z_1 - Z_2, k/2)
\]

return \( (Z) \)
Matrix Multiplication

Problem

Matrix Multiplication

Instance: Two \( n \) by \( n \) matrices, \( A \) and \( B \).

Question: Compute the \( n \) by \( n \) matrix product \( C = AB \).

The naive algorithm for Matrix Multiplication has complexity \( \Theta(n^3) \).
Matrix Multiplication: Problem Decomposition

\[ A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}, \quad C = AB = \begin{pmatrix} r & s \\ t & u \end{pmatrix} \]

If \( A, B \) are \( n \) by \( n \) matrices, then \( a, b, ..., h, r, s, t, u \) are \( \frac{n}{2} \) by \( \frac{n}{2} \) matrices, where

\[
\begin{align*}
  r &= ae + bg \\
  s &= af + bh \\
  t &= ce + dg \\
  u &= cf + dh
\end{align*}
\]

We require 8 multiplications of \( \frac{n}{2} \) by \( \frac{n}{2} \) matrices in order to compute \( C = AB \).
Efficient D&C Matrix Multiplication

Define

\[ P_1 = a(f - h) \]
\[ P_3 = (c + d)e \]
\[ P_5 = (a + d)(e + h) \]
\[ P_7 = (a - c)(e + f). \]

Then, compute

\[ r = P_5 + P_4 - P_2 + P_6 \]
\[ t = P_3 + P_4 \]
\[ s = P_1 + P_2 \]
\[ u = P_5 + P_1 - P_3 - P_7. \]

We now require only 7 multiplications of \( \frac{n}{2} \) by \( \frac{n}{2} \) matrices in order to compute \( C = AB \).
Selection

**Problem**

**Selection**

**Instance:** An array $A[1], \ldots, A[n]$ of distinct integer values, and an integer $k$, where $1 \leq k \leq n$.

**Find:** The $k$th smallest integer in the array $A$.

The problem **Median** is the special case of **Selection** where $k = \lceil \frac{n}{2} \rceil$. 

QuickSelect

Suppose we choose a pivot element $y$ in the array $A$, and we restructure $A$ so that all elements less than $y$ precede $y$ in $A$, and all elements greater than $y$ occur after $y$ in $A$. (This is exactly what is done in Quicksort, and it takes linear time.)


Then the $k$th smallest element of $A$ is

$$\begin{cases} 
  y & \text{if } k = posn \\
  \text{the } k\text{th smallest element of } A_L & \text{if } k < posn \\
  \text{the } (k - posn)\text{th smallest element of } A_R & \text{if } k > posn.
\end{cases}$$

We make (at most) one recursive call at each level of the recursion.
Average-case Analysis of QuickSelect

We say that a pivot is **good** if $posn$ is in the middle half of $A$.
The probability that a pivot is good is $1/2$.
On average, after **two iterations**, we will encounter a good pivot.
If a pivot is good, then $|A_L| \leq 3n/4$ and $|A_R| \leq 3n/4$.
With an **expected** linear amount of work, the size of the subproblem is reduced by at least 25%.
It follows that the average-case complexity of the **QuickSelect** is linear.
Achieving $O(n)$ Worst-Case Complexity: A Strategy for Choosing the Pivot

We choose the pivot to be a certain median-of-medians:

**step 1** Given $n \geq 15$, write $n = 10r + 5 + \theta$, where $r \geq 1$ and $0 \leq \theta \leq 9$.

**step 2** Divide $A$ into $2r + 1$ disjoint subarrays of 5 elements. Denote these subarrays by $B_1, \ldots, B_{2r+1}$.

**step 3** For $1 \leq i \leq 2r + 1$, find the median of $B_i$ (nonrecursively), and denote it by $m_i$.

**step 4** Define $M$ to be the array consisting of elements $m_1, \ldots, m_{2r+1}$.

**step 5** Find the median $y$ of the array $M$ (recursively).

**step 6** Use the element $y$ as the pivot for $A$. 
Median-of-medians-QuickSelect

Algorithm: $\text{Mom-QuickSelect}(k, n, A)$
1. if $n \leq 14$ then sort $A$ and return $(A[k])$
2. write $n = 10r + 5 + \theta$, where $0 \leq \theta \leq 9$
3. construct $B_1, \ldots, B_{2r+1}$ (subarrays of $A$, each of size 5)
4. find medians $m_1, \ldots, m_{2r+1}$ (non-recursively)
5. $M \leftarrow [m_1, \ldots, m_{2r+1}]$
6. $y \leftarrow \text{Mom-QuickSelect}(r + 1, 2r + 1, M)$
7. $(A_L, A_R, \text{posn}) \leftarrow \text{Restructure}(A, y)$
8. if $k = \text{posn}$ then return $(y)$
9. else if $k < \text{posn}$ then return $(\text{Mom-QuickSelect}(k, \text{posn} - 1, A_L))$
10. else return $(\text{Mom-QuickSelect}(k - \text{posn}, n - \text{posn}, A_R))$
Worst-case Analysis of Mom-QuickSelect

When the pivot is the median-of-medians, we have that $|A_L| \leq \left\lceil \frac{7n+12}{10} \right\rceil$ and $A_R \leq \left\lceil \frac{7n+12}{10} \right\rceil$.

The *Mom-QuickSelect* algorithm requires two recursive calls.

The worst-case complexity $T(n)$ of this algorithm satisfies the following recurrence:

$$T(n) \leq \begin{cases} T\left(\left\lfloor \frac{n}{5} \right\rfloor \right) + T\left(\left\lceil \frac{7n+12}{10} \right\rceil \right) + \Theta(n) & \text{if } n \geq 15 \\ \Theta(1) & \text{if } n \leq 14 \end{cases}$$

It can be shown that $T(n)$ is $O(n)$. 
Table of Contents

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

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

**local evaluation criterion**

For any \( y \in \mathcal{X} \), \( g(y) \) 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

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. Choose the **earliest starting** interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $s_i$).

2. 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. 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: $GreedyIntervalSelection(A)$

rename the intervals, by sorting if necessary, so that $f_1 \leq \cdots \leq f_n$

$B \leftarrow \{A_1\}$

$prev \leftarrow 1$

comment: $prev$ is the index of the last selected interval

for $i \leftarrow 2$ to $n$

\[
\begin{cases}
\text{if } s_i \geq f_{prev} \\
\text{then } B \leftarrow B \cup \{A_i\} \\
\phantom{\text{if }} prev \leftarrow i
\end{cases}
\]

return $(B)$
Problem

Interval Colouring

Instance: A set $\mathcal{A} = \{A_1, \ldots, A_n\}$ 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 $\text{col} : \mathcal{A} \to \{1, \ldots, c\}$ that assigns each interval a colour such that two intervals receiving the same colour are always disjoint.

Find: A $c$-colouring of $\mathcal{A}$ 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 the 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?
A Greedy Algorithm for Interval Colouring

Algorithm: \textit{GreedyIntervalColouring}(\mathcal{A})

rename the intervals, by sorting if necessary, so that \( s_1 \leq \cdots \leq s_n \)

\( d \leftarrow 1 \)
\( \text{colour}[1] \leftarrow 1 \)
\( \text{finish}[1] \leftarrow f_1 \)

\textbf{for} \( i \leftarrow 2 \) \textbf{to} \( n \)

\[
\begin{cases}
 \text{flag} \leftarrow \text{false} \\
 c \leftarrow 1 \\
 \textbf{while} \ c \leq d \text{ and } (\text{not flag}) \\
 \quad \begin{cases}
 \text{if} \ \text{finish}[c] \leq s_i \text{ then} \\
 \quad \begin{cases}
 \text{colour}[i] \leftarrow c \\
 \text{finish}[c] \leftarrow f_i \\
 \text{flag} \leftarrow \text{true} \\
 \end{cases} \\
 \quad \text{else} \ c \leftarrow c + 1 \\
 \end{cases} \\
 \text{if} \ \text{not flag} \text{ then} \\
 \quad \begin{cases}
 d \leftarrow d + 1 \\
 \text{colour}[i] \leftarrow d \\
 \text{finish}[d] \leftarrow f_i \\
 \end{cases}
\end{cases}
\]

\textbf{return} \( (d, \text{colour}) \)
Comments and Questions

In the algorithm on the previous slide, at any point in time, $\text{finish}[c]$ denotes the finishing time of the last interval that has received colour $c$. Therefore, a new interval $A_i$ can be assigned colour $c$ if $s_i \geq \text{finish}[c]$.

The complexity of the algorithm is $O(n \times D)$, where $D$ is the value of $d$ returned by the algorithm.

