# Algorithms and Complexity

## Table of Contents

Chapter 1 – Introduction: Some Representative Problems ................................................................. 3

- Example problem: stable matching ........................................................................................................ 3
- 5 representative problems ............................................................................................................................ 3

Chapter 2 – Basics of Algorithm Analysis ............................................................................................. 5

- Computational Tractability .......................................................................................................................... 5
- Asymptotic order of growth ........................................................................................................................... 5
- Survey of common running times ............................................................................................................... 5

Chapter 3 – Graphs ................................................................................................................................. 6

- Basic definitions ........................................................................................................................................ 6
- Graph representation ..................................................................................................................................... 6
- Paths and connectivity ................................................................................................................................. 6
- Testing bipartiteness .................................................................................................................................... 7
- Connectivity in directed graphs .................................................................................................................. 8
- Directed acyclic graphs ............................................................................................................................... 9

Chapter 4 – Greedy Algorithms ............................................................................................................. 11

- Interval scheduling .................................................................................................................................... 11
- Interval partitioning ................................................................................................................................. 12
- Scheduling to minimise lateness ............................................................................................................... 13
- Optimal caching ......................................................................................................................................... 15
- Shortest paths in a graph ........................................................................................................................... 17

Chapter 5 – Divide and Conquer .......................................................................................................... 23

- Mergesort ................................................................................................................................................ 23
- Counting inversions ..................................................................................................................................... 24
- Closest pair of points ............................................................................................................................... 25

Chapter 6 – Dynamic Programming ..................................................................................................... 29

- Algorithmic Paradigms ............................................................................................................................. 29
- Weighted interval scheduling .................................................................................................................... 29
- Knapsack Problem ...................................................................................................................................... 31
- Sequence alignment ................................................................................................................................. 32
- Sequence alignment in linear space ......................................................................................................... 33
- Shortest Paths ........................................................................................................................................... 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance Vector Protocol</td>
<td>36</td>
</tr>
<tr>
<td>Negative Cycles in a Graph</td>
<td>37</td>
</tr>
<tr>
<td>Chapter 7 – Network Flow</td>
<td>38</td>
</tr>
<tr>
<td>Maximum flow and minimum cut</td>
<td>38</td>
</tr>
<tr>
<td>Choosing good augmenting paths</td>
<td>43</td>
</tr>
<tr>
<td>Bipartite matching</td>
<td>44</td>
</tr>
<tr>
<td>Perfect matching</td>
<td>46</td>
</tr>
<tr>
<td>Disjoint paths</td>
<td>46</td>
</tr>
<tr>
<td>Chapter 8 – NP and Computational Intractability</td>
<td>49</td>
</tr>
<tr>
<td>Polynomial-time reductions</td>
<td>49</td>
</tr>
<tr>
<td>Reduction by simple equivalence</td>
<td>49</td>
</tr>
<tr>
<td>Reduction from special case to general case</td>
<td>50</td>
</tr>
<tr>
<td>Reduction by encoding with gadgets</td>
<td>50</td>
</tr>
<tr>
<td>Definition of NP</td>
<td>51</td>
</tr>
<tr>
<td>NP-Completeness</td>
<td>53</td>
</tr>
<tr>
<td>Sequencing problems</td>
<td>56</td>
</tr>
<tr>
<td>Partitioning problems</td>
<td>58</td>
</tr>
<tr>
<td>Graph colouring</td>
<td>59</td>
</tr>
<tr>
<td>Numerical problems</td>
<td>61</td>
</tr>
<tr>
<td>Co-NP and the asymmetry of NP</td>
<td>61</td>
</tr>
<tr>
<td>A partial taxonomy of hard problems</td>
<td>63</td>
</tr>
<tr>
<td>Chapter 10 – Extending the Limits of Tractability</td>
<td>64</td>
</tr>
<tr>
<td>Coping with NP-completeness</td>
<td>64</td>
</tr>
<tr>
<td>Finding small vertex covers</td>
<td>64</td>
</tr>
<tr>
<td>Solving NP-hard problems on trees</td>
<td>65</td>
</tr>
<tr>
<td>Circular arc colouring</td>
<td>66</td>
</tr>
<tr>
<td>Chapter 11 – Approximation Algorithms</td>
<td>69</td>
</tr>
<tr>
<td>Load balancing</td>
<td>69</td>
</tr>
<tr>
<td>Centre selection</td>
<td>71</td>
</tr>
<tr>
<td>The pricing method: vertex cover</td>
<td>72</td>
</tr>
<tr>
<td>LP rounding: vertex cover</td>
<td>73</td>
</tr>
<tr>
<td>Knapsack problem</td>
<td>75</td>
</tr>
<tr>
<td>The Median Algorithm and Analysis of Quicksort</td>
<td>77</td>
</tr>
<tr>
<td>Randomised analysis of quicksort</td>
<td>79</td>
</tr>
</tbody>
</table>
General

- Lecturer
  - Taso Viglas
  - SIT 413
  - tasos@it.usyd.edu.au
- Assessment
  - 20% Assignments
  - 20% Midterm quiz
  - 60% Final exam
- Reference Book
  - Kleinberg and Tardos – Algorithm Design

Chapter 1 – Introduction: Some Representative Problems

Example problem: stable matching

- Goal: given a set of preferences among two groups, match pairs such that all pairs are stable.
- Unstable: if \( x \) prefers \( y \) to its assigned \( Y \), or if \( y \) prefers \( x \) to its assigned \( X \).
- Stable: naturally desirable conditions
- Proof of correctness
  - Termination – algorithm terminates after at most \( n^2 \) iteration of while loop
  - Perfection – Every entity is matched
  - Stability – no unstable pairs
- Gale & Shapley proved (1962) that it is always possible to solves the stable marriage problem.
- Gale-Shapley algorithm:
  - Initialise each person to be free
  - Each man has a list of preferences for women, and proposes to every woman in order
  - A woman becomes engaged to him if she is free or prefers this man over her current fiancé, otherwise rejects him
  - \( O(n^2) \) time, man-optimal

5 representative problems

- **Interval Scheduling** – find maximum cardinality subset of mutually compatible jobs with start and finish times

\[
\begin{align*}
\text{\begin{tabular}{|c|c|c|}\hline
\text{Job} & \text{Start} & \text{Finish} \\
\hline
a & 0 & 2 \\
b & 1 & 4 \\
c & 3 & 5 \\
d & 5 & 7 \\
e & 6 & 9 \\
f & 8 & 10 \\
g & 9 & 11 \\
h & 10 & 11 \\
\hline
\end{tabular}}
\end{align*}
\]

- \( n \log n \) greedy algorithm
• **Weighted interval scheduling** – find maximum weight subset of mutually compatible jobs with start and finish times, and weights
  
  ![Weighted Interval Scheduling Diagram]

  - \( n \log n \) dynamic programming algorithm

• **Bipartite matching** – find maximum cardinality matching in bipartite graph
  
  ![Bipartite Matching Diagram]

  - \( n^k \) max-flow based algorithm

• **Independent set** – find maximum cardinality independent (subset of nodes such that no two are joined by an edge) set in graph
  
  ![Independent Set Diagram]

  - NP-complete

• **Competitive facility location** – select a maximum weight subset of nodes in graph with weighted nodes
  
  ![Competitive Facility Location Diagram]

  - Game: two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbours have been selected.

  - PSPACE-complete
Chapter 2 – Basics of Algorithm Analysis

Computational Tractability

- **Polynomial time**
  - Natural brute-force search algorithm, checking every possible solution
  - Usually $2^N$ time or worse
  - Desirable scaling property for an algorithm to be polynomial time:
    - There exists constants $c > 0$ and $d > 0$ such that on every input of size $N$, its running time is bounded by $cN^d$ steps.

- **Worst case analysis**
  - Obtain bound using largest possible running time given an input of size $N$
  - Captures general efficiency in practice

- **Average case running time**
  - Hard/Impossible to model real instances by random distribution
  - Algorithm tuned for a certain distribution may perform poorly on other inputs

- **Polynomial time = efficient**
  - In practice, the constant and exponents are usually low.
  - Some algorithms do have high constants/exponents, and are useless in practice
  - Some algorithms have rare worst-case instances

Asymptotic order of growth

- **Upper bounds** – $O(f(n))$
- **Lower bounds** – $\Omega(f(n))$
- **Tight bounds** – $\Theta(f(n))$ i.e. both $O$ and $\Omega$
- **Notation:** Use $T(n) \in O(f(n))$ rather than $'= '$

Survey of common running times

- **Linear**
  - visits each element exactly once
  - E.g. merging
- $O(n \log n)$
  - Arises in divide-and-conquer algorithms

- **Quadratic time**
  - Enumerate all pairs of elements

- **Cubic time**
  - Enumerate all triples of elements

- **Polynomial time $O(n^k)$**
  - Enumerate all subsets of $k$ nodes

- **Exponential time**
  - Enumerate all subsets
Chapter 3 – Graphs

Basic definitions

- Undirected graph $G = (V, E)$
- Captures pairwise relationship between objects
- Graph size: $n = |V|$, $m = |E|$  

Graph representation

- Adjacency matrix
  - $n$-by-$n$ matrix with $A_{uv} = 1$ if $(u, v)$ is an edge
  - 2 representations of each edge
  - Space proportional to $n^2$
  - Checking for edge takes $O(1)$ time
  - Identifying all edges takes $\Theta(n^2)$ time

- Adjacency list
  - 2 representations of each edge
  - Space proportional to $m + n$
  - Checking for edge takes $O(\deg(u))$ time
  - Identifying edges takes $\Theta(m + n)$ time

Paths and connectivity

- A path in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair $v_i, v_{i+1}$ is joined by an edge in $E$
- A path is simple if all nodes are distinct
- An undirected graph is connected if for every pair of nodes $u$ and $v$, there is a path between $u$ and $v$.
- A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k-1$ nodes are all distinct
- An undirected graph is a tree if it is connected and does not contain a cycle
Any two of the following statements imply the third:

- G is connected
- G does not contain a cycle
- G has $n - 1$ edges

**Testing bipartiteness**

- An undirected graph $G = (V, E)$ is **bipartite** if the nodes can be coloured red or blue such that every edge has one red and one blue end.

- If the graph is bipartite, many problems become easier (matching), or tractable (independent set)

**Lemma.** If a graph $G$ is bipartite, it cannot contain an odd length cycle

[Diagram showing bipartite graphs]

[Diagram showing non-bipartite graphs]

**Lemma.** Let $G$ be a connected graph, and let $L_0, ..., L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds:

- No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite
- An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (thus not bipartite)

[Diagram showing cases (i) and (ii)]

**Lemma.** Let $G$ be a connected graph, and let $L_0, ..., L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds:

- No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
  
  **Pf.** Suppose no edge joins two nodes in the layer.
  
  By previous lemma, this implies all edges join nodes on the same level.
  
  Bipartition: red nodes on odd levels, and blue on even levels.

[Diagram showing layers and bipartition]

- An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (thus not bipartite)
• **Pf.** Suppose \((x, y)\) is an edge with \(x, y\) in the same level \(L_j\). Let \(z = \text{lowest common ancestor lca}(x, y)\). Let \(L_i\) be the level containing \(z\). Consider the cycle that takes edge from \(x\) to \(y\), and then from \(y\) to \(z\), and \(z\) to \(x\). Its length is \(1 + (j - 1) + (j - i)\), which is odd.

- **Corollary.** A graph \(G\) is bipartite iff it does not contain odd length cycles.

**Connectivity in directed graphs**

- Directed graphs
  - Graph \(G = (V, E)\) where Edge \((u, v)\) goes from node \(u\) to node \(v\)

- Graph search
  - Directed reachability: Given a node \(s\), find all nodes reachable from \(s\)
  - Directed s-t shortest path: What is the length of the shortest path between two nodes \(s\) and \(t\)?
  - Graph search: BFS extends naturally to directed graphs

- Strong connectivity
  - Nodes \(u\) and \(v\) are mutually reachable is there is a path from \(u\) to \(v\) and also from \(v\) to \(u\)
  - A graph is strongly connected if every pair of nodes is mutually reachable
  - **Lemma.** Let \(s\) be any node. \(G\) is strongly connected \(\iff\) every node is reachable from \(s\), and \(s\) is reachable from any node.
    - **Pf.** \(\Rightarrow\) Follows from definition
    - **Pf.** \(\Leftarrow\) Path from \(u\) to \(v\): concatenate \(u\)-s path with \(s\)-v path
      Path from \(v\) to \(u\): concatenate \(v\)-s path with \(s\)-u path
  - **Theorem.** Can determine if \(G\) is strongly connected in \(O(m + n)\) time.
    - **Pf.** Pick any node \(s\).
      Run BFS from \(s\) in \(G\), then run BFS from \(s\) in \(G_{rev}\).
      Return true if and only if all nodes reached in both BFS executions.
Correctness follows immediately from previous lemma.

Directed acyclic graphs

- **Def.** A DAG is a directed graph that contains no directed cycles
- **Ex.** Precedence constraints: edge \((v_i, v_j)\) means node \(v_i\) must occur before \(v_j\)
- **Def.** A topological order of a directed graph \(G = (V, E)\) is an ordering of its nodes as \(v_1, v_2, \ldots, v_n\) so that for every edge \((v_i, v_j)\), we have \(i < j\).

- **Lemma.** If \(G\) is a DAG, then \(G\) has a node with no incoming edges.
  - **Pf.** (by contradiction)
    - Suppose the \(G\) is a DAG and every node has at least one incoming edge.
    - Pick any node \(v\), and begin following the edges backward from \(v\).
    - Since \(v\) has at least one incoming edge \((u, v)\), we can walk backward to \(u\).
    - Then, since \(u\) has at least one incoming edge \((x, u)\), we can walk backward to \(x\).
    - Repeat until we visit a node, say \(w\), twice.
    - Let \(C\) denote the sequence of nodes encountered between successive visits to \(w\).
    - \(C\) is a cycle.

- **Lemma.** If \(G\) is a DAG, then \(G\) has a topological ordering.
  - **Pf.** (by induction on \(n\))
    - Base case: true for \(n = 1\)
    - Give \(n > 1\) nodes, find a node \(v\) with no incoming edges.
    - \(G - \{v\}\) is a DAG, since deleting \(v\) cannot create cycles.
    - Inductive hypothesis: \(G - \{v\}\) has a topological ordering.
    - Place \(v\) first in topological ordering; then append nodes of \(G - \{v\}\) in topological order.
    - This is valid since \(v\) has no incoming edges.

To compute a topological ordering of \(G\):
Find a node \(v\) with no incoming edges and order it first
Delete \(v\) from \(G\)
Recursively compute a topological ordering of \(G - \{v\}\)
and append this order after \(v\)

- **Theorem.** Algorithm finds a topological order in \(O(m + n)\) time.
Pf.

- Maintain:
  - count[w], remaining number of incoming edges; and
  - S, set of remaining nodes with no incoming edges
- Initialise: O(m + n) via single scan through nodes
- Update: to delete v
  - Remove v from S
  - Decrement count[w] for all edges from v to w, and add w to S if count[w] hits 0
  - This is O(1) per edge
Chapter 4 – Greedy Algorithms

Interval scheduling

- Interval scheduling
  - Job starts at \( s_j \) and finishes at \( f_j \).
  - Two jobs are compatible if they don’t overlap.
  - Goal: find the maximum subset of mutually compatible jobs

- Greedy template. Consider jobs in some order. Take each job if it’s compatible with the ones already taken.
  - Earliest start time – consider jobs in ascending order of start time \( s_j \)
  - Earliest finishing time – consider jobs in ascending order of finish time \( f_j \)
  - Shortest interval – consider jobs in ascending order of interval length \( f_j - s_j \)
  - Fewest conflicts – count number of conflicting jobs \( c_j \) for each job. Schedule in ascending order

- Greedy algorithm
  - Consider jobs in increasing order of finishing time.
    - Sort jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).
    - Jobs selected
      ```
      A ← ∅
      For \( j = 1 \) to \( n \):
        if job \( j \) compatible with \( A \)
          \( A ← A \cup \{j\} \)
      return A
      ```
  - Implementation: \( O(n \log n) \)
    - Remember job \( j^* \) that was last added to \( A \)
    - Job \( j \) is compatible with \( A \) if \( s_j \geq s_{j^*} \)

- Analysis of greedy algorithm
  - Pf. (by contradiction)
    - Assume greedy is not optimal
    - Let \( i_1, i_2, \ldots, i_k \) denote jobs selected by greedy
    - Let \( j_1, j_2, \ldots, j_m \) denote jobs selected by optimal solution, with \( i = j \) up to \( i_r = j_r \) for the largest possible value of \( r \).
Interval partitioning

- Interval partitioning
  - Lecture \( j \) starts at \( s_j \) and finishes at \( f_j \)
  - Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room
  - E.g. this schedule uses 4 classrooms to schedule 10 lectures:

```
<table>
<thead>
<tr>
<th>Time</th>
<th>9:00</th>
<th>9:30</th>
<th>10:00</th>
<th>10:30</th>
<th>11:00</th>
<th>11:30</th>
<th>12:00</th>
<th>12:30</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- E.g. this schedule uses only 3:

```
<table>
<thead>
<tr>
<th>Time</th>
<th>9:00</th>
<th>9:30</th>
<th>10:00</th>
<th>10:30</th>
<th>11:00</th>
<th>11:30</th>
<th>12:00</th>
<th>12:30</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Lower bound on optimal solution
  - **Def.** The **depth** of a set of open intervals is the maximum number that contains any given time.
  - **Key observation.** Number of classrooms needed \( \geq \) depth.
  - **Ex.** Depth of schedule above = 3 \( \Rightarrow \) schedule below is optimal.
  - **Q.** Does there always exist a schedule equal to depth of intervals?