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?
Knapsack Problems

**Problem**

**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 $\sum_{i=1}^{n} p_i x_i$. 
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 0-1 Knapsack or Rational Knapsack problem?
A Greedy Algorithm for Rational Knapsack

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

rename the items, sorting if necessary, so that \(p_1/w_1 \geq \cdots \geq p_n/w_n\)

\(X \leftarrow [0, \ldots, 0]\)

\(i \leftarrow 1\)

\(CurW \leftarrow 0\)

**while** \((CurW < M) \text{ and } (i \leq n)\)

\[
\begin{align*}
\text{if } & \text{CurW} + w_i \leq M \\
\text{do} & \left\{ \\
\text{if} & \text{CurW} \leftarrow \text{CurW} + w_i \\
& i \leftarrow i + 1 \\
\text{else} & \left\{ \\
& x_i \leftarrow (M - \text{CurW})/w_i \\
& \text{CurW} := M \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\right. \\
\righth
Coin Changing

Problem

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 rename the coins, by sorting if necessary, so that \( d_1 > \cdots > d_n \)
    \item \( N \leftarrow 0 \)
    \item \textbf{for} \( i \leftarrow 1 \) \textbf{to} \( n \)
        \begin{align*}
            a_i & \leftarrow \left\lfloor \frac{T}{d_i} \right\rfloor \\
            T & \leftarrow T - a_id_i \\
            N & \leftarrow N + a_i
        \end{align*}
        \textbf{do}
    \item if \( T > 0 \)
        \begin{itemize}
            \item then \textbf{return} \((\text{fail})\)
            \item else \textbf{return} \(((a_1, \ldots, a_n), N)\)
        \end{itemize}
\end{itemize}
The Stable Marriage Problem

**Problem**

**Stable Marriage**

**Instance:** A set of \( n \) men, say \( M = [m_1, \ldots, m_n] \), and a set of \( n \) women, \( W = [w_1, \ldots, w_n] \).

Each man \( m_i \) has a **preference ranking** of the \( n \) women, and each woman \( w_i \) has a preference ranking of the \( n \) men: \( \text{pref}(m_i, j) = w_k \) if \( w_k \) is the \( j \)-th favourite woman of man \( m_i \); and \( \text{pref}(w_i, j) = m_k \) if \( m_k \) is the \( j \)-th favourite man of woman \( w_i \).

**Find:** A **matching** of the \( n \) men with the \( n \) women such that there does not exist a couple \((m_i, w_j)\) who are not engaged to each other, but prefer each other to their existing matches. A matching with this property is called a **stable matching**.
Overview of the Gale-Shapley Algorithm

Men propose to women.

If a woman accepts a proposal, then the couple is engaged.

An unmatched woman must accept a proposal.

If an engaged woman receives a proposal from a man whom she prefers to her current match, the she cancels her existing engagement and she becomes engaged to the new proposer; her previous match is no longer engaged.

If an engaged woman receives a proposal from a man, but she prefers her current match, then the proposal is rejected.

Engaged women never become unengaged.

A man might make a number of proposals (up to $n$); the order of the proposals is determined by the man’s preference list.
**Gale-Shapley Algorithm**

**Algorithm:** *Gale-Shapley*(M, W, pref)

\[ Match \leftarrow \emptyset \]

while there exists an unengaged man \( m_i \)

let \( w_j \) be the next woman in \( m_i \)'s preference list

if \( w_j \) is not engaged

then \( Match \leftarrow Match \cup \{m_i, w_j\} \)

else

suppose \( \{m_k, w_j\} \in Match \)

if \( w_j \) prefers \( m_i \) to \( m_k \)

then \( Match \leftarrow Match \setminus \{m_k, w_j\} \cup \{m_i, w_j\} \)

comment: \( m_k \) is now unengaged

return \( (Match) \)
Questions

How do we prove that the *Gale-Shapley* algorithm always *terminates*?

How many *iterations* does this algorithm require in the worst case?

How do we prove that this algorithm is *correct*, i.e., that it finds a stable matching?

Is there an efficient way to *identify* an unengaged man at any point in the algorithm? What *data structure* would be helpful in doing this?

What can we say about the *complexity* of the algorithm?
## Table of Contents

- Dynamic Programming Algorithms
  - Fibonacci Numbers
  - Design Strategy
  - 0-1 Knapsack
  - Coin Changing
  - Longest Common Subsequence
  - Minimum Length Triangulation
Computing Fibonacci Numbers Inefficiently

**Algorithm:** $BadFib(n)$

```
if $n = 0$ then $f \leftarrow 0$
else if $n = 1$ then $f \leftarrow 1$
else
    \[
    \begin{cases}
    f_1 \leftarrow BadFib(n - 1) \\
    f_2 \leftarrow BadFib(n - 2)
    \end{cases}
    \]
    \[
    f \leftarrow f_1 + f_2
    \]
return $(f)$;
```
Computing Fibonacci Numbers More Efficiently

Algorithm: \textit{BetterFib}(n)

\begin{verbatim}
    f[0] ← 0
    f[1] ← 1
    for i ← 2 to n
        do f[i] ← f[i − 1] + f[i − 2]
    return (f[n])
\end{verbatim}
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 $I$. 
0-1 Knapsack

**Problem**

**0-1 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 \( x_i \in \{0, 1\} \) for \( 1 \leq i \leq n \), and \( \sum_{i=1}^{n} w_ix_i \leq M \).

**Find:** A feasible solution \( X \) that maximizes \( \sum_{i=1}^{n} p_ix_i \).

Let \( P[i, m] \) denote the optimal solution to the subproblem consisting of the first \( i \) objects (having profits \( p_1, \ldots, p_i \) and weights \( w_1, \ldots, w_i \), respectively) and capacity \( m \).
A Dynamic Programming Algorithm for 0-1 Knapsack

Algorithm: 0-1Knapsack\((p_1, \ldots, p_n, w_1, \ldots, w_n, M)\)

for \(m \leftarrow 0\) to \(M\)

\[
\begin{align*}
\text{if } m &\geq w_1 \\
\text{do } &
\begin{cases}
\text{then } P[1, m] &\leftarrow p_1 \\
\text{else } P[1, m] &\leftarrow 0
\end{cases}
\end{align*}
\]

for \(i \leftarrow 2\) to \(n\)

\[
\begin{align*}
\text{for } m &\leftarrow 0\text{ to } M \\
\text{do } &
\begin{cases}
\text{if } m < w_i \\
\text{do } &
\begin{cases}
\text{then } P[i, m] &\leftarrow P[i - 1, m] \\
\text{else } P[i, m] &\leftarrow \max\{P[i - 1, m - w_i] + p_i, P[i - 1, m]\}
\end{cases}
\end{cases}
\end{align*}
\]

return \((P[n, M])\);
Computing the Optimal Knapsack $X$

Algorithm: $\text{0-1Knapsack}(p_1, \ldots, p_n, w_1, \ldots, w_n, M, P)$

1. $m \leftarrow M$
2. $p \leftarrow P[n, M]$
3. for $i \leftarrow n$ downto 2
   ```
   \begin{cases}
   \text{if } p = P[i - 1, m] \\
   \quad \text{then } x_i \leftarrow 0
   \end{cases}
   \begin{cases}
   \quad x_i \leftarrow 1
   \end{cases}
   \begin{cases}
   \text{else } \quad p \leftarrow p - p_i \\
   \quad m \leftarrow m - w_i
   \end{cases}
   ```
4. if $p = 0$
   ```
   \begin{cases}
   \text{then } x_1 \leftarrow 0
   \end{cases}
   ```
5. else $x_1 \leftarrow 1$
6. return $(X)$;
Coin Changing

Problem

Coin Changing

Instance: A list of coin denominations, $1 = d_1, d_2, \ldots, d_n$, 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 $T = \sum_{i=1}^{n} a_id_i$ and such that $N = \sum_{i=1}^{n} a_i$ is minimized.

Let $N[i, t]$ denote the optimal solution to the subproblem consisting of the first $i$ coin denominations $d_1, \ldots, d_i$ and target sum $t$. Let $A[i, t]$ denote the number of coins of denomination $d_i$ used in the optimal solution to this subproblem.
A Dynamic Programming Algorithm for Coin Changing

Algorithm: *Coin Changing* \((d_1, \ldots, d_n, T)\)

comment: \(d_1 = 1\)

for \(t \leftarrow 0\) to \(T\)

\[
\begin{align*}
N[1, t] & \leftarrow t \\
A[1, t] & \leftarrow t
\end{align*}
\]

for \(i \leftarrow 2\) to \(n\)

for \(t \leftarrow 0\) to \(T\)

\[
\begin{align*}
N[i, t] & \leftarrow N[i - 1, t] \\
A[i, t] & \leftarrow 0
\end{align*}
\]

for \(j \leftarrow 1\) to \(\lfloor (t/d_i) \rfloor\)

\[
\begin{align*}
\text{if } j + N[i - 1, t - jd_i] & < N[i, t] \\
\quad \text{do } \{ N[i, t] \leftarrow j + N[i - 1, t - jd_i] \} \\
\text{then } \{ A[i, t] \leftarrow j \}
\end{align*}
\]

return \((N[n, T])\);
**Problem**

**Longest Common Subsequence**

**Instance:** Two sequences \( X = (x_1, \ldots, x_m) \) and \( Y = (y_1, \ldots, y_n) \) over some finite alphabet \( \Gamma \).