- Greedy algorithm
Consider lectures in increasing order of start time, assigning lecture to any compatible classroom

```
Sort intervals by starting time so that \( s_1 \leq s_2 \leq \ldots \leq s_n \).
\( d \leftarrow 0 \) \quad \text{number of allocated classrooms}

for \( j = 1 \) to \( n \) {
    if (lecture \( j \) is compatible with some classroom \( k \))
        schedule lecture \( j \) in classroom \( k \)
    else
        allocate a new classroom \( d + 1 \)
        schedule lecture \( j \) in classroom \( d + 1 \)
        \( d \leftarrow d + 1 \)
}
```

Implementation. \( O(n \log n) \)
- For each classroom \( k \), maintain the finish time of the last job added.
- Keep the classrooms in a priority queue

**Observation.** Greedy algorithm never schedules two incompatible lectures in the same classroom

**Theorem.** Greedy algorithm is optimal.

**Pf.**
- Let \( d \) = number of classrooms that the greedy algorithm allocates.
- Classroom \( d \) is opened because we needed to schedule a job, say \( j \), that is incompatible with all \( d - 1 \) other classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than \( s_j \).
- Thus, we have \( d \) lectures overlapping at time \( s_j + \varepsilon \).
- Key observation. All schedules use \( \geq d \) classrooms.

### Scheduling to minimise lateness

- **Minimising lateness problem**
  - Single resource processes one job at a time
  - Job \( j \) required \( t_j \) units of processing time and is due at time \( d_j \)
  - If \( j \) starts at time \( s_j \), it finishes at time \( f_j = s_j + t_j \)
  - Lateness \( \ell_j = \max\{0, f_j - d_j\} \)
  - Goal: schedule all jobs to minimise maximum lateness \( L = \max \ell_j \)

**Ex:**

<table>
<thead>
<tr>
<th>( j )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_j )</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>( d_j )</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

- **Greedy template. Consider jobs in some order**
  - Shortest processing time first – Consider jobs in ascending order of processing time \( t_j \)
  - **Counterexample:**
    - Earliest deadline first – Consider jobs in ascending order of deadline \( d_j \)
- Smallest slack – Consider jobs in ascending order of slack $d_j - t_j$

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>10</td>
</tr>
<tr>
<td>$d_j$</td>
<td>10</td>
</tr>
</tbody>
</table>

Counterexample:

- Greedy algorithm.
  - Earliest deadline first

```
Sort n jobs by deadline so that $d_1 \leq d_2 \leq \ldots \leq d_n$

l = 0
for j from 1 to n
  Assign job j to interval $[t, t + t_j]$
  $d_j < t$, $f_j < t + t_j$
  $t \leftarrow t + t_j$

output intervals $[s_j, f_j]$
```

- Observation. There exists an optimal schedule with no idle time

<table>
<thead>
<tr>
<th>$d_j = 6$</th>
<th>$d_j = 9$</th>
<th>$d_j = 14$</th>
<th>$d_j = 15$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Inversions
  - Def. an inversion in schedule $S$ is a pair of jobs $i$ and $j$ such that $i < j$ but $j$ scheduled before $i$

- Observation. Greedy schedule has no inversions
  - If a schedule has an inversion, it has one with pair of inverted jobs scheduled consecutively
  - Claim. Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.
  - Pf. Let $\ell$ be the lateness before the swap, and let $\ell'$ be it afterwards.
    - $\ell'_k = \ell_k$ for all $k \neq i, j$
    - $\ell'_{i} \leq \ell_{i}$
    - If job $j$ is late:
      - Theorem. Greedy schedule $S$ is optimal
        - Pf. Define $S^*$ to be an optimal schedule that has the fewest number of inversions
          - Can assume $S^*$ has no idle time
          - If $S^*$ has no inversions, then $S = S^*$
          - If $S^*$ has an inversion, let $i - j$ be an adjacent inversion
Swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions
This contradicts the definition of $S^*$

Greedy algorithm strategies
- Greedy algorithm stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm’s.
- Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.
- Structural. Discover a simple “structural” bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

Optimal caching
- Caching
  - Cache with capacity to store $k$ items.
  - Sequence of $m$ item requests $d_1, d_2, \ldots, d_m$
  - Cache hit: item already in cache when requested
  - Cache miss: item not already in cache when requested; must bring into cache and evict an existing if full
  - Goal: eviction schedule that minimises number of cache misses.
  - Ex. $K = 2$, initial cache = ab. Requests: a, b, c, b, c, a, a, b. Optimal eviction schedule: 2 cache misses

Optimal offline caching: farthest-in-future
- Farthest-in-future. Evict item in the cache that is not requested until farthest in the future.
  - Current cache: a b c d e f
  - Future queries: g a b c e d a b b c a d f e a f d e f g h ...
  - Theorem. [Bellady, 1960s] FF is optimal eviction schedule.
  - Pf. Algorithm and theorem are intuitive, proof is subtle

Reduced eviction schedules
- Def. A reduced eviction schedule is a schedule that only inserts an item into the cache in a step in which that item is requested.
- Intuition. Can transform an unreduced schedule into a reduced one with no more cache misses
Claim. Can transform any unreduced schedule $S$ into a reduced schedule $S'$ with no more cache misses

**Pf.** (by induction on number of unreduced items)
- Suppose $S$ brings $d$ into the cache at time $t$, without a request.
- Let $c$ be the item $S$ evicts when it brings $d$ into the cache.
- Case 1: $d$ evicted at time $t'$, before the next request for $d$.
- Case 2: $d$ requested at time $t'$ before $d$ is evicted.

Theorem. FF is optimal eviction algorithm

**Invariant:** There exists an optimal reduced schedule $S$ that makes the same eviction schedule $S_{FF}$ through the first $j + 1$ requests.

**Pf.** (by induction on number of requests $j$)
- Let $S$ be reduced schedule that satisfies invariant through $j$ requests. We produce $S'$ that satisfies invariant after $j + 1$ requests.
- Consider $(j + 1)^{th}$ request $d = d_{j+1}$
- Since $S$ and $S_{FF}$ have agreed up until now, their cache contents are the same before request $j + 1$
- Case 1: ($d$ is already in the cache) $S' = S$ satisfies invariant
- Case 2: ($d$ is not in the cache; $S$ and $S_{FF}$ evict the same element) $S' = S$ satisfies invariant
- Case 3: ($d$ is not in the cache; $S_{FF}$ evicts $e$; $S$ evicts $f \neq e$)
  - Begin construction of $S'$ from $S$ by evicting $e$ instead of $f$
    
    | $j$ | same | $e$ | $f$ |
    |-----|------|--|-----|
    | $S$ |
    | $j+1$ | same | $e$ | $d$ |
    | $S'$ |
    
    Now $S'$ agrees with SFF on first $j + 1$ requests
- We show that having element $f$ in cache is no worse than having element $e$
- Let $j'$ be the first time after $j + 1$ that $S$ and $S'$ take a different action (involving $e/f$), and let $g$ be item requested at time $j'$
  
<table>
<thead>
<tr>
<th>$j'$</th>
<th>same</th>
<th>$e$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
  
  Case 3a: $g = e$. Can’t happen with FF since there must be a request for $f$ before $e$.
  
  Case 3b: $g = f$. Element $f$ can’t be in cache of $S$, so let $e'$ be the element that $S$ evicts.
  - If $e' = e$, $S'$ accesses $f$ from cache. Now $S$ and $S'$ have same cache
  - If $e' \neq e$, $S'$ evicts $e'$ and brings $f$ into the cache. Now $S$ and $S'$ have same cache (Note $S'$ is no longer reduced, but can be transformed into a reduced schedule that agrees with $S_{FF}$ through step $j + 1$)
  
  Case 3c: $g \neq e$, $f$. $S$ must evict $e$, other $S'$ would take the same action.
• Online vs offline algorithms
  o Offline – full sequence of requests known a priori
  o Online (reality) – requests are not known in advance
  o LIFO – evict page brought in most recently
  o LRU – evict page least recently used

• Theorem. FF is optimal offline eviction algorithm.
  o Provides basis for understanding and analysing online algorithms
  o LRU is k-competitive
  o LIFO is arbitrarily bad.

Shortest paths in a graph
• Shortest path network.
  o Directed graph \( G = (V, E) \)
  o Source \( s \), destination \( t \)
  o Length \( \ell_e \) = length of edge \( e \)

• Shortest path problem: find the shortest (in terms of cost) directed path from \( s \) to \( t \)

• Dijkstra’s algorithm
  o Maintain a set of explored nodes \( S \) and the shortest path distance \( d(u) \) so far from \( s \) to \( u \)
  o Initialise \( S = \{s\} \), \( d(s) = 0 \)
  o Repeatedly choose unexplored node \( v \) which minimises \( \pi(v) = \min_{e=(u,v) \in S} d(u) + \ell_e \)
  o Add \( v \) to \( S \), and set \( d(v) = \pi(v) \) \( \ell \) is the shortest path to \( u \) in explored part, followed by an edge \((u, v)\)

• Dijkstra: Proof of correctness
  o Invariant. For each node \( u \in S \), \( d(u) \) is the length of the shortest \( s-u \) path.
  o Pf. (by induction on \( |S| \))
    ▪ Base case: \( |S| = 1 \) is trivial
    ▪ Inductive hypothesis: Assume true for \( |S| = k \geq 1 \)
    ▪ Let \( v \) be next node added to \( S \), and let \( u-v \) be the chosen edge.
    ▪ The shortest \( s-u \) path plus \((u, v)\) is an \( s-v \) path of length \( \pi(v) \).
    ▪ Consider any \( s-v \) path \( P \). We’ll see that it’s no shorter than \( \pi(v) \).
    ▪ Let \( x-y \) be the first edge in \( P \) that leaves \( S \), and \( P' \) be the subpath to \( x \).
• P is already too long as soon as it leaves S.

- Dijkstra: Implementation
  - For each unexplored node, explicitly maintain $\pi(v) = \min_{e=(u,v) \in S} d(u) + \ell_e$
  - Next node to explore = node with minimum $\pi(v)$
  - When exploring $v$, for each incident edge $e = (v, w)$, update $\pi(w) = \min\{\pi(w), \pi(v) + \ell_e\}$
  - Efficient implementation. Maintain a priority queue of unexplored nodes, prioritised by $\pi(v)$

<table>
<thead>
<tr>
<th>PQ Operation</th>
<th>Dijkstra</th>
<th>Array</th>
<th>Binary heap</th>
<th>d-way Heap</th>
<th>Fib heap $\dagger$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$1$</td>
</tr>
<tr>
<td>Extract Min</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>Change Key</td>
<td>$m$</td>
<td>$1$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$1$</td>
</tr>
<tr>
<td>IsEmpty</td>
<td>$n$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
</tbody>
</table>

$\dagger$ Individual ops are amortized bounds

- Minimum Spanning Tree
  - Given a connected graph $G = (V, E)$ with real-valued edge weights $c_e$, an MST is a subset of the edges $T \subseteq E$ such that $T$ is a spanning tree whose sum of edge weights is minimised.

  $G = (V, E)$

  $T$, $\Sigma_{e \in T} c_e = 50$

  - Cayley’s Theorem. There are $n^{n-2}$ spanning trees of $K_n$. (So can’t solve by brute force)

- MST: Greedy algorithms
  - Kruskal’s algorithm. Start with $T = \emptyset$. Consider edges in ascending order of cost. Insert edge $e$ in $T$ unless doing so would create a cycle.
  - Reverse-delete algorithm. Start with $T = E$. Consider edges in descending order of cost. Delete edge $e$ from $T$ unless doing so would disconnect $T$.
  - Prim’s algorithm. Start with some root node $s$ and greedily grow a tree $T$ from $s$ outward. At each step, add the cheapest edge $e$ to $T$ that has exactly one endpoint in $T$.
  - Simplifying assumption: All edge costs $c_e$ are distinct

- Cycles and Cuts
  - Cut property. Let $S$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $S$. Then the MST contains $e.$
- **Cycle property.** Let $C$ be any cycle, and let $f$ be the max cost edge belonging to $C$. Then the MST does not contain $f$.

- **Cycle.** Set of edges that form $a$-$b$, $b$-$c$, $c$-$d$, ..., $y$-$z$, $z$-$a$

- **Cutset.** A cut is a subset of nodes $S$. The corresponding cutest $D$ is the subset of edges with exactly one endpoint in $S$.

- **Claim.** A cycle and a cutest interest in an even number of edges. (Cycle-cut intersection)

  - **Pf.** (by picture)

- **Greedy algorithms**
  - **Simplifying assumption:** All edge costs $c_e$ are distinct
  - **Cut property.** Let $S$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $S$. Then the MST $T^*$ contains $e$.
    - **Pf.** (exchange argument)
      - Suppose $e$ does not belong to $T^*$
      - Adding $e$ to $T^*$ creates a cycle $C$ in $T^*$
      - Edge $e$ is both in the cycle $C$ and in the cutest $D$ corresponding to $S$, which means there exists another edge, say $f$, that is in both $C$ and $D$.
      - $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
      - Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$
      - This is a contradiction.
o **Cycle property.** Let C be any cycle in G, and let f be the max cost edge belonging to C. Then the MST T* does not contain f.

  - Pf. (exchange argument)
    - Suppose f belongs to T*
    - Deleting f from T* creates a cut S in T*
    - Edge f is both in the cycle C and in the cutset D corresponding to S, which means there exists another edge, say e, that is in both C and D.
    - T' = T* ∪ {e} – {f} is also a spanning tree.
    - Since c_e < c_f, cost(T') < cost(T*)
    - This is a contradiction.

o **Prim’s algorithm:** proof of correctness

  - [Jarník 1930, Dijkstra 1957, Prim 1959]
  - Initialise S = any node
  - Apply cut property to S
  - Add min cost edge in cutset corresponding to S to T, and add one new explored node u to S

  ![Prim's Algorithm](image)

  - **Implementation.** Use a priority queue ala Dijkstra
    - Maintain set of explored nodes S
    - For each unexplored node v, maintain attachment cost a[v] = cost of cheapest edge v to a node in S
    - O(n^2) with an array; O(m log n) with a binary heap.

    ```java
    Prim(G, c) {
        foreach (v ∈ V) a[v] ← ∞
        Initialize an empty priority queue Q
        foreach (v ∈ V) insert v onto Q
        Initialize set of explored nodes S ← Ø

        while (Q is not empty) {
            u ← delete min element from Q
            S ← S ∪ {u}
            foreach (edge e = (u, v) incident to u)
                if ((v ∉ S) and (c_e < a[v]))
                    decrease priority a[v] to c_e
        }
    }
    ```

o **Kruskal’s algorithm:** proof of correctness

  - [Kruskal 1956]
  - Consider edges in ascending order of weight.
  - Case 1: if adding e to T creates a cycle, discard e according to cycle property
Case 2: otherwise, insert $e = (u, v)$ into $T$ according to cut property where $S =$ set of nodes in $u$’s connected component.

- **Implementation.** Use the union-find data structure.
  - Build set $T$ of edges in the MST
  - Maintain set for each connected component.
  - $O(m \log n)$ for sorting and $O(m \alpha(m, n))$ for union-find

```java
Kruskal(G, c) {
    Sort edges weights so that $c_1 < c_2 < \ldots < c_m$.
    $T \leftarrow \emptyset$
    foreach ($u \in V$) make a set containing singleton $u$
    for $i = 1$ to $m$ are $u$ and $v$ in different connected components?
      $(u, v) = e_i$
      if ($u$ and $v$ are in different sets) {
        $T \leftarrow T \cup \{e_i\}$
        merge the sets containing $u$ and $v$
      }
    return $T$
}
```

- **Lexicographic tiebreaking**
  - To remove the assumption that all edge costs are distinct, perturb all edge costs by tiny amounts to break any ties.
  - **Impact.** Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.
  - **Implementation.** Can handle arbitrarily small perturbations implicitly by breaking ties lexicographically, according to index.

```java
boolean less(i, j) {
    if (cost(e_i) < cost(e_j)) return true
    else if (cost(e_i) > cost(e_j)) return false
    else if (i < j) return true
    else return false
}
```

- **Clustering**
  - Given a set $U$ of $n$ objects labelled $p_1, p_2, \ldots, p_n$ classify into coherent groups
    - Distance function. Numeric value specifying “closeness” of two objects
    - Fundamental problem. Divide into clusters so that points in different clusters are far apart
  - **K-clustering.** Divide objects into $k$ non-empty groups
  - **Distance function.** Assume it satisfies several natural properties
    - Identity of indiscernibles $d(p_i, p_j) = 0$ iff $p_i = p_j$
    - Nonnegativity $d(p_i, p_j) \geq 0$
    - Symmetry $d(p_i, p_j) = d(p_j, p_i)$
  - **Spacing.** Minimum distance between any pair of points in different clusters
Clustering of maximum spacing. Given an integer $k$, find a $k$-clustering of maximum spacing.