**Find:** A maximum length sequence \( Z \) that is a subsequence of both \( X \) and \( Y \).

\( Z = (z_1, \ldots, z_\ell) \) is a subsequence of \( X \) if there exist indices \( 1 \leq i_1 < \cdots < i_\ell \leq m \) such that \( z_j = x_{i_j}, 1 \leq j \leq \ell \).

Similarly, \( Z \) is a subsequence of \( Y \) if there exist (possibly different) indices \( 1 \leq h_1 < \cdots < h_\ell \leq n \) such that \( z_j = y_{h_j}, 1 \leq j \leq \ell \).
Computing the Length of the LCS of $X$ and $Y$

**Algorithm:** \textit{LCS1}($X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n)$)

\begin{verbatim}
for $i \leftarrow 0$ to $m$
do $c[i, 0] \leftarrow 0$
for $j \leftarrow 0$ to $n$
do $c[0, j] \leftarrow 0$
for $i \leftarrow 1$ to $m$
    for $j \leftarrow 1$ to $n$
do        if $x_i = y_j$
do            then $c[i, j] \leftarrow c[i - 1, j - 1] + 1$
do            else $c[i, j] \leftarrow \max\{c[i, j - 1], c[i - 1, j]\}$
return $(c[m, n])$;
\end{verbatim}
Finding the LCS of $X$ and $Y$

Algorithm: $LCS2(X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n))$

for $i \leftarrow 0$ to $m$ do $c[i, 0] \leftarrow 0$
for $j \leftarrow 0$ to $n$ do $c[0, j] \leftarrow 0$

for $i \leftarrow 1$ to $m$
  for $j \leftarrow 1$ to $n$
    if $x_i = y_j$
      then
        $c[i, j] \leftarrow c[i - 1, j - 1] + 1$
        $\pi[i, j] \leftarrow \text{UL}$
    else if $c[i, j - 1] > c[i - 1, j]$
      then
        $c[i, j] \leftarrow c[i, j - 1]$
        $\pi[i, j] \leftarrow \text{L}$
    else
      $c[i, j] \leftarrow c[i - 1, j]$
      $\pi[i, j] \leftarrow \text{U}$

return $(c, \pi)$;
Finding the LCS

Algorithm: \textit{FindLCS}(c, \pi, v)

\begin{align*}
\text{seq} & \leftarrow () \\
i & \leftarrow m \\
j & \leftarrow n \\
\textbf{while} \quad \min\{i, j\} > 0 \\
& \begin{cases}
\text{if } \pi[i, j] = \text{UL} \\
\quad \text{then} \quad \begin{cases}
\text{seq} & \leftarrow x_i \parallel \text{seq} \\
i & \leftarrow i - 1 \\
j & \leftarrow j - 1 
\end{cases} \\
\text{else if } \pi[i, j] = \text{L} \quad \text{then } j & \leftarrow j - 1 \\
\text{else } i & \leftarrow i - 1
\end{cases}
\end{cases} \\
\text{return } (\text{seq})
\end{align*}
Minimum Length Triangulation

Problem

Minimum Length Triangulation v1
Instance: \( n \) points \( q_1, \ldots, q_n \) in the Euclidean plane that form a convex \( n \)-gon \( P \).
Find: A triangulation of \( P \) such that the sum \( S_c \) of the lengths of the \( n - 3 \) chords is minimized.

Problem

Minimum Length Triangulation v2
Instance: \( n \) points \( q_1, \ldots, q_n \) in the Euclidean plane that form a convex \( n \)-gon \( P \).
Find: A triangulation of \( P \) such that the sum \( S_p \) of the perimeters of the \( n - 2 \) triangles is minimized.

Let \( L \) denote the perimeter of \( P \). Then we have that \( S_p = L + 2S_c \).
Hence the two versions have the same optimal solutions.
Problem Decomposition

We consider version 2 of the problem.

The edge $q_nq_1$ is in a triangle with a third vertex $q_k$, where $k \in \{2, \ldots, n - 1\}$.

For a given $k$, we have:

1. the triangle $q_1q_kq_n$,
2. the polygon with vertices $q_1, \ldots, q_k$,
3. the polygon with vertices $q_k, \ldots, q_n$.

The optimal solution will consist of optimal solutions to the two subproblems in 2 and 3, along with the triangle in 1.
Recurrence Relation

For $1 \leq i < j \leq n$, let $S[i, j]$ denote the optimal solution to the subproblem consisting of the polygon having vertices $q_i, \ldots, q_j$.

Let $\Delta(q_i, q_k, q_j)$ denote the perimeter of the triangle having vertices $q_i, q_k, q_j$.

The we have the recurrence relation

$$S[i, j] = \min\{\Delta(q_i, q_k, q_j) + S[i, k] + S[k, j] : i < k < j\}.$$ 

The base cases are given by

$$S[i, i + 1] = 0$$

for all $i$.

We compute all $S[i, j]$ with $j - i = c$, for $c = 2, 3, \ldots, n - 1$. 
Table of Contents

6 Graph Algorithms
- Graphs
- Breadth-first Search
- Depth-first Search
- Topological Ordering
- Strongly Connected Components
- Minimum Spanning Trees
- Single Source Shortest Paths
- All-Pairs Shortest Paths
Graphs and Digraphs

A **graph** is a pair $G = (V, E)$. $V$ is a set whose elements are called **vertices** and $E$ is a set whose elements are called **edges**. Each edge joins two distinct vertices. An edge can be represented as a set of two vertices, e.g., $\{u, v\}$, where $u \neq v$. We may also write this edge as $uv$ or $vu$.

We often denote the number of vertices by $n$ and the number of edges by $m$. Clearly $m \leq \binom{n}{2}$.

A **directed graph** or **digraph** is also a pair $G = (V, E)$. The elements of $E$ are called **directed edges** or **arcs** in a digraph. Each arc joins two vertices, and an arc can be represented as a ordered pair, e.g., $(u, v)$. The arc $(u, v)$ is directed from $u$ (the **tail**) to $v$ (the **head**), and we allow $u = v$.

If we denote the number of vertices by $n$ and the number of arcs by $m$, then $m \leq n^2$. 
Data Structures for Graphs: Adjacency Matrices

There are two main data structures to represent graphs: an \textbf{adjacency matrix} and a set of \textbf{adjacency lists}.

Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$. The \textbf{adjacency matrix} of $G$ is an $n \times n$ matrix $A = (a_{u,v})$, which is indexed by $V$, such that

$$a_{u,v} = \begin{cases} 1 & \text{if } \{u, v\} \in E \\ 0 & \text{otherwise.} \end{cases}$$

There are exactly $2m$ entries of $A$ equal to 1.

If $G$ is a digraph, then

$$a_{u,v} = \begin{cases} 1 & \text{if } (u, v) \in E \\ 0 & \text{otherwise.} \end{cases}$$

For a digraph, there are exactly $m$ entries of $A$ equal to 1.
Data Structures for Graphs: Adjacency Lists

Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$.

An adjacency list representation of $G$ consists of $n$ linked lists. For every $u \in V$, there is a linked list (called an adjacency list) which is named $Adj[u]$.

For every $v \in V$ such that $uv \in E$, there is a node in $Adj[u]$ labelled $v$. (This definition is used for both directed and undirected graphs.)

In an undirected graph, every edge $uv$ corresponds to nodes in two adjacency lists: there is a node $v$ in $Adj[u]$ and a node $u$ in $Adj[v]$.

In a directed graph, every edge corresponds to a node in only one adjacency list.
Breadth-first Search of an Undirected Graph

A breadth-first search of an undirected graph begins at a specified vertex $s$.

The search “spreads out” from $s$, proceeding in layers.

First, all the neighbours of $s$ are explored.

Next, the neighbours of those neighbours are explored.

This process continues until all vertices have been explored.

A queue is used to keep track of the vertices to be explored.
Breadth-first Search

Algorithm: \(BFS(G, s)\)

for each \(v \in V(G)\) do 
  \(colour[v] \leftarrow \text{white}\)
  \(\pi[v] \leftarrow \emptyset\)

\(colour[s] \leftarrow \text{gray}\)

InitializeQueue\((Q)\)

Enqueue\((Q, s)\)

while \(Q \neq \emptyset\) do 
  \(u \leftarrow \text{Dequeue}(Q)\)
  for each \(v \in \text{Adj}[u]\) do 
    if \(colour[v] = \text{white}\) then 
      \(colour[v] \leftarrow \text{gray}\)
      \(\pi[v] \leftarrow u\)
      Enqueue\((Q, v)\)
    \(colour[u] \leftarrow \text{black}\)
Properties of Breadth-first Search

A vertex is **white** if it is **undiscovered**.

A vertex is **gray** if it has been **discovered**, but we are still processing its adjacent vertices.

A vertex becomes **black** when all the adjacent vertices have been processed.

If $G$ is **connected**, then every vertex eventually is coloured black and every vertex $v \neq s$ has a unique predecessor $\pi[v]$ in the BFS tree.

When we explore an edge $\{u, v\}$ starting from $u$:

- if $v$ is **white**, then $uv$ is a **tree edge** and $\pi[v] = u$ is the **predecessor** of $v$ in the **BFS tree**
- otherwise, $uv$ is a **cross edge**.

The BFS tree consists of all the tree edges.
Shortest Paths via Breadth-first Search

Algorithm: \( \text{BFS}(G, s) \)

\[
\text{for each } v \in V(G) \text{ do } \begin{cases} \text{colour}[v] \leftarrow \text{white} \\ \pi[v] \leftarrow \emptyset \end{cases}
\]

\[
\text{colour}[s] \leftarrow \text{gray} \\
\text{dist}[s] \leftarrow 0
\]

\( \text{InitializeQueue}(Q) \)

\( \text{Enqueue}(Q, s) \)

\( \text{while } Q \neq \emptyset \)

\[
\begin{cases} u \leftarrow \text{Dequeue}(Q) \\
\text{for each } v \in \text{Adj}[u] \text{ do } \begin{cases} \text{if colour}[v] = \text{white} \text{ then } \begin{cases} \text{colour}[v] = \text{gray} \\ \pi[v] \leftarrow u \\ \text{Enqueue}(Q, v) \\ \text{dist}[v] \leftarrow \text{dist}[u] + 1 \end{cases} \\ \text{colour}[u] \leftarrow \text{black} \end{cases} \end{cases}
\]
Distances in Breadth-first Search

**Lemma**

If $u$ is discovered before $v$, then $\text{dist}[u] \leq \text{dist}[v]$.

**Lemma**

If $\{u, v\}$ is any edge, then $|\text{dist}[u] - \text{dist}[v]| \leq 1$.

**Theorem**

$\text{dist}[v]$ is the length of the shortest path from $s$ to $v$. 
A graph is **bipartite** if the vertex set can be partitioned as \( V = X \cup Y \), in such a way that all edges have one endpoint in \( X \) and one endpoint in \( Y \).

A graph is bipartite if and only if it does not contain an **odd cycle**.

**BFS** can be used to test if a graph is bipartite:

- if we encounter an edge \( \{u, v\} \) with \( \text{dist}[u] = \text{dist}[v] \), then \( G \) is not bipartite, whereas

- if no such edge is found, then define \( X = \{u : \text{dist}[u] \text{ is even}\} \) and \( Y = \{u : \text{dist}[u] \text{ is odd}\} \); then \( X, Y \) forms a bipartition.
Depth-first Search of a Directed Graph

A depth-first search uses a stack (or recursion) instead of a queue. We define predecessors and colour vertices as in BFS. It is also useful to specify a discovery time $d[v]$ and a finishing time $f[v]$ for every vertex $v$. We increment a time counter every time a value $d[v]$ or $f[v]$ is assigned. We eventually visit all the vertices, and the algorithm constructs a depth-first forest.

The complexity of depth-first search is $\Theta(n + m)$. 
Depth-first Search

Algorithm: \( DFS(G) \)

\[
\text{for each } v \in V(G) \\
\text{do } \begin{cases} 
\text{colour}[v] \leftarrow \text{white} \\
\pi[v] \leftarrow \emptyset \\
\end{cases}
\]

\[
\text{time} \leftarrow 0
\]

\[
\text{for each } v \in V(G) \\
\text{do } \begin{cases} 
\text{if } \text{colour}[v] = \text{white} \\
\text{then } DFSvisit(v) \\
\end{cases}
\]
Depth-first Search (cont.)

Algorithm: $DFSvisit(v)$

$\text{colour}[v] \leftarrow \text{gray}$

$time \leftarrow time + 1$

$d[v] \leftarrow time$

\text{comment: } d[v] \text{ is the discovery time for vertex } v$

\text{for each } w \in Adj[v]$

\begin{align*}
\text{if } \text{colour}[w] &= \text{white} \\
\text{then } \{ & \pi[w] \leftarrow v \}
\end{align*}

$\text{colour}[v] \leftarrow \text{black}$

$time \leftarrow time + 1$

$f[v] \leftarrow time$

\text{comment: } f[v] \text{ is the finishing time for vertex } v$
Classification of Edges in Depth-first Search

- $uv$ is a **tree edge** if $u = \pi[v]$
- $uv$ is a **forward edge** if it is not a tree edge, and $v$ is a descendant of $u$ in a tree in the depth-first forest
- $uv$ is a **back edge** if $u$ is a descendant of $v$ in a tree in the depth-first forest
- any other edge is a **cross edge**.
Properties of Edges in Depth-first Search

In the following table, we indicate the colour of a vertex $v$ when an edge $uv$ is discovered, and the relation between the start and finishing times of $u$ and $v$, for each possible type of edge $uv$.

<table>
<thead>
<tr>
<th>edge type</th>
<th>colour of $v$</th>
<th>discovery/finish times</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>white</td>
<td>$d[u] &lt; d[v] &lt; f[v] &lt; f[u]$</td>
</tr>
<tr>
<td>forward</td>
<td>black</td>
<td>$d[u] &lt; d[v] &lt; f[v] &lt; f[u]$</td>
</tr>
<tr>
<td>back</td>
<td>gray</td>
<td>$d[v] &lt; d[u] &lt; f[u] &lt; f[v]$</td>
</tr>
<tr>
<td>cross</td>
<td>black</td>
<td>$d[v] &lt; f[v] &lt; d[u] &lt; f[u]$</td>
</tr>
</tbody>
</table>

Observe that two intervals $(d[u], f[u])$ and $(d[v], f[v])$ never overlap. Two intervals are either disjoint or nested. This is sometimes called the parenthesis theorem.
Topological Orderings and DAGs

A directed graph $G$ is a **directed acyclic graph**, or **DAG**, if $G$ contains no directed cycle.

A directed graph $G = (V, E)$ has a **topological ordering**, or **topological sort**, if there is a linear ordering $<$ of all the vertices in $V$ such that $u < v$ whenever $uv \in E$.

Some interesting/useful facts:

- A DAG contains a vertex of indegree 0.
- A directed graph $G$ has a topological ordering if and only if it is a DAG.
- A directed graph $G$ is a DAG if and only if a DFS of $G$ has no back edges.
- If $uv$ is an edge in a DAG, then a DFS of $G$ has $f[v] < f[u]$. 

Topological Ordering via Depth-first Search

Algorithm: \textit{DFS}(G)

\begin{algorithmic}
\State \textit{InitializeStack}(S)
\State \text{DAG}$ \leftarrow \text{true}$
\For {each} \(v \in V(G)\)
\Do
\State \text{colour}[v] \leftarrow \text{white}
\State \pi[v] \leftarrow \emptyset
\EndDo
\State \text{time} \leftarrow 0$
\EndFor
\For {each} \(v \in V(G)\)
\Do
\If {\text{colour}[v] = \text{white}}
\Then \text{DFSvisit}(v)$
\EndIf
\EndDo
\EndFor
\If {\text{DAG}}
\Return (S)$
\Else 
\Return (DAG)$
\EndIf
\end{algorithmic}
### Topological Ordering via Depth-first Search (cont.)

**Algorithm:** `DFSvisit(v)`

- `colour[v] ← gray`
- `time ← time + 1`
- `d[v] ← time`

**comment:** `d[v]` is the discovery time for vertex `v`

**for each** `w ∈ Adj[v]`

  **do**

  ```
  if `colour[w] = white`
  then
    ```
    π[w] ← v
    DFSvisit(w)
  ```

  **if** `colour[w] = gray` **then** `DAG ← false`

- `colour[v] ← black`

  **Push(S, v)**

- `time ← time + 1`
- `f[v] ← time`

**comment:** `f[v]` is the finishing time for vertex `v`
Strongly Connected Components of a Digraph $G$

For two vertices $x$ and $y$ of $G$, define $x \sim y$ if $x = y$; or if $x \neq y$ and there exist directed paths from $x$ to $y$ and from $y$ to $x$.

The relation $\sim$ is an equivalence relation.

The strongly connected components of $G$ are the equivalence classes of vertices defined by the relation $\sim$.

The component graph of $G$ is a directed graph whose vertices are the strongly connected components of $G$. There is an arc from $C_i$ to $C_j$ if and only if there is an arc in $G$ from some vertex of $C_i$ to some vertex of $C_j$.

For a strongly connected component $C$, define $f[C] = \max \{ f[v] : v \in C \}$ and $d[C] = \min \{ d[v] : v \in C \}$.

Some interesting/useful facts:

- The component graph of $G$ is a DAG.
- If $C_i, C_j$ are strongly connected components, and there is an arc from $C_i$ to $C_j$ in the component graph, then $f[C_i] > f[C_j]$. 
An Algorithm to Find the Strongly Connected Components

step 1 Perform a depth-first search of $G$, recording the finishing times $f[v]$ for all vertices $v$.

step 2 Construct a directed graph $H$ from $G$ by reversing the direction of all edges in $G$.

step 3 Perform a depth-first search of $H$, considering the vertices in decreasing order of the values $f[v]$ computed in step 1.

step 4 The strongly connected components of $G$ are the trees in the depth-first forest constructed in step 3.
Depth-first Search of $H$

Assume that $f[v_{i_1}] > f[v_{i_2}] > \cdots > f[v_{i_n}]$.