Single link $k$-clustering algorithm.
- Form a graph on the vertex set $U$, corresponding to $n$ clusters
- Find the closest pair of objects from different clusters, and add an edge between them.
- Repeat $n - k$ times until there are exactly $k$ clusters.

Key observation. This procedure is precisely Kruskal’s algorithm, stopping early.

Remark. Equivalent to finding an MST and deleting the $k - 1$ most expensive edges.

Theorem. Let $C^*$ denote the clustering $C^*_1, ..., C^*_k$ formed by deleting the $k - 1$ most expensive edges of a MST. $C^*$ is a $k$-clustering of max spacing.

Pf. Let $C$ denote some other clustering $C_1, ..., C_k$
- The spacing of $C^*$ is the length $d^*$ of the $(k - 1)^{st}$ most expensive edge.
- Let $p_i, p_j$ be in the same cluster in $C^*$, say $C^*_r$, but different clusters in $C$, say $C_s$ and $C_t$
- Some edge $(p, q)$ on $p_i-p_j$ path in $C^*_r$ spans two different clusters in $C$
- All edges on $p_i-p_j$ path have length $\leq d^*$ since Kruskal chose them
- Spacing of $C$ is $\leq d^*$ since $p$ and $q$ are in different clusters
**Chapter 5 – Divide and Conquer**

- **Divide and conquer**
  - Break up problem into several parts
  - Solve each part recursively
  - Combine solutions to sub-problems into overall solution

- **Most common usage**
  - Break up problem of size \( n \) into two equal parts of size \( \frac{n}{2} \)
  - Solve two parts recursively
  - Combine two solutions into overall solution in **linear time**

- **Consequence**
  - Brute force: \( n^2 \)
  - Divide and conquer: \( n \log n \)

**Mergesort**

- **Mergesort**
  - Divide array into two halves
  - Recursively sort each half
  - Merge two halves to make sorted whole

```
ALGORITHM TERMS
ALGORITHM TERMS
AGLOR
HIMST
AGHILMORST
```

- **Merging.** Combine two pre-sorted lists into a sorted whole.
  - Merge efficiently using a linear number of comparisons and a temporary array.

```
AGE1
```

- **A useful recurrence relation**
  - **Def.** \( T(n) \) = number of comparisons to mergesort an input of size \( n \).
  - **Mergesort recurrence.**
    \[
    T(n) \leq \begin{cases} 
    0 & \text{if } n = 1 \\
    2T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n & \text{otherwise} 
    \end{cases}
    \]
  - **Solution.** \( T(n) = O(n \log_2 n) \)
    - In the proofs, initially assume \( n \) is a power of 2, and replace \( \leq \) with =

- **Proof by recursion tree**
• Proof by telescoping
  o Claim. If T(n) satisfies this recurrence, then T(n) = O(n log₂ n)
  o Pf. for n > 1:

\[
\frac{T(n)}{n} = \frac{2T(n/2)}{n} + 1
\]

\[
- \frac{T(n/2)}{n/2} + 1
\]

\[
= \frac{T(n/4)}{n/4} + 1 + 1
\]

\[
\ldots
\]

\[
= \frac{T(n/n)}{n/n} + 1 + \ldots + 1 \frac{1}{log_2 n}
\]

= \log_2 n

• Proof by induction
  o Claim. If T(n) satisfies this recurrence, then T(n) = O(n log₂ n)
  o Pf. (by induction on n)
    ▪ Base case n = 1
    ▪ Inductive hypothesis: T(n) = O(n log₂ n)
    ▪ Goal: show that T(2ⁿ) = O(2ⁿ log₂ 2ⁿ)

\[
T(2ⁿ) = 2T(n) + 2ⁿ
\]

= 2ⁿlog₂ n + 2ⁿ

= 2ⁿlog₂ (2ⁿ) - 1 + 2ⁿ

= 2ⁿlog₂ (2ⁿ)

• Analysis of mergesort recurrence
  o Claim. If T(n) satisfies the recurrence, then T(n) ≤ n \lceil log₂ n \rceil
  o Pf. (by induction on n)
    ▪ Base case: n = 1
    ▪ Define \( n₁ = \left\lfloor \frac{n}{2} \right\rfloor, n₂ = \left\lceil \frac{n}{2} \right\rceil \)
    ▪ Induction step: assume true for 1, 2, ..., n – 1

\[
T(n) ≤ T(n₁) + T(n₂) + n
\]

= \( n₁ \left\lceil \log n₁ \right\rceil + \left\lfloor \log n₂ \right\rceil + n \)

= \( n₁ \left\lceil \log n₁ \right\rceil + \left\lfloor \log n₂ \right\rceil + n \)

= \( n₁ \left\lceil \log n₂ \right\rceil + n \)

= n(\log n₂ - 1) + n

\[
\frac{n₂}{n/2}
\]

≤ \( 2\left\lfloor \frac{\log n₂}{2} \right\rceil + n²/2 \)

≤ \( 2\left\lfloor \frac{\log n₂}{2} - 1 \right\rceil + n²/2 \)

⇒ \log n₂ ≤ \left\lfloor \log n \right\rceil - 1

• Counting inversions
  o Counting inversions
    ▪ Similarity metric: number of inversions between two sequences
    ▪ Items i and j are inverted if i < j, but aᵢ > aⱼ.
    ▪ Brute force: check all \( \Theta(n^2) \) pairs i and j.
• Divide and conquer

\[
\begin{array}{cccccccc}
1 & 5 & 4 & 8 & 10 & 2 & 6 & 9 & 12 & 11 & 3 & 7 \\
\end{array}
\]

Divide: \( O(1) \).

\[
\begin{array}{cccccccc}
1 & 5 & 4 & 8 & 10 & 2 & 6 & 9 & 12 & 11 & 3 & 7 \\
\end{array}
\]

Conquer: \( 2T(n/2) \).

5 blue-blue inversions
8 green-green inversions

9 blue-green inversions
5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Combine: ???

Total = 5 + 8 + 9 = 22.

- Divide: separate list into two pieces
- Conquer: recursively count inversions in each half
- Combine: count inversions where \( a_i \) and \( a_j \) are in different halves, and return sum of three quantities
  - Assume each half is sorted
  - Count inversions where \( a_i \) and \( a_j \) are in different halves
  - Merge two sorted halves into a sorted whole (maintains sorted invariant)

\[
\begin{array}{cccccccc}
3 & 7 & 10 & 14 & 18 & 19 \\
6 & 11 & 16 & 17 & 23 & 25 \\
\end{array}
\]

13 blue-green inversions: \( 6 + 3 + 2 + 2 + 0 + 0 \)

Count: \( O(1) \).

\[
\begin{array}{cccccccc}
2 & 3 & 7 & 10 & 11 & 14 & 16 & 17 & 18 & 19 & 23 & 25 \\
\end{array}
\]

Merge: \( O(n) \).

\[
I(n) \leq I\left(\lceil n/2 \rceil \right) + I\left(\lceil n/2 \rceil \right) + O(n) \Rightarrow I(n) = O(n \log n)
\]

• Implementation
  - Pre-condition. (Merge-and-Count) A and B are sorted.
  - Post-condition. (Sort-and-Count) L is sorted.

```
Sort-and-Count(L) {
    if list L has one element
        return 0 and the list L

    divide the list into two halves A and B
    \( (r_A, A) = \) Sort-and-Count(A) \\
    \( (r_B, B) = \) Sort-and-Count(B) \\
    \( (r, L) = \) Merge-and-Count(A, B) \\

    return \( r = r_A + r_B + r \) and the sorted list L
}
```

Closest pair of points

• Closest pair
  - Given \( n \) points in the plane, find a pair with smallest Euclidean distance between them.
  - A fundamental geometric primitive
  - Brute force. Check all pairs of points \( p \) and \( q \) with \( \Theta(n^2) \) comparisons.
  - Assumption. (to make presentation cleaner) No two points have same x coordinate.

• First attempt
  - Divide. Sub-divide region into 4 quadrants.
- **Obstacle.** Impossible to ensure $\frac{n}{4}$ points in each piece.

  ![Obstacle Diagram]

- **Algorithm.**
  - **Divide.** Draw vertical line $L$ so that roughly $\frac{n}{2}$ points on each side.
    ![Divide Diagram]
  - **Conquer.** Find closest pair in each side recursively.
    ![Conquer Diagram]
  - **Combine.** Find closest pair with one point in each side. (seems like $\Theta(n^2)$)
    ![Combine Diagram]

- **Return best of 3 solutions.**
- **Find closest pair with one point on each side, assuming that distance < $\delta$**
  ![Distance Less Than Delta Diagram]

- **Observation.** Only need to consider points within $\delta$ of line $L$
  ![Observation Diagram]
- Sort points in $2\delta$-strip by their $y$ coordinate.

- Only check distances of those within 11 positions in sorted list!

- **Def.** Let $s_i$ be the point in the $2\delta$-strip, with the $i^{th}$ smallest $y$-coordinate

- **Claim.** If $|i - j| \geq 12$, then the distance between $s_i$ and $s_j$ is at least $\delta$.
  - **Pf.**
    - No two points lie in same $\frac{\delta}{2}$ by $\frac{\delta}{2}$ box.
    - Two points at least 2 rows apart have distance $\geq 2\left(\frac{\delta}{2}\right)$.

- **Fact.** Still true if we replace 12 with 7.

- Closest pair algorithm

  ```
  Closest-Pair(p_1, ..., p_n) {
    Compute separation line $L$ such that half the points are on one side and half on the other side.
    $\delta_1 = \text{Closest-Pair(left half)}$
    $\delta_2 = \text{Closest-Pair(right half)}$
    $\delta = \min(\delta_1, \delta_2)$
    Delete all points further than $\delta$ from separation line $L$
    Sort remaining points by $y$-coordinate.
    Scan points in $y$-order and compare distance between each point and next 11 neighbors. If any of these distances is less than $\delta$, update $\delta$.
    return $\delta$.
  }
  ```
- **Running time.** \( T(n) \leq 2T\left(\frac{n}{2}\right) + O(n \log n) \rightarrow T(n) = O(n \log^2 n) \)

- Can we achieve \( O(n \log n) \)? Yes.
  - Don’t sort points in strip from scratch each time.
  - Each recursive returns two lists: all points sorted by \( y \), and all points sorted by \( x \) coordinate.
  - Sort by merging two pre-sorted lists.
  - \( T(n) \leq T\left(\frac{n}{2}\right) + O(n) \rightarrow T(n) = O(n \log n) \)
Chapter 6 – Dynamic Programming

Algorithmic Paradigms

- **Greedy.** Build up a solution incrementally, myopically optimising some local criterion.
- **Divide-and-conquer.** Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problem to form solution to original problem.
- **Dynamic programming.** Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
- Bellman (1950s) pioneered the systematic study of dynamic programming

Weighted interval scheduling

- Weighted interval scheduling problem
  - Job \( j \) starts at \( s_j \), finishes at \( f_j \), and has weight or value \( v_j \).
  - Two jobs are compatible if they don’t overlap.
  - Goal: find maximum weight subset of mutually compatible jobs.

- Unweighted interval scheduling review
  - Recall that greedy algorithm works if all weights are 1.
  - **Observation.** Greedy algorithm can fail spectacularly if arbitrary weights are allowed.

- Weighted interval scheduling
  - **Notation.** Label jobs by finishing time: \( f_1 \leq f_2 \leq \ldots \leq f_n \).
  - **Def.** \( p(j) \) = largest index \( i < j \) such that job \( i \) is compatible with \( j \).
  - **Ex.** \( p(8) = 5, p(7) = 3, p(2) = 0 \)

- Dynamic programming: binary choice
  - **Notation.** \( \text{OPT}(j) \) = value of optimal solution to the problem consisting of job requests 1, 2, ..., \( j \)
  - Each case has an optimal substructure:
    - Case 1: \( \text{OPT} \) selects job \( j \).
      - Can’t use incompatible jobs \( p(j) + 1, p(j) + 2, \ldots, j - 1 \)
      - Must include optimal solution consisting of remaining compatible jobs 1, 2, ..., \( p(j) \)
    - Case 2: \( \text{OPT} \) does not select job \( j \).
• Must include optimal solution consisting of remaining compatible jobs 1, 2, ..., j – 1

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \left\{v_j + OPT(j), OPT(j-1)\right\} & \text{otherwise}
\end{cases}
\]

• Brute force algorithm

Input: \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)

Sort jobs by finish times so that \( f_1 < f_2 < \ldots < f_n \).

Compute \( p(1), p(2), \ldots, p(n) \)

Compute-Opt(\( j \)) {
  if (\( j = 0 \))
    return 0
  else
    return \( \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1)) \)
}

- **Observation.** Recursive algorithm fails spectacularly because of redundant sub-problems \( \rightarrow \) exponential algorithms.

- **Ex.** Number of recursive calls for family of “layered” instances grows like Fibonacci sequence.

\[ \text{p(0)=0, p(1)=2, p(2)=3, p(3)=4, p(4)=5} \]


- **Memoisation.** Store results of each sub-problem in a cache; lookup as needed

Input: \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)

Sort jobs by finish times so that \( f_1 < f_2 < \ldots < f_n \).

Compute \( p(1), p(2), \ldots, p(n) \)

for \( j = 1 \) to \( n \)
  \( M[j] = \text{empty} \) \( \leftarrow \) global array
  \( M[j] = 0 \)

M-Compute-Opt(\( j \)) {
  if (\( M[j] \) is empty)
    \( M[j] = \max(v_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j-1)) \)
  return \( M[j] \)
}

- **Running time**
  - **Claim.** Memoised version of algorithm takes exactly \( O(n \log n) \) time.
    - Sort by finish time: \( O(n \log n) \)
    - Computing \( p(\cdot) \): \( O(n) \) after sorting by starting time.
    - M-Compute-Opt(\( j \)): each invocation takes \( O(1) \) time and either:
      - Returns an existing value \( M[j] \), or
      - Fills in one new entry \( M[j] \) and makes two recursive calls
    - Progress measure \( \Phi = \# \) nonempty entries of \( M[] \)
      - Initially \( \Phi = 0 \), throughout \( \Phi \leq n \)
      - Filling in new entry increases \( \Phi \) by 1 \( \rightarrow \) at most \( 2n \) recursive calls.
  - **Remark.** \( O(n) \) if jobs are pre-sorted by start and finish times.

- **Automated memorisation.** Many functional programming languages have built-in support for memorisation

- **Finding a solution**
  - Dynamic programming algorithms compute optimal value. What if we want the solution itself?
Do some post-processing. # of recursive calls \( \leq n \rightarrow O(n) \)

```
Run W-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
  if (j == 0)
    output nothing
  else if (v_j + M[p(j)] > M[j-1])
    print j
    Find-Solution(p(j))
  else
    Find-Solution(j-1)
}
```

**Bottom-up dynamic programming. Unwind recursion**

- **Knapsack Problem**
  - Knapsack problem
  - Given \( n \) objects and a “knapsack”
  - Item \( i \) weighs \( w_i > 0 \) kilograms, and has value \( v_i > 0 \)
  - Knapsack has capacity of \( W \) kilograms
  - Goal: fill knapsack to maximise total value.
  - Ex. \{3, 4\} has value 40.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>7</td>
</tr>
</tbody>
</table>

  - Greedy solution: repeatedly add item with maximum ratio \( \frac{v_i}{w_i} \)
  - Ex. \{5, 2, 1\} achieves only value = 35 \( \rightarrow \) greedy is not optimal

  - Dynamic programming: false start
    - **Def.** \( OPT(i) \) max profit subset of items 1, ..., \( i \).
    - Case 1: \( OPT \) does not select item \( i \).
      - \( OPT \) selects best of 1, 2, ..., \( i-1 \)
    - Case 2: \( OPT \) selects item \( i \).
      - Accepting item \( i \) does not immediately imply that we will have to reject other items.
      - Without knowing what other items were selected before \( i \), we don’t even know if we have enough room for \( i \).
    - Conclusion: need more sub-problems.

  - Dynamic programming: adding a new variable
    - **Def.** \( OPT(i, w) \) max profit subset of items 1, ..., \( i \) with weight limit \( w \).
    - Case 1: \( OPT \) does not select item \( i \).
      - \( OPT \) selects best of \{1, 2, ..., \( i-1 \)\} using weight limit \( w \)
    - Case 2: \( OPT \) selects item \( i \).
      - New weight limit = \( w - w_i \)
      - \( OPT \) selects best of \{1, 2, ..., \( i-1 \)\} using this new weight limit
• Bottom-up solution: fill up an \( n \)-by-\( W \) array

```
Input: \( n, W, x_1, \ldots, x_n, y_1, \ldots, y_n \)
for \( w = 0 \) to \( W \)
  \( M[0, w] = 0 \)
for \( i = 1 \) to \( n \)
  for \( w = 1 \) to \( W \)
    if \( w > x_i \)
      \( M[i, w] = M[i-1, w-1] \)
    else
      \( M[i, w] = \min \{ M[i-1, w], w + M[i-1, w-x_i] \} \)
return \( M[n, W] \)
```

• Running time. \( \Theta(nW) \)
  o Not polynomial in input size – “pseudo-polynomial”
  o Decision version of Knapsack is NP-complete.