Algorithm: $DFS(H)$

for $j \leftarrow 1$ to $n$
    do $\text{colour}[v_{ij}] \leftarrow \text{white}$

$scc \leftarrow 0$

for $j \leftarrow 1$ to $n$
    if $\text{colour}[v_{ij}] = \text{white}$
        do
            then $scc \leftarrow scc + 1$
            $\text{DFSvisit}(H, v_{ij}, scc)$

return $(\text{comp})$

comment: $\text{comp}[v]$ is the strongly connected component containing $v$
DFSvisit for $H$

Algorithm: $DFSvisit(H, v, scc)$

1. $colour[v] \leftarrow \text{gray}$
2. $comp[v] \leftarrow scc$
3. for each $w \in Adj[v]$
   - if $colour[w] = \text{white}$
     - then $DFSvisit(H, w, scc)$
   - $colour[v] \leftarrow \text{black}$
Minimum Spanning Trees

A spanning tree in a connected, undirected graph $G = (V, E)$ is a subgraph $T$ that is a tree which contains every vertex of $V$.

$T$ is a spanning tree of $G$ if and only if $T$ is an acyclic subgraph of $G$ that has $n - 1$ edges (where $n = |V|$).

Problem

Minimum Spanning Tree

Instance: A connected, undirected graph $G = (V, E)$ and a weight function $w : E \to \mathbb{R}$.

Find: A spanning tree $T$ of $G$ such that

$$\sum_{e \in T} w(e)$$

is minimized (this is called a minimum spanning tree, or MST).
Kruskal’s Algorithm

Assume that \( w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m) \), where \( m = |E| \).

**Algorithm:** \textit{Kruskal}(G, w)

\[
A \leftarrow \emptyset \\
\text{for } j \leftarrow 1 \text{ to } m \\
\quad \text{do } \begin{cases} 
\quad \text{if } A \cup \{e_j\} \text{ does not contain a cycle} \\
\quad \quad \text{then } A \leftarrow A \cup \{e_j\}
\end{cases}
\]

return \( (A) \)
**Prim’s Algorithm (idea)**

We initially choose an arbitrary vertex $u_0$ and define $A = \{e\}$, where $e$ is the minimum weight edge incident with $u_0$.

$A$ is always a single tree, and at each step we select the minimum weight edge that joins a vertex in $V_A$ to a vertex not in $V_A$.

**Remark:** $V_A$ denotes the set of vertices in the tree $A$.

For a vertex $v \not\in V_A$, define

\[
N[v] = u, \text{ where } \{u, v\} \text{ is a minimum weight edge such that } u \in V_A \\
W[v] = w(N[v], v).
\]

Assume $w(u, v) = \infty$ if $\{u, v\} \not\in E$. 
Prim’s Algorithm

Algorithm: \textit{Prim}(G, w)

\begin{align*}
A & \leftarrow \emptyset \\
V_A & \leftarrow \{u_0\}, \text{ where } u_0 \text{ is arbitrary} \\
\text{for all } & v \in V \backslash \{u_0\} \\
\text{do} & \begin{cases} 
W[v] & \leftarrow w(u_0, v) \\
N[v] & \leftarrow u_0 
\end{cases} \\
\text{while } & |A| < n - 1 \\
\text{do} & \begin{cases} 
\text{choose } v \in V \backslash V_A \text{ such that } W[v] \text{ is minimized} \\
V_A & \leftarrow V_A \cup \{v\} \\
u & \leftarrow N[v] \\
A & \leftarrow A \cup \{uv\} \\
\text{for all } & v' \in V \backslash V_A \\
\text{do} & \begin{cases} 
\text{if } w(v, v') < W[v'] \\
\text{do} & \begin{cases} 
W[v'] & \leftarrow w(v, v') \\
N[v'] & \leftarrow v 
\end{cases} \\
\text{then} & \end{cases} 
\end{cases} \\
\text{return } & (A)
\end{align*}
Definitions

Let $G = (V, E)$ be a graph. A **cut** is a partition of $V$ into two non-empty (disjoint) sets, i.e., a pair $(S, V \setminus S)$, where $S \subseteq V$ and $1 \leq |S| \leq n - 1$.

Let $(S, V \setminus S)$ be a cut in a graph $G = (V, E)$. An edge $e \in E$ is a **crossing edge** with respect to the cut $(S, V \setminus S)$ if $e$ has one endpoint in $S$ and one endpoint in $V \setminus S$.

Let $A \subseteq E$. A cut $(S, V \setminus S)$ **respects** the set of edges $A$ provided that no edge in $A$ is a crossing edge.
A General Greedy Algorithm to Find an MST

Algorithm: \textit{GreedyMST}(G, w)

\begin{align*}
A & \leftarrow \emptyset \\
\text{while} \quad |A| < n - 1 & \\
\quad & \left\{ \\
\quad & \text{let } (S, V \setminus S) \text{ be a cut that respects } A \\
\quad & \text{do } \left\{ \\
\quad & \quad \text{let } e \text{ be a minimum weight crossing edge} \\
\quad & \quad A \leftarrow A \cup \{e\} \\
\quad & \right\} \\
\quad & \right\}
\end{align*}

return \( (A) \)
Single Source Shortest Paths

**Problem**

**Single Source Shortest Paths**

**Instance:** A directed graph \( G = (V, E) \), a non-negative weight function \( w : E \rightarrow \mathbb{R}^+ \cup \{0\} \), and a source vertex \( u_0 \in V \).

**Find:** For every vertex \( v \in V \), a directed path \( P \) from \( u_0 \) to \( v \) such that

\[
w(P) = \sum_{e \in P} w(e)
\]

is minimized.

The term *shortest path* really means *minimum weight path*.

We are asked to find \( n \) different shortest paths, one for each vertex \( v \in V \).

If all edges have weight 1, we can just use *BFS* to solve this problem.
Dijkstra’s Algorithm (Main Ideas)

$S$ is a subset of vertices such that the shortest paths from $u_0$ to all vertices in $S$ are known; initially, $S = \{u_0\}$.

For all vertices $v \in S$, $D[v]$ is the weight of the shortest path $P_v$ from $u_0$ to $v$, and all vertices on $P_v$ are in the set $S$.

For all vertices $v \notin S$, $D[v]$ is the weight of the shortest path $P_v$ from $u_0$ to $v$ in which all interior vertices are in $S$.

For $v \neq u_0$, $\pi[v]$ is the predecessor of $v$ on the path $P_v$.

At each stage of the algorithm, we choose $v \in V \setminus S$ so that $D[v]$ is minimized, and then we add $v$ to $S$.

Then the arrays $D$ and $\pi$ are updated appropriately.
Dijkstra’s Algorithm

Algorithm: \textit{Dijkstra}(G, w, u_0)

\begin{align*}
S & \leftarrow \{u_0\} \\
D[u_0] & \leftarrow 0 \\
\text{for all } v \in V \setminus \{u_0\} & \text{ do } \\
& \quad \begin{cases} 
D[v] & \leftarrow w(u_0, v) \\
\pi[v] & \leftarrow u_0 
\end{cases} \\
\text{while } |S| < n & \text{ do } \\
& \quad \begin{cases} 
\text{choose } v \in V \setminus S \text{ such that } D[v] \text{ is minimized} \\
S & \leftarrow S \cup \{v\} \\
\text{for all } v' \in V \setminus S & \text{ do } \\
& \quad \begin{cases} 
\text{if } D[v] + w(v, v') < D[v'] & \text{ do } \\
& \quad \begin{cases} 
D[v'] & \leftarrow D[v] + w(v, v') \\
\pi[v'] & \leftarrow v
\end{cases} \end{cases}
\end{cases}
\end{align*}

return \((D, \pi)\)
Finding the Shortest Paths

**Algorithm: **$\text{FindPath}(u_0, \pi, v)$

$\text{path} \leftarrow v$
$u \leftarrow v$

while $u \neq u_0$

\[
\begin{align*}
\text{do} & \quad \begin{cases} u \leftarrow \pi[u] \\ \text{path} \leftarrow u \parallel \text{path} \end{cases} \\
\text{return} & \quad \text{(path)}
\end{align*}
\]
Shortest Paths in a DAG

If $G$ is a DAG, we perform a topological ordering of the vertices. Suppose the resulting ordering is $v_1, \ldots, v_n$. Then we find all the shortest paths in $G$ with source $v_1$.

Note: This algorithm is correct even if there are negative-weight edges.

Algorithm: **DAG Shortest paths** $(G, w, v_1)$

```plaintext
for $j \leftarrow 1$ to $n$
  do $\left\{ D[v_1] \leftarrow \infty$
       $\pi[v_j] \leftarrow \text{undefined}$
     \$D[v_1] \leftarrow 0$

for $j \leftarrow 1$ to $n - 1$
  for all $v' \in \text{Adj}[v_j]$
    do $\left\{ \begin{array}{l}
           \text{if } D[v_j] + w(v_j, v') < D[v'] \\
           \text{then } \left\{ D[v'] \leftarrow D[v_j] + w(v_j, v') \\
                       \pi[v'] \leftarrow v_j \end{array} \right. \right.$

return $(D, \pi)$
```
All-Pairs Shortest Paths

**Problem**

**All-Pairs Shortest Paths**

**Instance:** A directed graph $G = (V, E)$, and a **weight matrix** $W$, where $W[i, j]$ denotes the weight of edge $ij$, for all $i, j \in V$, $i \neq j$.

**Find:** For all pairs of vertices $u, v \in V$, $u \neq v$, a directed path $P$ from $u$ to $v$ such that

$$w(P) = \sum_{ij \in P} W[i, j]$$

is minimized.

We allow edges to have negative weights, but we assume there are no negative-weight directed cycles in $G$. 
First Solution

Algorithm: \textit{SlowAllPairsShortestPath}(W)

\begin{align*}
L_1 &\leftarrow W \\
\text{for } m &\leftarrow 2 \text{ to } n - 1 \\
\quad \text{for } i &\leftarrow 1 \text{ to } n \\
\quad \quad \text{for } j &\leftarrow 1 \text{ to } n \\
\quad \quad \quad \text{for } k &\leftarrow 1 \text{ to } n \\
\quad \quad \quad \quad \ell &\leftarrow \infty \\
\quad \quad \quad \quad \text{do } \ell \leftarrow \min\{\ell, L_{m-1}[i, k] + W[k, j]\} \\
\quad \quad \quad \quad L_m[i, j] &\leftarrow \ell
\end{align*}

return \((L_{n-1})\)
Second Solution

Algorithm: \textit{FasterAllPairsShortestPath}(W)

\begin{itemize}
    \item $L_1 \leftarrow W$
    \item $m \leftarrow 1$
    \item \textbf{while} $m < n - 1$
    \item \hspace{1em} \textbf{for} $i \leftarrow 1$ \textbf{to} $n$
    \item \hspace{2em} \textbf{for} $j \leftarrow 1$ \textbf{to} $n$
    \item \hspace{3em} \textbf{for} $k \leftarrow 1$ \textbf{to} $n$
    \item \hspace{4em} $\ell \leftarrow \infty$
    \item \hspace{4em} \textbf{for} $k \leftarrow 1$ \textbf{to} $n$
    \item \hspace{5em} $\ell \leftarrow \min\{\ell, L_m[i, k] + L_m[k, j]\}$
    \item \hspace{5em} $L_{2m}[i, j] \leftarrow \ell$
    \item \hspace{1em} $m \leftarrow 2m$
    \item \textbf{return} $(L_m)$
\end{itemize}
Third Solution

Algorithm: *FloydWarshall*\( (W) \)

\[
D_0 \leftarrow W
\]

\[
\text{for } m \leftarrow 1 \text{ to } n
\]

\[
\text{for } i \leftarrow 1 \text{ to } n
\]

\[
\text{do } \begin{cases} 
\text{for } j \leftarrow 1 \text{ to } n \\
\text{do } \{ 
D_m[i, j] \leftarrow \min\{D_{m-1}[i, j], D_{m-1}[i, m] + D_{m-1}[m, j]\} 
\}
\end{cases}
\]

return \((D_n)\)
Table of Contents

7 Intractability and Undecidability

- Decision Problems
- The Complexity Class P
- Decision, Optimal Value and Optimization Problems
- The Complexity Class NP
- Reductions
- NP-completeness and NP-complete Problems
- Undecidability
Decision Problems

**Decision Problem:** Given a problem instance $I$, answer a certain question “yes” or “no”.

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

**Problem Solution:** Correct answer (“yes” or “no”) for the specified problem instance. $I$ is a **yes-instance** if the correct answer for the instance $I$ is “yes”. $I$ is a **no-instance** if the correct answer for the instance $I$ is “no”.

**Size of a problem instance:** $\text{Size}(I)$ is the number of bits required to specify (or encode) the instance $I$. 
Algorithm Solving a Decision Problem: An algorithm $A$ is said to solve a decision problem $\Pi$ provided that $A$ finds the correct answer (“yes” or “no”) for every instance $I$ of $\Pi$ in finite time.

Polynomial-time Algorithm: An algorithm $A$ for a decision problem $\Pi$ is said to be a polynomial-time algorithm provided that the complexity of $A$ is $O(n^k)$, where $k$ is a positive integer and $n = \text{Size}(I)$.

The Complexity Class $P$ denotes the set of all decision problems that have polynomial-time algorithms solving them. We write $\Pi \in P$ if the decision problem $\Pi$ is in the complexity class $P$. 
### Cycles in Graphs

#### Problem

**Cycle**

**Instance:** An undirected graph $G = (V, E)$.

**Question:** Does $G$ contain a cycle?

#### Problem

**Hamiltonian Cycle**

**Instance:** An undirected graph $G = (V, E)$.

**Question:** Does $G$ contain a hamiltonian cycle?

A **hamiltonian cycle** is a cycle that passes through every vertex in $V$ exactly once.
Knapsack Problems

**Problem**

**0-1 Knapsack-Dec**

**Instance:** a list of profits, \( P = [p_1, \ldots, p_n] \); a list of weights, \( W = [w_1, \ldots, w_n] \); a capacity, \( M \); and a target profit, \( T \).

**Question:** Is there an \( n \)-tuple \( [x_1, x_2, \ldots, x_n] \in \{0, 1\}^n \) such that \( \sum w_i x_i \leq M \) and \( \sum p_i x_i \geq T \)?

**Problem**

**Rational Knapsack-Dec**

**Instance:** a list of profits, \( P = [p_1, \ldots, p_n] \); a list of weights, \( W = [w_1, \ldots, w_n] \); a capacity, \( M \); and a target profit, \( T \).

**Question:** Is there an \( n \)-tuple \( [x_1, x_2, \ldots, x_n] \in [0, 1]^n \) such that \( \sum w_i x_i \leq M \) and \( \sum p_i x_i \geq T \)?
Polynomial-time Turing Reductions

Suppose $\Pi_1$ and $\Pi_2$ are problems (not necessarily decision problems). A (hypothetical) algorithm $A_2$ to solve $\Pi_2$ is called an oracle for $\Pi_2$.

Suppose that $A$ is an algorithm that solves $\Pi_1$, assuming the existence of an oracle $A_2$ for $\Pi_2$. ($A_2$ is used as a subroutine within the algorithm $A$.) Then we say that $A$ is a Turing reduction from $\Pi_1$ to $\Pi_2$, denoted $\Pi_1 \leq_T \Pi_2$.

A Turing reduction $A$ is a polynomial-time Turing reduction if the running time of $A$ is polynomial, under the assumption that the oracle $A_2$ has unit cost running time.

If there is a polynomial-time Turing reduction from $\Pi_1$ to $\Pi_2$, we write $\Pi_1 \leq_{TP} \Pi_2$.

Informally: Existence of a polynomial-time Turing reduction means that if we can solve $\Pi_2$ in polynomial time, then we can solve $\Pi_1$ in polynomial time.
# Travelling Salesperson Problems

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TSP-Optimization</strong></td>
</tr>
</tbody>
</table>
| **Instance:**              | A graph $G$ and edge weights $w : E \to \mathbb{Z}^+$.  
| **Find:**                  | A hamiltonian cycle $H$ in $G$ such that $w(H) = \sum_{e \in H} w(e)$ is minimized.  
|  
| **TSP-Optimal Value**       |  
| **Instance:**              | A graph $G$ and edge weights $w : E \to \mathbb{Z}^+$.  
| **Find:**                  | The minimum $T$ such that there exists a hamiltonian cycle $H$ in $G$ with $w(H) = T$.  
|  
| **TSP-Decision**            |  
| **Instance:**              | A graph $G$, edge weights $w : E \to \mathbb{Z}^+$, and a target $T$.  
| **Question:**              | Does there exist a hamiltonian cycle $H$ in $G$ with $w(H) \leq T$?  

**TSP-Optimal Value** \( \leq_{\text{TP}} \) **TSP-Dec**

**Algorithm:** *TSP-OptimalValue-Solver* \((G, w)\)

- **external** *TSP-Dec-Solver*
- \( hi \leftarrow \sum_{e \in E} w(e) \)
- \( lo \leftarrow 0 \)
- **if** not *TSP-Dec-Solver* \((G, w, hi)\) **then** return \((\infty)\)
- **while** \( hi > lo \)
  - \( mid \leftarrow \left\lfloor \frac{hi + lo}{2} \right\rfloor \)
  - **do**
    - **if** *TSP-Dec-Solver* \((G, w, mid)\)
      - **then** \( hi \leftarrow mid \)
    - **else** \( lo \leftarrow mid + 1 \)
  - **return** \((hi)\)
TSP-Optimization $\leq_{TP}^{T} \text{TSP-Dec}$

**Algorithm:** \( \text{TSP-Optimization-Solver}(G = (V, E), w) \)

*external* \( \text{TSP-OptimalValue-Solver}, \text{TSP-Dec-Solver} \)

\( T^* \leftarrow \text{TSP-OptimalValue-Solver}(G, w) \)

if \( T^* = \infty \) then return (“no hamiltonian cycle exists”)

\( w_0 \leftarrow w \)

\( H \leftarrow \emptyset \)

for all \( e \in E \)

\[
\begin{cases}
    w_0[e] \leftarrow \infty \\
    \text{if not } \text{TSP-Dec-Solver}(G, w_0, T^*) \\
    \text{then } \left\{ \\
        w_0[e] \leftarrow w[e] \\
        H \leftarrow H \cup \{e\}
    \right\}
\end{cases}
\]

return \( (H) \)
Certificates

Certificate: Informally, a certificate for a yes-instance $I$ is some “extra information” $C$ which makes it easy to verify that $I$ is a yes-instance.