• Knapsack approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum.

### Sequence alignment

• Sequence alignment. How similar are two strings, say \( \text{occurrance} \) and \( \text{occurrence} \)?

<table>
<thead>
<tr>
<th>a</th>
<th>c</th>
<th>a</th>
<th>r</th>
<th>m</th>
<th>a</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c</td>
<td>a</td>
<td>r</td>
<td>m</td>
<td>a</td>
<td>m</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
<td>r</td>
<td>m</td>
<td>a</td>
<td>m</td>
<td>a</td>
</tr>
</tbody>
</table>

- **Goal:** given two strings \( X = x_1 x_2 \ldots x_m \) and \( Y = y_1 y_2 \ldots y_n \) find alignment of minimum cost
- **Def.** An alignment \( M \) is a set of ordered pairs \( x_i - y_j \) such that each item occurs in at most one pair and no crossings.
- **Def.** The pair \( x_i - y_j \) and \( x_i' - y_j' \) **cross** if \( i < i' \) but \( j > j' \)

\[
\text{cost}(M) = \sum_{(x_i, y_j)} \alpha_{x_i, y_j} + \sum_{\text{gap}} \delta + \sum_{\text{match}} \delta
\]

• Problem structure
  - **Def.** \( \text{OPT}(i, j) = \) min cost of aligning strings \( x_1 x_2 \ldots x_i \) and \( y_1 y_2 \ldots y_j \)
    - **Case 1:** OPT matches \( x_i - y_j \).
      - Pay mismatch for \( x_i - y_j \) + min cost of aligning two strings \( x_1 x_2 \ldots x_{i-1} \) and \( y_1 y_2 \ldots y_{j-1} \)
    - **Case 2a:** OPT leaves \( x_i \) unmatched.
      - Pay gap for \( x_i \) and min cost of aligning \( x_1 x_2 \ldots x_{i-1} \) and \( y_1 y_2 \ldots y_j \)
    - **Case 2b:** OPT leaves \( y_j \) unmatched.
      - Pay gap for \( y_j \) and min cost of aligning \( x_1 x_2 \ldots x_i \) and \( y_1 y_2 \ldots y_{j-1} \)

\[
\text{OPT}(i, j) = \begin{cases} 
\delta & \text{if } i = 0 \\
\alpha_{x_i, y_j} + \text{OPT}(i-1, j-1) & \text{if } i > 0 \\
\min(\delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1)) & \text{otherwise}
\end{cases}
\]
• Algorithm

```java
Sequence-Alignment(m, n, x₁x₂...xₘ, y₁y₂...yₙ, δ, α) |
  for i = 0 to m
    M[0, i] = iα
  for j = 0 to n
    M[j, 0] = jα
  for i = 1 to m
    for j = 1 to n
      M[i, j] = min(α(xᵢ, yⱼ) + M[i-1, j-1],
                  δ = M[i-1, j],
                  δ = M[i, j-1])
  return M[m, n]
```

- **Analysis.** $Θ(mn)$ time and space
  - English words or sentences: $m, n \leq 10$
  - Computational biology: $m = n = 100,000$. Memory requirements too large

**Sequence alignment in linear space**

- Can we avoid using quadratic space?
  - **Easy.** Optimal value in $O(m + n)$ space and $O(mn)$ time.
    - Compute $OPT(i, \cdot)$ from $OPT(i-1, \cdot)$
    - No longer a simple way to recover alignment itself
  - **Theorem.** [Hirschberg 1975] Optimal alignment in $O(m + n)$ space and $O(mn)$ time.
    - Clever combination of divide-and-conquer and dynamic programming.
    - Inspired by idea of Savitch from complexity theory.

- Edit distance graph.
  - Let $f(i, j)$ be shortest path from $(0, 0)$ to $(i, j)$.
  - Observation: $f(i, j) = OPT(i, j)$.

  ![Edit distance graph](image)

  - Can compute $f(\cdot, j)$ for any $j$ in $O(mn)$ time and $O(m + n)$ space.

  ![Edit distance graph](image)

  - Let $g(i, j)$ be shortest path from $(i, j)$ to $(m, n)$.

  ![Edit distance graph](image)
- Can compute by reversing the edge orientations and inverting the roles of $(0, 0)$ and $(m, n)$

- Can compute $g(\cdot, j)$ for any $j$ in $O(mn)$ time and $O(m + n)$ space.

- **Observation 1.** The cost of the shortest path that uses $(i, j)$ is $f(i, j) + g(i, j)$

- **Observation 2.** Let $q$ be an index that minimises $f(q, \frac{n}{2}) + g(q, \frac{n}{2})$.

  Then the shortest path from $(0, 0)$ to $(m, n)$ uses $(q, \frac{n}{2})$.

- **Divide:** find index $q$ that minimises $f(q, \frac{n}{2}) + g(q, \frac{n}{2})$ using DP. Align $x_q$ and $y_{n/2}$.

- **Conquer:** recursively compute optimal alignment in each piece.
Theorem. Let $T(m, n) = \max$ running time on strings of length at most $m$ and $n$. $T(m, n) = O(mn \log n)$.

Remark. Analysis is not tight because two sub-problems are of size $(q, \frac{n}{2})$ and $(m - q, \frac{n}{2})$.

Running time analysis

Theorem. Let $T(m, n) = \max$ running time on strings of length at most $m$ and $n$. $T(m, n) = O(mn \log n)$.

Pf. (by induction on $n$)

- $O(mn)$ time to compute $f(\cdot, \frac{n}{2})$ and $g(\cdot, \frac{n}{2})$ and find index $q$.
- $T(q, \frac{n}{2}) + T(m - q, \frac{n}{2})$ time for two recursive calls.
- Choose constant $c$ so that:
  
  \[
  \begin{aligned}
  T(m, 2) &\leq cm \\
  T(2, n) &\leq cm \\
  T(m, n) &\leq cmn + T(q, n/2) + T(m - q, n/2)
  \end{aligned}
  \]

  - Base cases: $m = 2$ or $n = 2$.
  - Inductive hypothesis: $T(m, n) \leq 2cmn$.

**Shortest Paths**

- **Shortest path problem.** Given a directed graph $G = (V, E)$ with edge weights $c_{vw}$ (negative weights allowed), find the shortest path from node $s$ to node $t$.
- **Ex.** Nodes represent agents in a financial setting, and $c_{vw}$ is cost of transaction in which we buy from agent $v$ and sell immediately to $w$.
- **Failures:**
  - Dijkstra. Fails on negative edge costs
  - Re-weighting. Adding a constant to every edge weight can fail.

- **Shortest paths: negative cost cycles**
  - If some path from $s$ to $t$ contains a negative cost cycle, there does not exist a shortest $s$-$t$ path; otherwise, there exists one that is simple.

  - **Def.** $OPT(i, v) =$ length of shortest $v$-$t$ path $P$ using at most $i$ edges.
    - Case 1. $P$ uses at most $i - 1$ edges.
      - $OPT(i, v) = OPT(i - 1, v)$
    - Case 2. $P$ uses exactly $i$ edges.
      - If $(v, w)$ is first edge, then $OPT$ uses $(v, w)$, and then selects best $w$-$t$ path using at most $i - 1$ edges.
\[
OPT(i, v) = \begin{cases} 
0 & \text{if } i = 0 \\
\min \left[ OPT(i-1, v), \min_{(v, w) \in E} \left( OPT(i-1, w) + c_{vw} \right) \right] & \text{otherwise}
\end{cases}
\]

Remark. By previous observation, if no negative cycles, then \(OPT(n-1, v) = \text{length of shortest v-t path.}\)

- **Implementation**

```java
 Shortest-Path(G, t) {
     foreach node v ∈ V
         M[i, v] ← ∞
     M[0, t] ← 0
     for i = 1 to n-1
         foreach node v ∈ V
             M[i, v] ← M[i-1, v]
         foreach edge (v, w) ∈ E
             M[i, v] ← min { M[i, v], M[i-1, w] + c_{vw} } 
 }
```

- **Analysis.** \(\Theta(mn)\) time, \(\Theta(n^2)\) space.
- Finding the shortest paths. Maintain a “successor” for each table entry

- **Practical improvements**
- Maintain only one array \(M[v] = \text{shortest v-t path that we have found so far.}\)
- No need to check edges of the form \((v, w)\) unless \(M[w]\) changed in previous iteration.
- **Theorem.** Throughout the algorithm, \(N[v] = \text{length of some v-t path, and after } i \text{ rounds of updates, the}
value \(M[v]\) is no larger than the length of shortest v-t path using \(\leq i\) edges.

- Overall impact.
  - Memory: \(O(m + n)\)
  - Running time: \(O(mn)\) worst case, but substantially faster in practice.

- **Bellman-Ford: efficient implementation**

```java
 Push-Based-Shortest-Path(G, s, t) {
     foreach node v ∈ V {
         M[v] ← ∞
         successor[v] ← φ
     }
     M[t] = 0
     for i = 1 to n-1 {
         foreach node w ∈ V {
             if \((v, w) \in E\) \(\text{and } (M[w] \text{ has been updated in previous iteration)}\) {
                 foreach node v such that \((v, w) \in E\) {
                     if \((M[v] > M[w] + c_{vw})\) {
                         M[v] ← M[w] + c_{vw}
                         successor[v] ← w
                     }
                 }
             }
         }
         If no M[w] \text{ value changed in iteration } i, \text{ stop.}
     }
 }
```

**Distance Vector Protocol**

- Used in communication network, where nodes are routers, edges are links and costs are delays
  - Dijkstra’s require global information of network, while Bellman-Ford uses only local knowledge of neighbours
  - Synchronisation. We don’t expect routers to run into a deadlock. The order in which each foreach loop executes is not important. Moreover, algorithm still converges even if updates are asynchronous.

- **Distance vector protocol**
  - Each router maintains a vector of shortest path lengths to every other node (distances) and the first hop on each path (directions).
Algorithm: each router performs n separate computations, one for each potential destination node.

“Routing by rumour”

Caveat. Edge costs may change or fail completely during the algorithm

Path vector protocols: Link state routing

- Each router also stores the entire path – not just the distance and the first hop
- Based on Dijkstra’s algorithm
- Avoids “counting to infinity” problem, and related difficulties.
- Requires significantly more storage.
- Ex. Border Gateway Protocol (BGP), Open Shortest Path First (OSPF)

Negative Cycles in a Graph

- Detecting negative cycles
  - Lemma. If OPT(n,v) = OPT(n – 1, v) for all v, then no negative cycles.
    - Pf. Bellman-Ford algorithm
  - Lemma. If OPT(n,v) < OPT(n – 1, v) for some node v, then (any) shortest path from v to t contains a cycle W. Moreover, W has negative cost.
    - Pf. (by contradiction)
      - Since OPT(n,v) < OPT(n – 1, v), we know P has exactly n edges.
      - By pigeonhole principle, P must contain a directed cycle W.
      - Deleting W yields a v – t path with < n edges \( \rightarrow W \) has negative cost.

- Theorem. Can detect negative cost cycle in O(mn) time.
  - Add new node t and connect all nodes to t with 0-cost edge.
  - Check if OPT(n,v) = OPT(n – 1, v) for all nodes v.
    - If yes, then no negative cycles
    - If no, then extract cycle from shortest path from v to t

Application: finding an arbitrage opportunity in currency exchange

Summary

- Bellman-Ford. O(mn) time, O(m + n) space.
- Run Bellman-Ford for n iterations (instead of n – 1)
- Upon termination, Bellman-Ford successor variables trace a negative cycle if one exists.
- See p.288 for improved version and early termination rule.
Chapter 7 – Network Flow

Maximum flow and minimum cut

- Max flow and min cut
  - Rich algorithmic problems, with a beautiful duality
  - Cornerstone problems in combinatorial optimisation

- Flow network
  - Abstraction for material flowing through the edges
  - $G = (V,E)$ = directed graph, no parallel edges.
  - Two distinguished nodes: $s = \text{source}, t = \text{sink}$
  - $c(e)$ = capacity of edge $e$

- Cuts
  - **Def.** An $s$-$t$ cut is a partition $(A, B)$ of $V$ with $s \in A$ and $t \in B$.
  - **Def.** The capacity of a cut $(A, V)$ is: $\text{cap}(A, B) = \sum_{e \in A \rightarrow B} c(e)$

- **Min $s$-$t$ cut problem.** Find an $s$-$t$ cut of minimum capacity.
• Flows
  o **Def.** An s-t flow is a function that satisfies:
    - For each $e \in E$: $0 \leq f(e) \leq c(e)$ (capacity)
    - For each $v \in V = \{s, t\}$:
      $\sum_{e \in E \text{ into } v} f(e) = \sum_{e \in E \text{ out of } v} f(e)$ (conservation)
  o **Def.** The value of a flow $f$ is:
    $v(f) = \sum_{e \in E \text{ out of } s} f(e)$

• Maximum flow problem. Find s-t flow of maximum value.

• Flows and cuts
  o **Flow value lemma.** Let $f$ be any flow, and let $(A, B)$ be any s-t cut. The net flow sent across the cut is equal to the amount leaving $s$.
    \[
    \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) = v(f)
    \]

  o **Pf.**
    \[
    v(f) = \sum_{e \text{ out of } s} f(e)
    \]
    \[
    (\text{conservation: all terms except } v = s \text{ are zero}) = \sum_{v \in A} \left( \sum_{e \text{ out of } v} f(e) - \sum_{e \text{ into } v} f(e) \right)
    \]
    \[
    = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)
    \]
Weak duality. Let $f$ be any flow. For any s-t cut $(A, B)$, we have $v(f)$ the value of the flow $\leq \text{cap}(A, B)$ the capacity of the cut.

- Pf.

$$
\begin{align*}
  v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\
  &\leq \sum_{e \text{ out of } A} f(e) \\
  &\leq \sum_{e \text{ out of } A} c(e) \\
  &= \text{cap}(A, B)
\end{align*}
$$

- Certificate of optimality

  - Corollary. Let $f$ be any flow, and let $(A, B)$ be any cut. If $v(f) = \text{cap}(A, B)$, then $f$ is a max flow and $(A, B)$ is a min cut.

Towards a max flow algorithm

- Greedy algorithm.
  - Start with $f(e) = 0$ for all edge $e \in E$
  - Find an s-t path $P$ where each edge has $f(e) < c(e)$, i.e. not at capacity
  - Augment flow along path $P$.
  - Repeat until you get stuck
- But getting stuck might be a local optimum, not global optimum

- Residual graph: $G_f = (V, E_f)$
  - **Original edge**: $e = (u, v) \in E$. Flow $f(e)$, capacity $c(e)$
  - **Residual edge**: “undo” flow sent $e = (u, v), e^R = (v, u)$
  - **Residual capacity**: $e_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e) & \text{if } e^R \in E \end{cases}$
  - Residual edges with positive residual capacity
  - $E_f = \{e : f(e) < c(e)\} \cup \{e^R : e(e) > 0\}$

- Ford-Fulkerson algorithm

- **Augmenting path theorem.** Flow $f$ is a max flow iff there are no augmenting paths.
- **Max-flow min-cut theorem.** [Ford-Fulkerson 1956] The value of the max flow is equal to the value of the min cut.
- **Proof strategy.** We prove both simultaneously by showing the TFAE:
  1. There exists a cut $(A, B)$ such that $v(f) = \text{cap}(A, B)$
  2. Flow $f$ is a max flow.
  3. There is no augmenting path relative to $f$.
- 1→2 This was the corollary to weak duality lemma
- 2→3 Contrapositive: If $\exists$ an augmenting path to a flow $f$, $f$ can be improved by sending flow along it.
- 3→1 Let $f$ be flow with no augmenting paths; $A$ be set of vertices reachable from $s$ in residual path.

By definition of $A, s \in A$. By definition of $f, t \notin A$. 

$$v(f) = \sum_{e \in \text{in}(A)} f(e) - \sum_{e \in \text{out}(A)} f(e) = \sum_{e \in \text{out}(A)} c(e) - \text{cap}(A, B)$$


Running time

- Assumption. All capacities are integers between 1 and C.
- **Invariant.** Every flow value \( f(e) \) and every residual capacities \( c_f(e) \) remains an integer throughout the algorithm
- **Theorem.** The algorithm terminates in at most \( v(f^*) \leq nC \) iterations.
  - Pf. Each augmentation increase value by at least 1
- **Corollary.** If \( C = 1 \), Ford-Fulkerson runs in \( O(mn) \) time.
- **Integrality theorem.** If all capacities are integers, then there exists a max flow \( f \) for which every flow value \( f(e) \) is an integer.
  - Pf. Since algorithm terminates, theorem follows from invariant.