Certificate Verification Algorithm: Suppose that $Ver$ is an algorithm that verifies certificates for yes-instances. Then $Ver(I, C)$ outputs “yes” if $I$ is a yes-instance and $C$ is a valid certificate for $I$. If $Ver(I, C)$ outputs “no”, then either $I$ is a no-instance, or $I$ is a yes-instance and $C$ is an invalid certificate.

Polynomial-time Certificate Verification Algorithm: A certificate verification algorithm $Ver$ is a polynomial-time certificate verification algorithm if the complexity of $Ver$ is $O(n^k)$, where $k$ is a positive integer and $n = \text{Size}(I)$. 
The Complexity Class NP

Certificate Verification Algorithm: A certificate verification algorithm $Ver$ is said to solve a decision problem $\Pi$ provided that

- for every yes-instance $I$, there exists a certificate $C$ such that $Ver(I, C)$ outputs “yes”.
- for every no-instance $I$ and for every certificate $C$, $Ver(I, C)$ outputs “no”.

The Complexity Class $NP$ denotes the set of all decision problems that have polynomial-time certificate verification algorithms solving them. We write $\Pi \in NP$ if the decision problem $\Pi$ is in the complexity class $NP$.

Finding Certificates vs Verifying Certificates: It is not required to be able to find a certificate $C$ for a yes-instance in polynomial time in order to say that a decision problem $\Pi \in NP$.

Important Fact: $P \subseteq NP$. 
Certificate Verification Algorithm for Hamiltonian Cycle

A certificate consists of an $n$-tuple, $X = [x_1, \ldots, x_n]$, that might be a hamiltonian cycle for a given graph $G = (V, E)$ (where $n = |V|$).

**Algorithm: Hamiltonian Cycle Certificate Verification** $(G, X)$

1. $flag \leftarrow true$
2. $Used \leftarrow \{x_1\}$
3. $j \leftarrow 2$
4. while $(j \leq n)$ and $flag$
   - do
     - $flag \leftarrow (x_j \not\in Used)$ and $(\{x_{j-1}, x_j\} \in E)$
     - if $(j = n)$ then $flag \leftarrow flag$ and $(\{x_n, x_1\} \in E)$
     - $Used \leftarrow Used \cup \{x_j\}$
     - $j \leftarrow j + 1$
   - return $(flag)$
Polynomial-time Reductions

For a decision problem $\Pi$, let $\mathcal{I}(\Pi)$ denote the set of all instances of $\Pi$. Let $\mathcal{I}_{\text{yes}}(\Pi)$ and $\mathcal{I}_{\text{no}}(\Pi)$ denote the set of all yes-instances and no-instances (respectively) of $\Pi$.

Suppose that $\Pi_1$ and $\Pi_2$ are decision problems. We say that there is a polynomial-time reduction (AKA polynomial transformation) from $\Pi_1$ to $\Pi_2$ (denoted $\Pi_1 \leq_P \Pi_2$) if there exists a function $f : \mathcal{I}(\Pi_1) \rightarrow \mathcal{I}(\Pi_2)$ such that the following properties are satisfied:

- $f(I)$ is computable in polynomial time (as a function of $\text{size}(I)$, where $I \in \mathcal{I}(\Pi_1)$)
- if $I \in \mathcal{I}_{\text{yes}}(\Pi_1)$, then $f(I) \in \mathcal{I}_{\text{yes}}(\Pi_2)$
- if $I \in \mathcal{I}_{\text{no}}(\Pi_1)$, then $f(I) \in \mathcal{I}_{\text{no}}(\Pi_2)$
# Two Graph Theory Decision Problems

## Problem

### Clique

**Instance:** An undirected graph $G = (V, E)$ and an integer $k$, where $1 \leq k \leq |V|$.

**Question:** Does $G$ contain a clique of size $\geq k$? (A **clique** is a subset of vertices $W \subseteq V$ such that $uv \in E$ for all $u, v \in W$, $u \neq v$.)

## Problem

### Vertex Cover

**Instance:** An undirected graph $G = (V, E)$ and an integer $k$, where $1 \leq k \leq |V|$.

**Question:** Does $G$ contain a vertex cover of size $\leq k$? (A **vertex cover** is a subset of vertices $W \subseteq V$ such that $\{u, v\} \cap W \neq \emptyset$ for all edges $uv \in E$.)
Clique \leq_P Vertex-Cover

Suppose that \( I = (G, k) \) is an instance of Clique, where \( G = (V, E) \), \( V = \{v_1, \ldots, v_n\} \) and \( 1 \leq k \leq n \).

Construct an instance \( f(I) = (H, \ell) \) of Vertex Cover, where \( H = (V, F) \), \( \ell = n - k \) and

\[
v_i v_j \in F \iff v_i v_j \notin E.
\]

\( H \) is called the complement of \( G \), because every edge of \( G \) is a non-edge of \( H \) and every non-edge of \( G \) is an edge of \( H \).
Properties of Polynomial-time Reductions

Suppose that $\Pi_1, \Pi_2, \ldots$ are decision problems.

**Theorem**

If $\Pi_1 \leq_P \Pi_2$ and $\Pi_2 \in P$, then $\Pi_1 \in P$.

**Theorem**

$\Pi_1 \leq_P \Pi_2$ and $\Pi_2 \leq_P \Pi_3$, then $\Pi_1 \leq_P \Pi_3$. 
The Complexity Class \textbf{NPC}

The complexity class \textit{NPC} denotes the set of all decision problems \( \Pi \) that satisfy the following two properties:

- \( \Pi \in \text{NP} \)
- For all \( \Pi' \in \text{NP} \), \( \Pi' \leq_P \Pi \).

\textit{NPC} is an abbreviation for \textbf{NP-complete}.

\textbf{Theorem}

\textit{If} \( \mathcal{P} \cap \text{NPC} \neq \emptyset \), \textit{then} \( \mathcal{P} = \text{NP} \).
Satisfiability and the Cook-Levin Theorem

Problem

**CNF-Satisfiability**

**Instance:** A boolean formula $F$ in $n$ boolean variables $x_1, \ldots, x_n$, such that $F$ is the conjunction (logical “and”) of $m$ clauses, where each clause is the disjunction (logical “or”) of literals. (A literal is a boolean variable or its negation.)

**Question:** Is there a truth assignment such that $F$ evaluates to true?

Theorem

**CNF-Satisfiability** $\in NPC$. 
Proving Problems NP-complete

Now, given any NP-complete problem, say $\Pi_1$, other problems in $NP$ can be proven to be NP-complete via polynomial reductions from $\Pi_1$, as stated in the following theorem:

**Theorem**

*Suppose that the following conditions are satisfied:*

- $\Pi_1 \in NPC$,
- $\Pi_1 \leq_P \Pi_2$, and
- $\Pi_2 \in NP$.

*Then $\Pi_2 \in NPC$.***
More Satisfiability Problems

**Problem**

**3-CNF-Satisfiability**

**Instance:** A boolean formula $F$ in $n$ boolean variables, such that $F$ is the conjunction of $m$ clauses, where each clause is the disjunction of exactly three literals.

**Question:** Is there a truth assignment such that $F$ evaluates to true?

**Problem**

**2-CNF-Satisfiability**

**Instance:** A boolean formula $F$ in $n$ boolean variables, such that $F$ is the conjunction of $m$ clauses, where each clause is the disjunction of exactly two literals.