**Choosing good augmenting paths**
- **Ford-Fulkerson:** exponential number of augmentations
  - If max capacity is \( C \), then the algorithm can take \( C \) iterations.
  - Selection of augmenting paths may lead to polynomial or exponential algorithms
  - Need to choose augmenting paths efficiently and with few iterations.
- **Edmonds-Karp 1972, Dinitz 1970**
  - Choose augmenting paths with max bottleneck capacity
  - Sufficiently large bottleneck capacity
  - Fewest number of edges
- **Capacity scaling**
  - **Intuition.** Choosing path with highest bottleneck capacity increases flow by max possible amount
    - Don’t worry about finding the absolute highest bottleneck path
    - Maintain scaling parameter \( \Delta \)
    - Let \( G_f(\Delta) \) be the subgraph of the residual graph consisting of only arcs with capacity at least \( \Delta \)
  - **Correctness**
- **Assumption.** All edge capacities are integers between 1 and C
- **Integrality invariant.** All flow and residual capacities are integral
- **Correctness.** If the algorithm terminates, then \( f \) is a max flow
- **Pf.**
  - By integrality invariant, when \( \Delta = 1 \rightarrow G_f(\Delta) = G_t \)
  - Upon termination of \( \Delta = 1 \) phase, there are no augmenting paths
- **Running time**
  - **Lemma 1.** The outer while loop repeats \( 1 + \lfloor \log_2 C \rfloor \) times
    - **Pf.** Initially \( C \leq \Delta \leq 2C. \Delta \) decreases by a factor of 2 each iteration
  - **Lemma 2.** Let \( f \) be the flow at the end of a \( \Delta \)-scaling phase. Then the value of the maximum flow is at most \( v(f) + m \).
    - **Pf.** (almost identical to proof of max-flow min-cut theorem)
  - **Lemma 3.** There are at most \( 2m \) augmentations per scaling phase.
    - Let \( f \) be the flow at the end of the previous scaling phase.
    - By Lemma 2, \( v(f^\ast) \leq v(f) + m(2\Delta) \).
    - Each augmentation in a \( \Delta \)-phase increases \( v(f) \) by at least \( \Delta \)
- **Theorem.** The scaling max-flow algorithm finds a max flow in \( O(m \log C) \) augmentations. It can be implemented to run in \( O(m^2 \log C) \) time

**Bipartite matching**
- Matching
  - **Input.** Undirected graph \( G = (V, E) \)
  - **M \subseteq E** is a matching if each node appears in at most one edge in \( M \)
  - **Max matching.** Find a maximum cardinality matching
• Bipartite matching

  \[
  \begin{array}{c}
  1 & 2 & 3 & 4 & 5 \\
  \hline
  L & 6 & 7 & 8 & R
  \end{array}
  \]

  • Input. Undirected, bipartite graph \( G = (L \cup R, E) \)
  • \( M \subset E \) is a matching if each node appears in at most one edge in \( M \)
  • Max matching. Find a maximum cardinality matching

• Max flow formulation

  \[
  \begin{array}{c}
  1 & 2 & 3 & 4 & 5 \\
  \hline
  L & 6 & 7 & 8 & R
  \end{array}
  \]

  • Create digraph \( G' = (L \cup R \cup \{s, t\}, E') \)
  • Direct all edges from \( L \) to \( R \), and assign infinite (or unit) capacity.
  • Add source \( s \), and unit capacity edges from \( s \) to each node in \( L \)
  • Add sink \( t \), and unit capacity edges from each node in \( R \) to \( t \).

• Proof of correctness

  • **Theorem.** Max cardinality matching in \( G \) = value of max flow in \( G' \)
  
    \[
    \begin{array}{c}
    1 & 2 & 3 & 4 & 5 \\
    \hline
    L & 6 & 7 & 8 & R
    \end{array}
    \]

    • **Pf. \( \leq \)**
      
      • Given max matching \( M \) of cardinality \( k \)
      • Consider flow \( f \) that sends 1 unit along each of \( k \) paths.
      • \( f \) is a flow, and has cardinality \( k \).

    • **Pf. \( \geq \)**
      
      • Let \( f \) be a max flow in \( G' \) of value \( k \).
      • Integrality theorem \( \rightarrow k \) is integral and can assume \( f \) is 0-1.
Consider \( M = \) set of edges from \( L \) to \( R \) with \( f(e) = 1 \)
- Each node in \( L \) and \( R \) participates in at most one edge in \( M \)
- \( |M| = k \): consider cut \((L \cup s, R \cup t)\)

**Perfect matching**
- **Def.** A matching \( M \subseteq E \) is **perfect** if each node appears in exactly one edge in \( M \)
- **Notation.** Let \( S \) be a subset of nodes, and let \( N(S) \) be the set of nodes adjacent to nodes in \( S \).
- **Observation.** If a bipartite graph \( G = (L \cup R, E) \) has a perfect matching, then \( |N(S)| \geq |S| \) for all subsets \( S \subseteq L \)
  - **Pf.** Each node in \( S \) has to be matched to a different node in \( N(S) \)

**Marriage theorem.** [Frobenius 1917, Hall 1935]
- Let \( G = (L \cup R, E) \) be a bipartite graph with \(|L| = |R|\). Then, \( G \) has a perfect matching iff \(|N(S)| \geq |S| \) for all subsets \( S \subseteq L \)
  - **Pf. \( \rightarrow \):** This was the previous observation.
  - **Pf. \( \leftarrow \):** Suppose \( G \) does not have a perfect matching.
    - Formulate as a max flow problem and let \((A, B)\) be min cut in \( G' \)
    - By max-flow min-cut, \( \text{cap}(A, B) < |L| \)
    - Define \( L_A = L \cap A, L_B = L \cap B, R_A = R \cap A \)
    - \( \text{cap}(A, B) = |L_B| + |R_A| \)
    - Since min cut can't use \( \infty \) edges: \( N(L_A) \subseteq R_A \)
    - \(|N(L_A)| \leq |R_A| = \text{cap}(A, B) - |L_B| < |L| - |L_B| = |L_A| \)
    - Choose \( S = L_A \)

**Running time** – depends on max flow algorithm
- Generic augmenting path: \( O(m \text{ val}(f^*)) = O(mn) \)
- Capacity scaling: \( O(m^2 \log C) = O(m^2) \)
- Shortest augmenting path: \( O(mn^{1/2}) \)

**Disjoint paths**
- **Disjoint path problem.** Given a digraph \( G = (V, E) \) and two nodes \( s \) and \( t \), find the max number of edge-disjoint \( s-t \) paths.
- **Def.** Two paths are edge-disjoint if they have no edge in common.

- **Max flow formulation.** Assign unit capacity to every edge

- **Theorem.** Max number edge-disjoint s-t paths equals max-flow value
  - **Pf.** \( \leq \)
    - Suppose there are \( k \) edge-disjoint paths \( P_1, ..., P_k \)
    - Set \( f(e) = 1 \) if \( e \) participates in some path \( P_i \); else set \( f(e) = 0 \)
    - Since paths are edge-disjoint, \( f \) is a flow of value \( k \)
  
- **Pf.** \( \geq \)
  - Suppose max flow value is \( k \)
  - Integrality theorem \( \rightarrow \) there exists 0-1 flow \( f \) of value \( k \)
  - Consider edge \((s, u)\) with \( f(s, u) = 1 \)
    - By conservation, there exists an edge \((u, v)\) with \( f(u, v) = 1 \)
    - Continue until reach \( t \), always choosing a new edge
  - Produces \( k \) (not necessarily simple) edge-disjoint paths
    - Can eliminate cycles to get simple paths if desired

- **Network connectivity**

  - **Network connectivity.** Given a digraph \( G = (V, E) \) and two nodes \( s \) and \( t \), find min number of edges whose removal disconnects \( t \) from \( s \).
  - **Def.** A set of edges \( F \subseteq E \) disconnects \( t \) from \( s \) if all \( s \)-\( t \) paths uses at least one edge in \( F \)
  - **Theorem.** [Menger 1927] The max number of edge-disjoint \( s \)-\( t \) paths is equal to the min number of edges whose removal disconnects \( t \) from \( s \)
    - **Pf.** \( \leq \)
      - Suppose the removal of \( F \subseteq E \) disconnects \( t \) from \( s \), and \( |F| = k \)
      - All \( s \)-\( t \) paths use at least one edge of \( F \), hence the number of edge-disjoint paths is at most \( k \).
\begin{itemize}
\item Pf. \geq
\end{itemize}

- Suppose max number of edge-disjoint paths is \( k \)
- Then max flow value is \( k \)
- Max-flow min-cut \( \rightarrow \) cut \((A, B)\) of capacity \( k \)
- Let \( F \) be set of edges going from \( A \) to \( B \)
- \( |F| = k \) and disconnects \( t \) from \( s \)
Chapter 8 – NP and Computational Intractability

Polynomial-time reductions


<table>
<thead>
<tr>
<th>Yes</th>
<th>Probably no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest path</td>
<td>Longest path</td>
</tr>
<tr>
<td>Matching</td>
<td>3D-matching</td>
</tr>
<tr>
<td>Min cut</td>
<td>Max cut</td>
</tr>
<tr>
<td>2-SAT</td>
<td>3-SAT</td>
</tr>
<tr>
<td>Planar 4-color</td>
<td>Planar 3-color</td>
</tr>
<tr>
<td>Bipartite vertex cover</td>
<td>Vertex cover</td>
</tr>
<tr>
<td>Primality testing</td>
<td>Factoring</td>
</tr>
</tbody>
</table>

- Classify problems
  - Desiderata. Classify problems into those that can be solved in polynomial-time and those that cannot
  - Provably requires exponential-time
  - Huge number of fundamental problems have defied classification for decades – appears to be different manifestations of one really hard problem.

- Polynomial-time reduction
  - Desiderata. Suppose we could solve X in polynomial-time. What else could we solve in polynomial time?
  - Reduction. Problem X polynomial reduces to problem Y if arbitrary instances of x can be solved using:
    - Polynomial number of standard computation steps, plus
    - Polynomial number of calls to oracle that solves problem Y
      - Oracle = computational model supplemented by special piece of hardware that solves instances of Y in a single step
  - Notation. $X \leq_P Y$
  - Remarks.
    - We pay for time to write down instance sent to black box $\rightarrow$ instance of Y must be of polynomial size.
    - Note: Cook reducibility in contrast to Karp reductions
  - Purpose. Classify problems according to relative difficulty
  - Design algorithms. If $X \leq_P Y$ and Y can be solved in polynomial-time, then X can also be solved in polynomial time.
  - Establish intractability. If $X \leq_P Y$ and X cannot be solved in polynomial-time, then Y cannot be solved in polynomial time.
  - Establish equivalence (up to cost of reduction). If $X \leq_P Y$ and $Y \leq_P X$, we use notation $X \equiv_P Y$.

Reduction by simple equivalence

- Independent set. Given a graph $G = (V, E)$ and an integer $k$, is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in $S$?
  - Ex. Is there an independent set of size $\geq 6$? Yes.
  - Ex. Is there an independent set of size $\geq 7$? No.

- Vertex cover. Given a graph $G = (V, E)$ and an integer $k$, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge, at least one of its endpoints is in $S$?
  - Ex. Is there a vertex cover of size $\leq 4$? Yes.
  - Ex. Is there a vertex cover of size $\leq 3$? No.
**Claim.** Vertex Cover \( \equiv_P \) Independent Set.

**Pf.** We show \( S \) is an independent set iff \( V - S \) is a vertex cover

- Pf. \( \Rightarrow \)
  1. Let \( S \) be any independent set.
  2. Consider an arbitrary edge \((u, v)\).
  3. \( S \) independent \( \Rightarrow u \notin S \) or \( v \notin S \) \( \Rightarrow u \in V - S \) or \( v \in V - S \).
  4. Thus, \( V - S \) covers \((u, v)\).

- Pf. \( \Leftarrow \)
  1. Let \( V - S \) be any vertex cover.
  2. Consider two nodes \( u \in S \) and \( v \in S \).
  3. Observe that \((u, v) \notin E\) since \( V - S \) is a vertex cover.
  4. Thus, no two nodes in \( S \) are joined by an edge \( \Rightarrow S \) independent set.

**Reduction from special case to general case**

- **Set Cover.** Given a set \( U \) of elements, a collection \( S_1, S_2, \ldots, S_m \) of subsets of \( U \), and an integer \( k \), does there exist a collection of \( \leq k \) of these sets whose union is equal to \( U \)?

- **Claim.** Vertex Cover \( \leq_P \) Set Cover
  1. Pf. Given a vertex cover instance \( G = (V, E), k \), we construct a set cover instance whose size equals the size of the vertex cover instance.
  2. Construction.
    - Create set-cover instance: \( k = k \), \( U = E \), \( S_v = \{e \in E : e \text{ incident to } v\} \).
    - Set cover of size \( \leq k \) iff vertex cover of size \( \leq k \).

**Reduction by encoding with gadgets**

- **Satisfiability**
  1. **Literal.** A Boolean variable or its negation. \( x_i \) or \( \overline{x_i} \).
  2. **Clause.** A disjunction of literals. \( C_j = x_1 \lor \overline{x_2} \lor x_3 \).
  3. **Conjunctive normal form.** A propositional formula, a conjunction of clauses. \( \Phi = C_1 \land C_2 \land C_3 \land C_4 \).
  4. **SAT.** Given CNF formula \( \Phi \), does it have a satisfying truth assignment?
  5. **3-SAT:** SAT where each clause contains exactly 3 literals (each corresponding to a different variable).
    - Example: \((x_1 \lor \overline{x_2} \lor x_3) \land (\overline{x_1} \lor x_4 \lor x_5) \land (x_6 \lor \overline{x_7} \lor x_8) \land (x_1 \lor x_2 \lor \overline{x_3})\).
    - Yes: \( x_1 \) is true, \( x_2 \) is false, \( x_3 \) is true.

- **3-SAT reduces to independent set**
  1. **Claim.** 3-SAT \( \leq_P \) Independent Set
o **Pf.** Given an instance $\Phi$ of 3-SAT, we construct an instance $(G, k)$ of independent set with size $k$ iff $\Phi$ is satisfiable

o **Construction.**

- $G$ contains 3 vertices for each clause, one for each literal.
- Connect 3 literals in a clause in a triangle
- Connect literal to each of its negations

o **Pf. $\rightarrow$** Let $S$ be independent set of size $k$.
  - $S$ must contain exactly one vertex in each triangle
  - Set these literals to true (and any other variables in a consistent way)
  - Truth assignment is consistent and all clauses are satisfied.

o **Pf. $\leftarrow$** Given satisfying assignment, select one true literal from each triangle. This is an independent set of size $k$.

- **Basic reduction strategies**
  - Simple equivalence independent set $\equiv_P$ vertex cover
  - Special case to general case vertex cover $\leq_P$ set cover
  - Encoding with gadgets 3-SAT $\leq_P$ independent set

- **Transitivity.** If $X \leq_P Y$ and $Y \leq_P Z$, then $X \leq_P Z$
  - Pf idea. Compose the two algorithms
  - Ex. 3-SAT $\leq_P$ independent set $\leq_P$ vertex cover $\leq_P$ set cover

- **Self reducibility**
  - **Decision problem.** Does there exist a vertex cover of size $\leq k$?
  - **Search problem.** Find vertex cover of minimum cardinality
  - **Self reducibility**
    - Search problem $\leq_P$ decision version
    - Applies to all (NP-complete) problems in this chapter
    - Justifies our focus on decision problems
  - Ex. Find min cardinality vertex cover
    - (binary) search for cardinality $k^*$ of min vertex cover
    - Find a vertex $v$ such that $G - \{v\}$ has a vertex cover of size $\leq k^* - 1$
      - Any vertex in any min vertex cover will have this property
    - Include $v$ in the vertex cover
    - Recursively find a min vertex cover in $G - \{v\}$

**Definition of NP**

- **Decision problems**
  - $X$ is a set of strings, and $s$ is an instance of string
  - Algorithm $A$ solves problem $X$: $A(s) = \text{yes}$ iff $s \in X$

- **Polynomial time.** Algorithm $A$ runs in poly-time if for every string $s$, $A(s)$ terminates in at most $p(|s|)$ “steps”, where $P(\cdot)$ is some polynomial.

- **Definition of P**
- NP
  - Certifier views things from “managerial” viewpoint
  - Certifier doesn’t determine whether \( s \in X \) on its own. Rather, it checks a proposed proof \( t \) that \( s \in X \)
  - **Def.** Algorithm \( C(s, t) \) is a certifier for problem \( X \) if for every string \( s \), \( s \in X \) iff there exists a string \( t \) (“certificate” or “witness”) such that \( C(s, t) = \text{yes} \).
  - **NP.** Decision problems for which there exists a poly-time certifier.
  - **Remark.** \( NP \) stands for non-deterministic polynomial-time

- Certifiers and certificates: composite
  - **Composites.** Given an integer \( s \), is \( s \) composite?
  - Certificate. A non-trivial factor \( t \) of \( s \). Such a certificate exists iff \( s \) is composite. Moreover, \( |t| \leq |s| \).
  - Certifier.

```java
boolean C(s, t) {
    if (t = 1 or t = s)
        return false
    else if (s is a multiple of t)
        return true
    else
        return false
}
```

- Ex. Instance \( s = 437669 \); certificate \( t = 541 \) or \( 809 \)
  - Conclusion. Composites is in NP

- Certifiers and certificates: 3-SAT
  - **SAT.** Given a CNF formula \( \Phi \), is there a satisfying assignment?
  - Certificate. An assignment of truth values to the \( n \) Boolean variables
  - Certifier. Check that each clause in \( \Phi \) has at least one true literal
  - Ex. Instance \( (x_1 \lor x_2 \lor x_3) \land (x_1 \lor \bar{x}_2 \lor \bar{x}_3) \land (x_1 \lor \bar{x}_2 \lor x_3) \land (x_2 \lor \bar{x}_3 \lor \bar{x}_1) \) certificate: \( x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1 \)
  - Conclusion. SAT is in NP

- Certifiers and certificates: Hamiltonian cycle
  - **Ham-Cycle.** Given an undirected graph \( G = (V, E) \) does there exist a simple cycle \( C \) that visits every node?
  - Certificate. A permutation of the \( n \) nodes
  - Certifier. Check that the permutation contains each node in \( V \) exactly once, and that there is an edge between each pair of adjacent nodes in the permutation.

- Conclusion. Ham-cycle is in NP.

- **P, NP, EXP**
  - **P.** Decision problems for which there is a polynomial-time algorithm
  - **EXP.** Decision problems for which there is an exponential-time algorithm
- **NP.** Decision problems for which there is a polynomial-time certifier.

- **Claim.** $P \subseteq NP$
  - **Pf.** Consider any problem $X$ in $P$
    - By definition, there exists a poly-time algorithm $A(s)$ that solves $X$
    - Certificate: $t = \varepsilon$, certifier $C(s, t) = A(s)$

- **Claim.** $NP \subseteq EXP$
  - **Pf.** Consider any problem $X$ in $NP$
    - By definition, there exists a poly-time certifier $C(s, t)$ for $X$
    - To solve input $s$, run $C(s, t)$ on all strings $t$ with $|t| \leq p(|s|)$
    - Return yes, if $C(s, t)$ returns yes for any of these

- **P versus NP**
  - Does $P = NP$? [Cook 1971, Edmonds, Levin, Yablonski, Gödel]
    - If yes: efficient algorithms for 3-colour, TSP, factoring, SAT, ... (factoring would break RSA cryptography)
    - If no: no efficient algorithms
    - Consensus opinion: probably no

**NP-Completeness**

- **Polynomial transformation**
  - **Def.** Problem $X$ polynomial reduces (Cook) to problem $Y$ if arbitrary instances of problem $X$ can be solved using:
    - Polynomial number of standard computational steps, plus
    - Polynomial number of calls to oracle that solves problem $Y$
  - **Def.** Problem $X$ polynomial transforms (Karp) to problem $Y$ if given any input $x$ to $X$, we can construct an input $y$ such that $x$ is a yes instance of $X$ iff $y$ is a yes instance of $Y$.
    - Require $|y|$ to be of size polynomial in $|x|$?
  - **Note.** Polynomial transformation is polynomial reduction with just one call to oracle for $Y$, exactly at the end of the algorithm for $X$. Almost all previous reductions were of this form

- **NP-complete**
  - **NP-complete.** A problem $Y$ in $NP$ with the property that for every problem $X$ in $NP$, $X \leq_P Y$
  - **Theorem.** Suppose $Y$ is an NP-complete problem. Then $Y$ is solvable in poly-time iff $P = NP$
  - **Pf.** $\rightarrow$ If $P = NP$ then $Y$ can be solved in poly-time since $Y$ is in $NP$
  - **Pf.** $\leftarrow$ Suppose $Y$ can be solved in poly-time
    - Let $X$ be any problem in $NP$. Since $X \leq_P Y$, we can solve $X$ in poly-time. This implies $NP \subseteq P$
    - We already know $P \subseteq NP$. Thus $P = NP$
• Circuit satisfiability

  o Circuit-SAT. Given a combinational circuit built out of AND, OR, and NOT gates, is there a way to set the circuit inputs so that the output is 1?
  o Circuit-SAT is the “first” NP-complete problem
  o Theorem. Circuit-SAT is NP-complete [Cook 2971, Levin 1973]
  o Pf. (Sketch)
    ▪ Any algorithm that takes a fixed number of bits \( n \) as input and produces a yes/no answer can be represented by such a circuit. Moreover, if algorithm takes poly-time, then circuit is poly-size.
    ▪ (Sketchy part of proof, fixing the number of bits is important, and reflects basic distinction between algorithms and circuits)
    ▪ Consider some problem \( X \) in NP. It has a poly-time certifier \( C(s, t) \). To determine whether \( s \) is in \( X \), need to know if there exists a certificate \( t \) of length \( p(|s|) \) such that \( C(s, t) = \text{yes} \)
    ▪ View \( C(s, t) \) as an algorithm on \( |s| + p(|s|) \) bits (input \( s \), certificate \( t \)) and convert it into a poly-size circuit \( K \).
      • First \( |s| \) bits are hard-coded with \( s \)
      • Remaining \( p(|s|) \) bits represent bits of \( t \)
    ▪ Circuit \( K \) is satisfiable iff \( C(s, t) = \text{yes} \)
  o Ex. Construction below creates a circuit \( K \) whose inputs can be set so that \( K \) outputs true iff graph \( G \) has an independent set of size 2.

• Establishing NP-completeness
  o Remark. Once we establish first “natural” NP-complete problem, others fall like dominoes.
  o Recipe to establish NP-completeness of problem \( Y \).
    ▪ Show that \( Y \) is in NP
    ▪ Choose an NP-complete problem \( X \)
    ▪ Probe that \( X \leq_p Y \)
  o Justification. If \( X \) is an NP-complete problem, and \( Y \) is a problem in NP with the property that \( X \leq_p Y \) then \( Y \) is NP-complete.
**Pf.** Let \( W \) be any problem in \( \text{NP} \). Then \( W \leq_p X \leq_p Y \) (by definition of \( \text{NP-complete} \), and by assumption)

- By transitivity, \( W \leq_p Y \)
- Hence \( Y \) is \( \text{NP-complete} \)

- **3-SAT** is \( \text{NP-complete} \)
  - **Theorem.** 3-SAT is in \( \text{NP-complete} \).
  - **Pf.** Suffices to show that circuit-SAT \( \leq_p \text{3-SAT} \) since 3-SAT is in \( \text{NP} \).

  - Let \( K \) be any circuit.
  - Create a 3-SAT variable \( x_i \) for each circuit element \( i \)
  - Make circuit compute correct values at each node:
    - \( x_2 = \neg x_3 \) \( \rightarrow \) add 2 clauses: \( x_2 \lor x_3, \neg x_2 \lor \neg x_3 \)
    - \( x_4 = x_4 \lor x_5 \) \( \rightarrow \) add 3 clauses: \( x_1 \lor \neg x_4, x_1 \lor \neg x_5, x_1 \lor x_4 \lor x_5 \)
    - \( x_0 = x_1 \lor x_2 \) \( \rightarrow \) add 3 clauses: \( \neg x_0 \lor x_1, \neg x_0 \lor x_2, x_0 \lor \neg x_1 \lor \neg x_2 \)
  - Hard-coded input values and output value.
    - \( x_5 = 0 \) \( \rightarrow \) add 1 clause: \( \neg x_5 \)
    - \( x_0 = 1 \) \( \rightarrow \) add 1 clause: \( x_0 \)
  - Final step: turn clauses of length < 3 into clauses of length exactly 3.

- **NP-completeness**
  - **Observation.** All problems below are \( \text{NP-complete} \) and polynomial reduce to one another!

- 6 basic genres of \( \text{NP-complete} \) problems an paradigmatic examples
  - Packing problems: set-packing, independent set
  - Covering problems: set-cover, vertex-cover
  - Constraint satisfaction problems: SAT, 3-SAT
  - Sequencing problems: Hamiltonian-cycle, TSP
  - Partitioning problems: 3D-Matching, 3-Colour
  - Numerical problems: Subset-sum, Knapsack

- Practice. Most \( \text{NP} \) problems are either known to be in \( \text{P} \) or \( \text{NP-complete} \)
- Notable exceptions. Factoring, graph isomorphism, Nash equilibrium
Sequencing problems

- **Hamiltonian cycle.** Given an undirected graph $G = (V, E)$, does there exist a simple cycle $\Gamma$ that contains every node in $V$?
  - Yes: vertices and faces of a dodecahedron
  - No: bipartite graph with odd number of nodes:

- **Directed Hamiltonian Cycle.** Given a digraph $G = (V, E)$, does there exist a simple directed cycle $\Gamma$ that contains every node in $B$?
  - **Claim.** \( \text{Dir-Ham-cycle} \leq_P \text{Ham-cycle} \)
  - **Pf.** Given a directed graph $G = (V, E)$, construct an undirected graph $G'$ with $3n$ nodes.
    - **Pf. \(\Rightarrow\)**
      - Suppose $G$ has a directed Hamiltonian cycle $\Gamma$
      - Then $G'$ has an undirected Hamiltonian cycle (same order)
    - **Pf. \(\Leftarrow\)**
      - Suppose $G'$ has an undirected Hamiltonian cycle $\Gamma'$
      - $\Gamma'$ must visit nodes in $G'$ using one of the following two orders:
        - ..., $B$, $G$, $R$, $B$, $G$, $R$, $B$, $G$, $R$, $B$, ...
      - Blue nodes in $\Gamma'$ make up directed Hamiltonian cycle $\Gamma$ in $G$, or reverse of one
  - **3-SAT reduces to directed Hamiltonian cycle**
    - **Claim.** \( \text{3-SAT} \leq_P \text{Dir-Ham-cycle} \)
    - **Pf.** Given an instance $\Phi$ of 3-SAT, we construct an instance of Dir-Ham-cycle that has Hamiltonian cycle iff $\Phi$ is satisfiable
o **Construction.** First, create graph that has $2^n$ possible truth assignments

- Given 3-SAT instance $\Phi$ with $n$ variables $x_i$ and $k$ clauses
- Construct G to have $2^n$ Hamiltonian cycles.
- Intuition. Traverse path $i$ from left to right $\leftrightarrow$ set variable $x_i = 1$
- For each clause, add a node and 6 edges:

o **Claim.** $\Phi$ is satisfiable iff G has a Hamiltonian cycle
  - **Pf. $\Rightarrow$**
    - Suppose 3-SAT instance has satisfying assignment $x^*$
    - Then, define Hamiltonian cycle in G as follows:
      o If $x_i^* = 1$, traverse row $i$ from left to right
      o If $x_i^* = 0$, traverse row $i$ from right to left
      o For each clause $C_j$, there will be at least one row $i$ in which we are going in “correct” direction to splice node $C_j$ into four
  - **Pf. $\Leftarrow$**
    - Suppose G has a Hamiltonian cycle $\Gamma$
    - If $\Gamma$ enters clause node $C_j$, it must depart on mate edge
      o Thus, nodes immediately before and after $C_j$ are connected by an edge $e$ in G
      o Removing $C_j$ from cycle, and replacing it with edge $e$ yields Hamiltonian cycle on $G - \{C_j\}$
    - Continuing in this way, we are left with Hamiltonian cycle $\Gamma'$ in $G = \{C_1, C_2, \ldots, C_k\}$
    - Set $x_i^* = 1$ iff $\Gamma'$ traverses row $i$ left to right
    - Since $\Gamma$ visits each clause node $C_j$, at least one of the paths is traversed in “correct” direction, and each clause is satisfied

- **Longest path**
  o **Shortest-path.** Given a digraph $G = (V, E)$, does there exist a simple path of length at most $k$ edges?
  o **Longest-path.** Given a digraph $G = (V, E)$, does there exist a simple path of length at least $k$ edges?
  o **Claim.** 3-SAT $\leq_p$ Longest-path
- **Pf 1.** Redo proof for Dir-Ham-cycle, ignoring back-edge from t to s
- **Pf 2.** Show Ham-cycle $\leq_P$ Longest-path

**Travelling salesman problem**

- **TSP.** Given a set of n cities and a pairwise distance function $d(u, v)$, is there a tour of length $\leq D$?
  - All 13,509 cities in US with a population of at least 500, and the optimal TSP tour:

- **Ham-cycle.** Given a graph $G = (V, E)$, does there exist a simple cycle that contains every node in $V$?
- **Claim.** Ham-cycle $\leq_P$ TSP
- **Pf.**
  - Given instance $G = (V, E)$ of Ham-cycle, create n cities with distance function:
    \[
    d(u, v) = \begin{cases} 
    1 & \text{if } (u, v) \in E \\
    2 & \text{if } (u, v) \notin E
    \end{cases}
    \]
  - TSP instance has tour of length $\leq n$ iff $G$ is Hamiltonian
- **Remark.** TSP instance in reduction satisfies $\Delta$-inequality

**Partitioning problems**

- **3-Dimensional matching**
  - **3D-matching.** Given n instructors, n courses and n times, and a list of the possible courses and times each instructor is willing to teach, is it possible to make an assignment so that all courses are taught at different times?
  - **3D-matching.** Given disjoint sets $X, Y, Z$ each of size n and a set $T \subseteq X \times Y \times Z$ of triples, does there exist a set of n triples in $T$ such that each element of $X \cup Y \cup Z$ is in exactly one of these triples?
  - **Claim.** 3-SAT $\leq_P$ Independent-cover
    - **Pf.** Given an instance $\Phi$ of 3-SAT, we construct an instance of 3D-matching that has a perfect matching iff $\Phi$ is satisfiable
  - **Construction.**
    - Create gadget for each variable $x_i$ with 2k core and tip elements.
    - No other triples will use core elements
    - In gadget $i$, 3D matching must use either both grey (set $x_i = true$) triples or both blue (set $x_i = false$) blue ones.
      - For each clause $C_j$ create two elements and thee triples
      - Exactly one of these triples will be used in any 3D-matching
Ensures any 3D-matching uses either (i) grey core of $x_1$, (ii) blue core of $x_2$ or (iii) grey core of $x_3$

- For each tip, add a cleanup gadget

**Claim.** Instance has a 3D-matching iff $\Phi$ is satisfiable

**Detail.** What are $X$, $Y$, and $Z$? Does each triple contain one element from each of $X$, $Y$ and $Z$?

---

**Graph colouring**

- 3-Colourability
  - **3-Colour.** Given an undirected graph $G$, does there exist a way to colour the nodes red, green and blue so that no adjacent nodes have the same colour?

- **Register allocation.** Assign program variables to machine register so that no more than $k$ registers are used and no two program variables that are needed at the same time are assigned to the same register.

- **Interference graph.** Nodes are program variables names, edge between $u$ and $v$ if there exists an operation where both $u$ and $v$ are "live" at the same time.
- **Observation.** [Chaitin 1982] Can solve register allocation problem iff interference graph is \( k \)-colourable.
- **Fact.** \( 3\text{-Colour} \leq \text{P} \) \( k \text{-register-allocation} \) for any constant \( k \geq 3 \).
- **Claim.** \( 3\text{-SAT} \leq \text{P} \) \( 3\text{-Colour} \)
  - Pf. Given 3-Sat instance \( \Phi \), we construct an instance of 3-colour that is 3-colourable iff \( \Phi \) is satisfiable
- **Construction.**
  i. For each literal, create a node
  ii. Create 3 new nodes \( T, F, B \); connect them in a triangle, and connect each literal to \( B \)
  iii. Connect each literal to its negation
  iv. For each clause, add gadget (to be described next) of 6 nodes and 13 edges
- **Claim.** Graph is 3-colourable iff \( \Phi \) is satisfiable
  - Pf. \( \rightarrow \) Suppose graph is 3-colourable
    - Consider assignment that sets all \( T \) literals to true
    - (ii) ensures each literal is \( T \) or \( F \)
    - (iii) ensures a literal and its negation are opposites
  - Pf. \( \leftarrow \) Suppose 3-SAT formula \( \Phi \) is satisfiable
    - Colour all true literals \( T \)
    - Colour node below green node \( F \), and node below that \( B \)
    - Colour remaining middle row nodes \( B \)
    - Colour remaining bottom nodes \( T \) or \( F \) as forced
Numerical problems

- Subset sum
  - **Subset-sum.** Given natural numbers \( w_1, \ldots, w_n \) and integer \( W \), is there a subset that adds up to exactly \( W \)?
  - **Remark.** With arithmetic problems, input integers are encoded in binary. Polynomial reduction must be polynomial in binary encoding.
  - **Claim.** 3-SAT \( \leq_P \) Subset-sum
  - **Pf.** Given instance \( \Phi \) of 3-SAT, construct an instance of subset-sum that has solution iff \( \Phi \) is satisfiable
  - **Construction.** Given 3-SAT instance \( \Phi \) with \( n \) variables and \( k \) clauses, form \( 2n + 2k \) decimal integers, each of \( n + k \) digits, as illustrated below.
  - **Claim.** \( \Phi \) is satisfiable iff there exists a subset that sums to \( W \).
  - **Pf.** No carries possible.