**Question:** Is there a truth assignment such that $F$ evaluates to true?

$3$-CNF-Satisfiability $\in NPC$, while $2$-CNF-Satisfiability $\in P$. 
CNF-Satisfiability $\leq_P 3$-CNF-Satisfiability

Suppose that $(X, C)$ is an instance of CNF-SAT, where $X = \{x_1, \ldots, x_n\}$ and $C = \{C_1, \ldots, C_m\}$. For each $C_j$, do the following:

**case 1** If $|C_j| = 1$, say $C_j = \{z\}$, construct four clauses

$$\{z, a, b\}, \{z, a, \overline{b}\}, \{z, \overline{a}, b\}, \{z, \overline{a}, \overline{b}\}.$$

**case 2** If $|C_j| = 2$, say $C_j = \{z_1, z_2\}$, construct two clauses

$$\{z_1, z_2, c\}, \{z_1, z_2, \overline{c}\}.$$

**case 3** If $|C_j| = 3$, then leave $C_j$ unchanged.

**case 4** If $|C_j| \geq 4$, say $C_j = \{z_1, z_2, \ldots, z_k\}$, then construct $k - 2$ new clauses

$$\{z_1, z_2, d_1\}, \{\overline{d_1}, z_3, d_2\}, \{\overline{d_2}, z_4, d_3\}, \ldots,$$

$$\{\overline{d_{k-4}}, z_{k-2}, d_{k-3}\}, \{\overline{d_{k-3}}, z_{k-1}, z_k\}.$$
3-CNF-Satisfiability \( \leq_P \) Clique

Let \( I \) be the instance of 3-CNF-SAT consisting of \( n \) variables, \( x_1, \ldots, x_n \), and \( m \) clauses, \( C_1, \ldots, C_m \). Let \( C_i = \{ z^i_1, z^i_2, z^i_3 \} \), \( 1 \leq i \leq m \).

Define \( f(I) = (G, k) \), where \( G = (V, E) \) according to the following rules:

- \( V = \{ v^i_j : 1 \leq i \leq m, 1 \leq j \leq 3 \} \),
- \( v^i_j v^{i'}_{j'} \in E \) if and only if \( i \neq i' \) and \( z^i_j \neq z^{i'}_{j'} \), and
- \( k = m \).
Subset Sum

Problem

Subset Sum

Instance: A list of sizes $S = [s_1, \ldots, s_n]$; and a target sum, $T$. These are all positive integers.

Question: Does there exist a subset $J \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in J} s_i = T$?
**Vertex Cover \(\leq_P\) Subset Sum**

Suppose \(I = (G, k)\), where \(G = (V, E)\), \(|V| = n\), \(|E| = m\) and \(1 \leq k \leq n\).

Suppose \(V = \{v_1, \ldots, v_n\}\) and \(E = \{e_0, \ldots, e_{m-1}\}\). For \(1 \leq i \leq n\), \(0 \leq j \leq m - 1\), let

\[
    c_{ij} = \begin{cases} 
        1 & \text{if } e_j \text{ is incident with } v_i \\
        0 & \text{otherwise.}
    \end{cases}
\]

Define \(n + m\) sizes and a target sum \(W\) as follows:

\[
    a_i = 10^m + \sum_{j=0}^{m-1} c_{ij}10^j \quad (1 \leq i \leq n)
\]

\[
    b_j = 10^j \quad (0 \leq j \leq m - 1)
\]

\[
    W = k \cdot 10^m + \sum_{j=0}^{m-1} 2 \cdot 10^j
\]

Then define \(f(I) = (a_1, \ldots, a_n, b_0, \ldots, b_{m-1}, W)\).
Subset Sum $\leq_P$ 0-1 Knapsack

Let $I$ be an instance of Subset Sum consisting of sizes $[s_1, \ldots, s_n]$ and target sum $T$.
Define

$$p_i = s_i, \; 1 \leq i \leq n$$
$$w_i = s_i, \; 1 \leq i \leq n$$
$$M = T$$

Then define $f(I)$ to be the instance of 0-1 Knapsack consisting of profits $[p_1, \ldots, p_n]$, weights $[w_1, \ldots, w_n]$, capacity $M$ and target profit $T$. 
Hamiltonian Cycle $\leq_P$ TSP-Dec

Let $I$ be an instance of Hamiltonian Cycle consisting of a graph $G = (V, E)$.

For the complete graph $K_n$, where $n = |V|$, define edge weights as follows:

$$w(uv) = \begin{cases} 
1 & \text{if } uv \in E \\
2 & \text{if } uv \notin E.
\end{cases}$$

Then define $f(I)$ to be the instance of TSP-Dec consisting of the graph $K_n$, edge weights $w$ and target $T = n$. 
Vertex Cover \( \leq_P \) Hamiltonian Cycle

This transformation is based on a small graph called a widget. Let \( u \neq v \). A widget \( W_{u,v} \) has 12 vertices, denoted \((u, v, 1), \ldots, (u, v, 6)\) and \((v, u, 1), \ldots, (v, u, 6)\).

The widget contains 14 edges:
- A path of five edges passing through the following six vertices:
  \[(u, v, 1), \ldots, (u, v, 6)\].
- A path of five edges passing through the following six vertices:
  \[(v, u, 1), \ldots, (v, u, 6)\].
- Four additional edges:
  \[(u, v, 1)(v, u, 3), (u, v, 3)(v, u, 1), (u, v, 4)(v, u, 6), (u, v, 6)(v, u, 4)\].

Observe that \( W_{u,v} \) is identical to \( W_{v,u} \).
Vertex Cover $\leq_P$ Hamiltonian Cycle (cont.)

There are precisely three ways to traverse all the vertices in a widget $W_{u,v}$:

- Traverse the two paths of length five separately
- Traverse the following path:
  \[(u, v, 1), (u, v, 2), (u, v, 3), (v, u, 1), \ldots, (v, u, 6), (u, v, 4), (u, v, 5), (u, v, 6).\]
- Traverse the following path:
  \[(v, u, 1), (v, u, 2), (v, u, 3), (u, v, 1), \ldots, (u, v, 6), (v, u, 4), (v, u, 5), (v, u, 6).\]

Observe that, in every case, we “enter” and “leave” on the same side.
**Vertex Cover \( \leq_P \text{ Hamiltonian Cycle} \) (cont.)**

Suppose \( I = (G, k) \) is an instance of Vertex Cover, where \( G = (V, E) \).

For each edge \( uv \in E \), construct a widget \( W_{u,v} \).

For each vertex \( u \in V \), let the vertices adjacent to be \( u \) be denoted \( u_1, \ldots, u_\ell \) (the ordering of these \( \ell \) vertices is arbitrary).

Link the \( \ell \) corresponding widgets \( W_{u,u_1}, \ldots, W_{u,u_\ell} \) (which we call a chain) in the same order by introducing edges

\[
(u, u_1, 6)(u, u_2, 1), \quad (u, u_2, 6)(u, u_3, 1), \quad \ldots, \quad (u, u_{\ell-1}, 6)(u, u_\ell, 1).
\]

Create \( k \) new selector vertices, denoted \( s_1, \ldots, s_k \).

Connect each selector vertex to the beginning and end of each chain. For example, given the chain above, we would connect each \( s_i \) to \( (u, u_1, 1) \) and \( (u, u_\ell, 6) \).

The resulting graph defines the instance \( f(I) \).
Reductions among NP-complete Problems (summary)

In the above diagram, arrows denote polynomial reductions.
NP-hard Problems

A problem $\Pi$ is **NP-hard** if there exists a problem $\Pi' \in NPC$ such that $\Pi' \leq_T \Pi$.

Every **NP-complete** problem is automatically **NP-hard**, but there exist **NP-hard** problems that are not **NP-complete**.

Typical examples of **NP-hard** problems are optimization problems corresponding to **NP-complete** decision problems.

For example, $\text{TSP-Optimization} \leq_T \text{TSP-Decision}$ and $\text{TSP-Decision} \in NPC$, so $\text{TSP-Optimization}$ is **NP-hard**.

This is a “trivial” Turing reduction; the reduction in the reverse direction, which was given on slide #173, is more complex.
Undecidable Problems

A decision problem $\Pi$ is **undecidable** if there does not exist an algorithm that solves $\Pi$.

If $\Pi$ is undecidable, then for every algorithm $A$, there exists at least one instance $I \in \mathcal{I}(\Pi)$ such that $A(I)$ does not find the correct answer (“yes” or “no”) in finite time.

**Problem**

**Halting**

**Instance:** A computer program $A$ and input $x$ for the program $A$.

**Question:** When program $A$ is executed with input $x$, will it halt in finite time?
Undecidability of the Halting Problem

Suppose that $Halt$ is a program that solves the Halting Problem. Consider the following algorithm $Strange$.

**Algorithm:** $Strange(A)$

- external $Halt$
- if not $Halt(A, A)$
  - then return (!)
- else
  - \[
    \begin{align*}
    i & \leftarrow 1 \\
    \text{while } i \neq 0 \text{ do } i & \leftarrow i + 1
    \end{align*}
  \]

What happens when we run $Strange(Strange)$?
The Post Correspondence Problem

The following problem is also undecidable.

**Problem**

**Post Correspondence**

**Instance:** two finite lists $\alpha_1, \ldots, \alpha_N$ and $\beta_1, \ldots, \beta_N$ of words over some alphabet $A$ of size $\geq 2$.

**Question:** Does there exist a finite list of indices, say $i_1, \ldots, i_K$, where $i_j \in \{1, \ldots, N\}$ for $1 \leq j \leq N$, such that

$$\alpha_{i_1} \cdots \alpha_{i_K} = \beta_{i_1} \cdots \beta_{i_K},$$

where a “product” of words denotes their concatenation.