- Scheduling with release times
  - **Schedule-release-times.** Given a set of \( n \) jobs with processing time \( t_i \), release time \( r_i \), and deadline \( d_i \), is it possible to schedule all jobs on a single machine such that job \( i \) is processed with a contiguous slot of \( t_i \) time units in the interval \([r_i, d_i]\)?
  - **Claim.** Subset-sum \( \leq_P \) Schedule-release times.
  - **Pf.** Given an instance of subset-sum \( w_1, \ldots, w_n \) and target \( W \),
    - Create \( n \) jobs with processing time \( t_i = w_i \), release time \( r_i = 0 \), and no deadline \( (d_i = 1 + \sum_j w_j) \)
    - Create job 0 with \( t_0 = 1 \), release time \( r_0 = W \), and deadline \( d_0 = W + 1 \)

Co-NP and the asymmetry of NP

- **Asymmetry of NP.** We only need to have short proofs of yes instances
  - **Ex 1.** SAT vs Tautology
    - Can prove a CNF formula is satisfiable by giving such an assignment
    - How could we prove that a formula is no satisfiable?
  - **Ex 2.** Ham-cycle vs NO-Ham-cycle
    - Can prove a graph is Hamiltonian by giving such a Hamiltonian cycle.
    - How could we prove that a graph is not Hamiltonian?
  - **Remark.** SAT is NP-complete and SAT \( \equiv_P \) Tautology, but how do we classify Tautology?
    - Not even known to be in NP

- NP and co-NP
  - **NP.** Decision problems for which there is a poly-time certifier
    - Ex. SAT, Ham-cycle, Composites
Def. Given a decision problem X, its complement $\overline{X}$ is the same problem with the yes and no answers reversed.

Co-NP. Complements of decision problems in NP.
- Ex. Tautology, No-Ham-cycle, Primes

NP = co-NP?
- Fundamental question. Does NP = co-NP?
  - Do yes instance have succinct certificates iff no instances do?
  - Consensus opinion: no.

Theorem. If NP $\neq$ co-NP, then P $\neq$ NP.
- Pf idea.
  - P is closed under complementation.
  - If P = NP, then NP is closed under complementation.
  - In other words, NP = co-NP.
  - This is the contrapositive of the theorem

Good characterisations [Edmonds 1965] NP $\cap$ co-NP
- If problem X is in both NP and co-NP, then:
  - For yes instance, there is a succinct certificate
  - For no instance, there is a succinct disqualifier
- Provides conceptual leverage for reasoning about a problem
- Ex. Given a bipartite graph, is there a perfect matching.
  - If yes, can exhibit a perfect matching.
  - If no, can exhibit a set of nodes S such that $|N(S)| < |S|$

Observation. P $\subseteq$ NP $\cap$ co-NP.
- Proof of max-flow min-cut theorem led to stronger result that max-flow and min-cut are in P.
- Sometimes finding a good characterisation seems easier than finding an efficient algorithm.

Prime is in NP $\cap$ co-NP
- Theorem. Primes is in NP $\cap$ co-NP
- Pf. We already know that Primes is in co-NP, so it suffices to prove that Primes is in NP.
- Pratt’s theorem. An odd integer s is prime iff there exists an integer $1 < t < s$ s.t.
  - $t^{s-1} \equiv 1 \pmod{s}$
  - $t^{(s-1)/p} \neq 1 \pmod{s}$ for all prime divisors p of s − 1

Factor is in NP $\cap$ co-NP
- Factorise. Given an integer x, find its prime factorisation.
- Factor. Given two integers x and y, does x have a nontrivial factor less than y?
- Theorem. Factor $\equiv_P$ Factorise
- Pf.
  - Certificate: a factor p of x that is less than y
  - Disqualifier: the prime factorisation of x (where each prime factor is less than y), along with a certificate that each factor is prime.
Primality testing and factoring
   - We established Primes ≤ P Composites ≤ P Factor
   - Natural question: Does Factor ≤ P Primes?
     - Consensus opinion: No.
     - State of the art: Primes is in P (proved in 2001), Factor not believed to be in P
   - RSA cryptosystem
     - Based on dichotomy between complexity of two problems
     - To use RSA, must generate large primes efficiently.
     - To break RSA, suffixes to find efficient factoring algorithm

A partial taxonomy of hard problems
   - Polynomial-time reductions

![Diagram of problem hierarchy including 3-SAT, independent set, vertex cover, and others.](image)
Chapter 10 – Extending the Limits of Tractability

Coping with NP-completeness

- Suppose I need to solve an NP-complete problem. What should I do?
- Theory says you’re unlikely to find a poly-time algorithm
- Must sacrifice one of three desired features
  - Solve problem to optimality
  - Solve problem in polynomial time
  - Solve arbitrary instances of the problem
- This chapter: solve some special cases of NP-complete problems that arise in practice

Finding small vertex covers

- **Vertex cover.** Given a graph \( G = (V, E) \) and an integer \( k \), is there a subset of vertices \( S \subseteq V \) such that \( |S| \leq k \), and for each edge \( (u, v) \) either \( u \in S \) or \( v \in S \), or both?

- What if \( k \) is small?
  - **Brute force.** \( O(kn^{k+1}) \)
    - Try all \( C(n, k) = O(n^k) \) subsets of size \( k \)
    - Takes \( O(kn) \) time to check whether a subset is a vertex cover.
  - **Goal.** Limit exponential dependency on \( k \), e.g. to \( O(2^kkn) \)
  - **Ex.** \( n = 1000, k = 10 \)
    - Brute: \( kn^k+1 \approx 10^{34} \) → infeasible
    - Better: \( 2^kkn = 10^7 \) → feasible
  - **Remark.** If \( k \) is a constant, algorithm is poly-time; if \( k \) is a small constant, then it’s also practical
  - **Claim.** Let \( u – v \) be an edge of \( G \). \( G \) has a vertex cover of size \( \leq k \) iff at least one of \( G – \{u\} \) and \( G – \{v\} \) has a vertex cover of size \( \leq k – 1 \)
    - **Pf. →**
      - Suppose \( G \) has a vertex cover \( S \) of size \( \leq k \)
      - \( S \) contains either \( u \) or \( v \) (or both). Assume it contains \( u \)
      - \( S – \{u\} \) is a vertex cover of \( G – \{u\} \)
    - **Pf. ←**
      - Suppose \( S \) is a vertex cover of \( G – \{u\} \) of size \( \leq k – 1 \)
      - Then \( S \cup \{u\} \) is a vertex cover of \( G \)
- **Algorithm.**
  - **Claim.** The following algorithm determines if \( G \) has a vertex cover of size \( \leq k \) in \( O(2^kkn) \) time

```java
boolean Vertex-Cover(G, k) {
    if (G contains no edges) return true
    if (G contains ≥ kn edges) return false
    let (u, v) be any edge of G
    a = Vertex-Cover(G – {u}, k-1)
    b = Vertex-Cover(G – {v}, k-1)
    return a or b
}
```
Pf.

- Correctness follows previous two claims
- There are $\leq 2^{k+1}$ nodes in the recursion tree; each invocation takes $O(\text{kn})$ time

\[
T(n, k) \begin{cases} \text{cn} & \text{if } k = 1 \\ \geq T(n, k-1) + c kn & \text{if } k > 1 \end{cases} \Rightarrow T(n, k) \leq 2^c k n
\]

Solving NP-hard problems on trees

- **Independent set on trees.** Given a tree, find a max cardinality subset of nodes such that no two share an edge.

  - **Fact.** A tree of at least two nodes has at least two leaf nodes (degree = 1).
  - **Key observation.** If $v$ is a leaf, then there exists a maximum size independent set containing $v$.
  - **Pf.** (exchange argument)
    - Consider a max cardinality independent set $S$
    - If $v \in S$, we’re done
    - If $u \notin S$ and $v \notin S$, then $S \cup \{v\}$ is independent $\Rightarrow S$ not maximum
    - If $u \in S$ and $v \notin S$, then $S \cup \{v\} - \{u\}$ is independent

- **Greedy algorithm**
  - **Theorem.** The following greedy algorithm finds a maximum cardinality independent set in forests (and hence trees)

    ```cpp
    Independent-Set-In-A-Forest(F) {
    S \leftarrow \emptyset
    while (F has at least one edge) {
    Let $e = (u, v)$ be an edge such that $v$ is a leaf
    Add $v$ to $S$
    Delete from $F$ nodes $u$ and $v$, and all edges incident to them.
    } return S
    }
    ```

    - **Pf.** Correctness follows from the previous key observation
    - **Remark.** Can implement in $O(n)$ time by considering nodes in postorder

- **Weighted independent set on trees.** Given a tree and node weights $w_v > 0$, find an independent set $S$ that maximises $\sum_{v \in S} w_v$
  - **Observation.** If $(u, v)$ is an edge such that $v$ is a leaf node, then either OPT includes $u$, or it includes all leaf nodes incident to $u
Dynamic programming solution. Root tree at some node, say $r$

- $\text{OPT}_{\text{in}}(u) = \text{max weight independent set rooted at } u, \text{ containing } u$
- $\text{OPT}_{\text{out}}(u) = \text{max weight independent set rooted at } u, \text{ not containing } u$.

![Dynamic Programming Equation]

Theorem. The dynamic programming algorithm finds a maximum weighted independent set in trees in $O(n)$ time.

Pf. Takes $O(n)$ time since we visit nodes in postorder and examine each edge exactly once.

- Context
  - Independent set on trees. This structured special case is tractable because we can find a node that breaks the communication among the subproblems in different subtrees.

![Diagrams of Communication Breaks]

- Graphs of bounded tree width. Elegant generalisation of trees that
  - Captures a rich class of graphs that arise in practice
  - Enable decomposition into independent pieces

Circular arc colouring

- Wavelength division multiplexing
- **Wavelength-division-multiplexing (WDM).** Allows \( m \) communication streams (arcs) to share a portion of a fibre optic cable, provided they are transmitted using different wavelengths.
- **Ring topology.** Special case if when network is a **cycle** on \( n \) nodes.
- **Bad news.** NP-complete, even on rights.
- **Brute force.** Can determine if \( k \) colours suffice in \( O(km) \) time by trying all \( k \)-colourings.
- **Goal.** \( O(f(k)) \cdot \text{poly}(m, n) \) on rings.

- **Interval colouring.** Greedy algorithm finds colouring such that number of colours equals depth of schedule.

- **Circular arc colouring.**
  - Weak duality: number of colours \( \geq \) depth.
  - Strong duality does not hold.
  - Max depth = 2; min colours = 3.

- **(Almost) transforming circular arc colouring to interval colouring.**
  - **Circular arc colouring.** Given a set of \( n \) arcs with depth \( d \leq k \), can the arcs be coloured with \( k \) colours?
  - **Equivalent problem.** Cut the network between nodes \( v_1 \) and \( v_n \). The arcs can be coloured with \( k \) colours iff the intervals can be coloured with \( k \) colours in such a way that “slices” arcs have the same colour.

- **Dynamic programming algorithm.**
  - Assign distinct colour to each interval which begins at cut node \( v_0 \).
  - At each node \( v_i \), some intervals may finish, and others may begin.
  - Enumerate all \( k \)-colourings of the intervals through \( v_i \) that are consistent with the colourings of the intervals through \( v_{i-1} \).
  - The arcs are \( k \)-colourable iff some colouring of intervals ending at cur node \( v_0 \) is consistent with original colouring of the same intervals.
Running time. $O(k! \cdot n)$
- $n$ phases of the algorithm
- Bottleneck in each phase is enumerating all consistent colourings
- There are at most $k$ intervals through $v_i$, so there are at most $k!$ colourings to consider

Remark. This algorithm is practical for small values of $k$ (say $k = 10$) even if the number of nodes $n$ (or paths) is large
Chapter 11 – Approximation Algorithms

- Theory says it is unlikely to find a poly-time algorithm for an NP-hard problem.
- Must sacrifice one of three desired features: optimality, poly-time or ability to solve arbitrary instances
- $\rho$-approximation algorithm
  - Guaranteed to run in poly-time
  - Guaranteed to solve arbitrary instance of the problem
  - Guaranteed to find solution within ratio $\rho$ of true optimum.
- Challenge. Need to prove a solution's value is close to optimum, without even knowing what optimum value is!

Load balancing

- Load balancing problem. Assign each job to a machine to minimise makespan
  - **Input.** $m$ identical machines, $n$ jobs where job $j$ has processing time $t_j$
    - Each job must run contiguously on one machine
    - Each machine processes one job at a time.
  - **Def.** let $J(i)$ be the subset of jobs assigned to machine $i$. The **load** of machine is $L_i = \sum_{j \in J(i)} t_j$
  - **Def.** The **makespan** is the maximum load on any machine $L = \max_i L_i$
- List scheduling algorithm
  - Consider $n$ jobs in some fixed order.
  - Assign job $j$ to machine whose load is smallest so far.

```
List-Scheduling(m, n, t_1, t_2, ..., t_n) {
  for i = 1 to m {
    L_i = 0          ← load on machine i
    J(i) = ∅         ← jobs assigned to machine i
  }
  for j = 1 to n {
    i = argmin_i L_i ← machine i has smallest load
    J(i) = J(i) U {j} ← assign job j to machine i
    L_i = L_i + t_j ← update load of machine i
  }
}
```
  - **Implementation.** $O(n \log n)$ using a priority queue.
- List scheduling analysis
  - **Theorem.** [Graham, 1966] Greedy algorithm is a 2-approximation.
    - First worst-case analysis of an approximation algorithm
    - Need to compare resulting solution with optimal makespan $L^*$
  - **Lemma.** The optimal makespan $L^* \geq \max_j t_j$
    - **Pf.** Some machine must process the most time-consuming job
  - **Lemma.** The optimal makespan $L^* \geq \frac{1}{m} \sum_j t_j$
    - **Pf.** The total processing time is $\sum_j t_j$
    - One of $m$ machines must do at least a $\frac{1}{m}$ fraction of total work
  - **Theorem.** Greedy algorithm is a 2-approximation
    - **Pf.** Consider load $L_i$ of bottleneck machine $i$
    - Let $j$ be the last job scheduled on machine $i$
    - When job $j$ assigned to machine $i$, $i$ had the smallest load.
• Machine load before assignment is $L_i - t_j \Rightarrow L_i - t_j \leq L_k \forall 1 \leq k \leq m$

\[ L_i - t_j \leq \frac{1}{m} \sum_k L_k \]

\[ = \frac{1}{m} \sum_k t_k \]

Lemma 1 \[ \leq L^* \]

• Sum inequalities over all $k$ and divide by $m$:

\[ L_i - t_j \leq \frac{1}{m} \sum_{k} L_k \]

\[ = \frac{1}{m} \sum_{k} t_k \]

Now:

\[ L_i = \frac{(L_i - t_j)}{L^*} + \frac{t_j}{L^*} \leq 2L^*. \]

Lemma 2

- $m$ machines, $m(m-1)$ jobs of length 1 plus one job of length $m$:

  \[ \text{LPT-List-Scheduling}(m, n, t_1, t_2, ..., t_n) \{
  \text{Sort jobs so that } t_1 \geq t_2 \geq \ldots \geq t_n
  
  \text{for } i = 1 \text{ to } m \{
  L_i = 0 \quad \leftarrow \text{load an machine } i
  
  J(i) = \emptyset \quad \leftarrow \text{jobs assigned to machine } i
  
  \}

  \text{for } j = 1 \text{ to } n \{
  i = \text{argmin}_k L_k \quad \leftarrow \text{machine } i \text{ has smallest load}
  
  J(i) = J(i) \cup \{j\} \quad \leftarrow \text{assign job } j \text{ to machine } i
  
  L_i = L_i + t_j \quad \leftarrow \text{update load of machine } i
  
  \}

  \} \]

- Longest processing time (LPT) rule:
  - Sort $n$ jobs in descending order of processing time, and then run list scheduling algorithm
Observation. If at most m jobs, then list-scheduling is optimal
   Pf. Each job put on its own machine

Lemma. If there are more than m jobs, L* = 2 \text{t}_{m+1}
   \begin{itemize}
     \item Consider first \( m + 1 \) jobs \( t_1, \ldots, t_{m+1} \)
     \item Since the \( t \)'s are in descending order, each takes at least \( t_{m+1} \) time.
     \item At least one machine gets two jobs, by pigeonhole principle \( (m + 1 \text{ jobs and } m \text{ machines}) \)
   \end{itemize}

Theorem. LPT rule is a 3/2 approximation algorithm
   Pf. Same basic approach as for list scheduling

Centre selection
   - Centre selection problem. Select \( k \) centres \( C \) to minimise the maximum distance from a site to nearest centre

Input. Set of \( n \) sites \( s_1, \ldots, s_n \)

Notation.
   \begin{itemize}
     \item \( \text{dist}(x, y) = \text{distance between } x \text{ and } y \)
     \item \( \text{dist}(s_i, C) = \min_{c \in C} \text{dist}(s_i, c) = \text{distance from } s_i \text{ to closest centre} \)
     \item \( r(C) = \max_{i} \text{dist}(s_i, C) = \text{smallest covering radius} \)
   \end{itemize}

Goal. Find set of centres \( C \) that minimises \( r(C) \), subject to \(|C| = k\)

Distance function properties.
   \begin{itemize}
     \item \( \text{dist}(x, x) = 0 \) identity
     \item \( \text{dist}(x, y) = \text{dist}(y, x) \) symmetry
     \item \( \text{dist}(x, y) \leq \text{dist}(x, z) + \text{dist}(z, y) \) triangle inequality
   \end{itemize}

Ex. each site is a point in the plane, a centre can be any point in the plane, \( \text{dist}(x, y) \) – Euclidean distance
   \begin{itemize}
     \item Remark. Search can be infinite.
   \end{itemize}

Greedy algorithm 1.
   \begin{itemize}
     \item Put the first centre at the best possible location for a single centre, and then keep adding centres so as to reduce the covering radius each time by as much as possible.
   \end{itemize}

Remark. Arbitrarily bad.

Greedy algorithm 2. Repeatedly choose the next centre to be the site farthest from any existing centre

```cpp
Greedy-Center-Selection(k, n, s_1, s_2, \ldots, s_n) \{
    C = \emptyset
    \text{repeat } k \text{ times} (\\)
    \quad \text{Select a site } s_i \text{ with maximum } \text{dist}(s_i, C) (\\)
    \quad \text{Add } s_i \text{ to } C (\\)
\}
return C
```

Observation. Upon termination, all centres in C are pairwise at least \( r(C) \) apart
- Pf. By construction of algorithm

Theorem. Let \( C^* \) be an optimal set of centres. Then \( r(C) \leq 2r(C^*) \)
- Pf. (by contradiction)
  - Assume \( r(C^*) \leq \frac{1}{2} r(C) \)
  - Consider ball of radius \( \frac{1}{2} r(C) \) around each site \( c_i \) in C, so there’s exactly one \( c_i^* \) in each
  - Let \( c_i \) be the site paired with \( c_i^* \)
  - Consider any site \( s \) and its closest centre \( c_i^* \) in \( C^* \)
  - \( \text{dist}(s, C) \leq \text{dist}(s, c_i) \leq \text{dist}(s, C_i^*) + \text{dist}(c_i^*, c_i) \) Triangle inequality
  - \( \leq 2r(C^*) \leq r(C^*) \) each since \( c_i^* \) is closer
  - thus \( r(C) \leq 2r(C^*) \)

Theorem. Greedy algorithm is a 2-approximation for centre selection problem
- Remark. Greedy algorithm always places centres at sites, but is still within a factor of 2 of best solution that is allowed to place centres anywhere. (e.g. points in the plane)
- Better approximation only in \( P = NP \)

The pricing method: vertex cover
- Weighted vertex cover. Given a graph \( G \) with vertex weights, find a vertex cover of minimum weight

- Pricing method. Each edge must be covered by some vertex \( i \). Edge \( e \) pays price \( p_e \geq 0 \) to use vertex \( i \)
- Fairness. Edges incident to vertex \( i \) should pay \( \leq w_i \) in total

  \[
  \sum_{i \in V} p_e \geq w_i
  \]

- Claim. For any vertex cover \( S \) and any fair prices \( p_e \), \( \sum_e p_e \leq w(S) \)
  - Pf. \( \sum_{e \in E} p_e \leq \sum_{i \in S} \sum_{e \in \delta(i)} p_e \) each edge \( e \) covered by at least one node in \( S \)
  - \( \leq \sum_{i \in S} w_i \) sum fairness inequalities for each node in \( S \)
  - \( = w(S) \)
- **Pricing method.** Set prices and find vertex cover simultaneously

```plaintext
Weighted-Vertex-Cover-Approx(G, w) {
    foreach e in E
        \( p_e = 0 \)
    while (some edge \( i - j \) such that neither \( i \) nor \( j \) are tight)
        select such an edge \( e \)
        increase \( p_e \) without violating fairness
    S ← set of all tight nodes
    return S
}
```

- **Pricing method analysis**
  - **Theorem.** Pricing method is a 2-approximation
  - **Pf.**
    - Algorithm terminates since at least one new node becomes tight after each iteration of loop
    - Let \( S \) = set of all tight nodes upon termination of algorithm.
      - \( S \) is a vertex cover: if some edge \( i - j \) is uncovered, then neither \( i \) nor \( j \) is tight.
      - But then the loop would not terminate
    - Let \( S^* \) be optimal vertex cover. We show \( w(S) \leq 2w(S^*) \)
      
      \[
      w(S) = \sum_{i \in S} w_i = \sum_{e \in (i, j)} p_e \quad \text{all nodes in } S \text{ are tight}
      \leq \sum_{i \in V} \sum_{e \in (i, j)} p_e \quad S \subseteq V, \text{prices } \geq 0
      = 2 \sum_{e \in E} p_e \quad \text{each edge counted twice}
      \leq 2w(S^*) \quad \text{fairness lemma}
      \]

**LP rounding: vertex cover**

- Weighted vertex cover. Given an undirected graph \( G = (V, E) \) with vertex weights \( w_i \geq 0 \), find a minimum weight subset of nodes \( S \) such that every edge is incident to at least one vertex in \( S \).

```
  10  6  9  16
  6   6  10  9
  6 23  9  33
  7  6  52  52
```

- **Total weight = 55**

- **Integer programming formulation**
  - Model inclusion of each vertex \( i \) using a 0/1 variable \( x_i \)
    - \( x_i = 0 \) if vertex is not in vertex cover
    - \( x_i = 1 \) if vertex is in vertex cover
  - Vertex covers in 1-1 correspondence with 0/1 assignments: \( S = \{ i \in V : x_i = 1 \} \)
  - Objective function: maximise \( \sum_i w_i x_i \)
  - Must take either \( i \) or \( j \): \( x_i + x_j \geq 1 \)
• Weighted vertex cover. Integer programming formulation

\[
\begin{align*}
(\text{ILP}) \min & \quad \sum_{i \in V} w_i x_i \\
\text{s.t.} & \quad x_i + x_j \geq 1, (i,j) \in E \\
& \quad x_i \in \{0,1\}, i \in V
\end{align*}
\]

- **Observation.** If \(x^*\) is optimal solution to ILP, then \(S = \{i \in V : x_i^* = 1\}\) is a min weight vertex cover

• Integer programming. Given integers \(a_{ij}\) and \(b_i\), find integers \(x_j\) that satisfy:

\[
\begin{align*}
\max & \quad \sum_i a_{ij} x_j \\
\text{s.t.} & \quad Ax \geq b \\
& \quad x \text{ integral}
\end{align*}
\]

- **Observation.** Vertex cover formulation proves that integer programming is NP-hard search problem, even if all coefficients are 0/1 and there are at most two variables per inequality.

• Linear programming. Max/min linear objective function subject to linear inequalities.

  - **Input.** Integers \(c_j\), \(b_i\), \(a_{ij}\)
  - **Output.** Real numbers \(x_j\)

\[
\begin{align*}
(\text{P}) \max & \quad \sum_i c_j x_j \\
\text{s.t.} & \quad \sum_i a_{ij} x_j \geq b_i, 1 \leq j \leq m \\
& \quad x_j \geq 0, 1 \leq j \leq n
\end{align*}
\]

- **Linear.** No \(x^2\), \(xy\), \(\arcsin(x)\), \(x(x-1)\), etc.
- **Simplex algorithm.** [Dantzig 1947] Can solve LP in practice
- **Ellipsoid algorithm.** [Khachian 1979] Can solve LP in poly time
- **LP geometry in 2D**

\[
\begin{array}{c}
\text{The region satisfying the inequalities} \\
\implies \begin{cases}
x_1 \geq 0, x_2 \geq 0 \\
x_2 \geq x_1 + 2 \\
x_1 + 2x_2 \geq 6
\end{cases}
\end{array}
\]

• LP relaxation

  - Weighted vertex cover linear programming formulation

\[
\begin{align*}
(\text{LP}) \min & \quad \sum_{i \in V} w_i x_i \\
\text{s.t.} & \quad x_i + x_j \geq 1, (i,j) \in E \\
& \quad x_i \geq 0, i \in V
\end{align*}
\]

- **Observation.** Optimal value of LP \(\leq\) optimal value of ILP
  - **Pf.** LP has fewer constraints
- **Note.** LP is not equivalent to vertex cover
- Using LP to help find a small vertex cover: Solve LP and then round fractional values
- **Theorem.** If \(x^*\) is optimal solution to LP, then \(S = \{i \in V : x_i^* \geq \frac{1}{2}\}\) is a vertex cover whose weight is at most twice the min possible weight.
  - **Pf.** [S as a vertex cover]
    - Consider edge \((i, j) \in E\)
Since \( x_i^* + x_j^* \geq 1 \), either \( x_i^* \geq \frac{1}{2} \) or \( x_j^* \geq \frac{1}{2} \) \( \Rightarrow (i, j) \) covered.

**Pf.** [S has desired cost]

- Let \( S^* \) be optimal vertex cover. Then
  \[
  \sum_{i \in S^*} w_i \geq \frac{1}{2} \sum_{i \in S^*} w_i x_i^* \geq \frac{1}{2} \sum_{i \in S} w_i
  \]
  LP is a relaxation \( x_i^* \geq \frac{1}{2} \)

  - **Theorem.** 2-approximation for weighted vertex cover
  - **Theorem.** [Dinur-Safre 2001] If \( P \neq NP \), no \( \rho \)-approximation for \( \rho < 10\sqrt{5} - 21 \) even with unit weights
  - Open research problem. Close the gap.

**Knapsack problem**

- Polynomial time approximation scheme (PTAS)
  - \((1 + \varepsilon)\)-approximation algorithm for any constant \( \varepsilon > 0 \)
  - Consequence. PTAS produces arbitrarily high quality solution, but trade off accuracy for time.
  - This section. PTAS for knapsack problem via rounding and scaling

- Knapsack is NP-complete
  - **Knapsack.** Given a finite set \( X \), nonnegative weights \( w_i \), nonnegative values \( v_i \), a weight limit \( W \), and a target value \( V \), is there a subset \( S \subseteq X \) such that:
    \[
    \sum_{i \in S} w_i \leq W \quad \sum_{i \in S} v_i \geq V
    \]
  - **Subset-sum.** Given a finite set \( X \), nonnegative values \( u_i \), and an integer \( U \), is there a subset \( S \subseteq X \) whose elements sum to exactly \( U \)?
  - **Claim.** Subset-sum \( \leq_p \) Knapsack.
    - **Pf.** Given instance \((u_1, ..., u_n, U)\) of subset-sum, create knapsack instance:
      \[
      v_i = w_i = u_i \quad \sum_{i \in S} u_i \leq U
      \]
      \[
      V = W = U \quad \sum_{i \in S} u_i \geq U
      \]

- Knapsack problem: dynamic programming 1
  - **Def.** \( \text{OPT}(i, w) = \text{max value subset of items } 1, ..., i \text{ with weight limit } w \).
  - Case 1. \( \text{OPT} \) does not select item \( i \)
    - \( \text{OPT} \) selects the best of \( 1, ..., i - 1 \) using up to weight limit \( w \)
  - Case 2. \( \text{OPT} \) selects item \( i \)
    - New weight limit = \( w - w_i \)
    - \( \text{OPT} \) selects best of \( 1, ..., i - 1 \) using up to weight limit \( w - w_i \)

  - **Running time.** \( O(nW) \)
    - \( W = \text{weight limit} \)
    - Not polynomial in input size!

- Knapsack problem: dynamic programming 2
  - **Def.** \( \text{OPT}(i, v) = \text{min weight subset of items } 1, ..., i \text{ that yields value exactly } v \).
  - Case 1. \( \text{OPT} \) does not select item \( i \)
    - \( \text{OPT} \) selects best of \( 1, ..., i - 1 \) that achieves exactly value \( v \)
  - Case 2. \( \text{OPT} \) selects item \( i \)
    - Consumes weight \( w_i \), new value needed = \( v - v_i \)
OPT selects best of 1, …, i – 1 that achieves value v

\[ OPT(i, v) = \begin{cases} 
0 & \text{if } v = 0 \\
\infty & \text{if } i = 0, v > 0 \\
OPT(i-1, v) & \text{if } v > \nu \\
\min\left\{ OPT(i-1, \nu_i), \nu_i + OPT(i-1, \nu - \nu_i) \right\} & \text{otherwise}
\end{cases} \]

- Running time. \( O(nV^*) = O(n^2v_{\max}) \) \( V^* \leq n\nu_{\max} \)
  - \( V^* = \) optimal value = maximum \( v \) such that \( OPT(n, v) \leq W \)
  - Not polynomial in input size!

- **Knapsack: FPTAS**
  - **Intuition for approximation algorithm**
    - Round all values up to lie in smaller range
    - Run dynamic programming algorithm on rounded instance
    - Return the optimal items in rounded instance
  - **Knapsack FPTAS:** Round up all values:
    - \( \bar{v}_i = \left\lfloor \frac{v_i}{\theta} \right\rfloor \theta \), \( \bar{v}_i = \left\lfloor \frac{v_i}{\theta} \right\rfloor \)
    - \( \nu_{\max} \) = largest value in original instance
    - \( \epsilon \) = precision parameter
    - \( \theta \) = scaling factor = \( \frac{\nu_{\max}}{n} \)

- **Observation.** Optimal solution to problems with \( \bar{v} \) or \( \bar{v} \) are equivalent
- **Intuition.** \( \bar{v} \) close to \( v \) so optimal solution using \( \bar{v} \) is nearly optimal; \( \bar{v} \) is small and integral so dynamic programming algorithm is fast.

- **Running time.** \( O\left(\frac{n^2}{\epsilon}\right) \)
  - Dynamic program 2 running time is \( O(n^2\bar{\nu}_{\max}) \) where \( \bar{\nu}_{\max} = \left\lfloor \frac{\nu_{\max}}{\theta} \right\rfloor = \left\lfloor \frac{n}{\epsilon} \right\rfloor \)

- **Theorem.** If \( S \) is solution found by our algorithm and \( S^* \) is any other feasible solution, then:
  \[
  (1 + \epsilon) \sum_{i \in S} \bar{v}_i \geq \sum_{i \in S^*} v_i
  \]

- **Pf.** Let \( S^* \) be any feasible solutions satisfying weight constraint.
  \[
  \sum_{i \in S^*} v_i \leq \sum_{i \in S} \bar{v}_i \]
  always round up
  \[
  \leq \sum_{i \in S} \bar{v}_i \]
  solve rounded instance optimally
  \[
  \leq \sum_{i \in S} (v_i + \theta) \]
  never round up by more than \( \theta \)
  \[
  \leq \sum_{i \in S} v_i + n\theta \]
  \( |S| \leq n \)
  \[
  \leq (1 + \epsilon) \sum_{i \in S} v_i
  \]
  \( n\theta = \epsilon \nu_{\max}, \nu_{\max} \leq \sum_{i \in S} v_i \)
The Median Algorithm and Analysis of Quicksort

- **Median**: the \( \left( \frac{n}{2} \right) \)-th largest number in a sequence of numbers

- **The peak in a unimodal array**
  - Given an array \( A \) with \( n \) entries, with each entry holding a distinct number
  - The sequence \( A[1], A[2], \ldots, A[n] \) is called unimodal if there exists an index \( p \) between 1 and \( n \), such that the entries increase until position \( p \) and decrease after that.
  - Given a unimodal array, find the peak value – in \( O(\log n) \)

- **The peak value**
  - Divide and conquer (binary search)
  - Look at the middle position \( \frac{n}{2} \), and find the “slope” at that position by looking at one before and after it
  - Continue heading upwards on the slope recursively
  - If item left and right of it are less than the current item, then that is the peak.

- **Analysis of finding peak value**
  - \( T(n) = T \left( \frac{n}{2} \right) + \text{constant, for } n > 1; \quad T(1) = 1 \)
  - Easy to see this is \( O(\log n) \)
    \[ T(n) \leq T \left( \frac{n}{2} \right) + c \]
    \[ \leq \left\lceil \frac{n}{4} \right\rceil + c = T \left( \frac{n}{4} \right) + 2c \]
    \[ \leq T \left( \frac{n}{2k} \right) + kc \]
    \[ \leq 1 + c \log n \]
  - Repeat \( k \) times total, stop when \( \frac{n}{2^k} = 1 \)

- **Divide and conquer for the largest, second largest, or \( k \)-th largest element**
  - \( \text{Top}(x_1, x_2, \ldots, x_n) \)
    - If \( (n \leq k) \) return \( (x_1, x_2, \ldots, x_n) \)
    - \( (a_1, a_2, \ldots, a_k) = \text{Top}(x_1, x_2, \ldots, x_{n/2}) \)
    - \( (b_1, b_2, \ldots, b_k) = \text{Top}(x_{n/2 + 1}, \ldots, x_n) \)
    - Return the top \( k \) from \( (a_1, a_2, \ldots, a_k, b_1, b_2, \ldots, b_k) \)

- **Analysis of finding \( k \)-th largest element**
  - \( T(n) \leq 2T \left( \frac{n}{2} \right) + c \)
    \[ \leq 2 \left[ 2T \left( \frac{n}{2} \right) + c \right] + c = 2^2 T \left( \frac{n}{4} \right) + 2c + c \]
    \[ \leq 2^3 T \left( \frac{n}{8} \right) + 2^2 c + 2c + c = 2^3 T \left( \frac{n}{8} \right) + 2^2 c + 2c + c \]
    \[ \leq 2^m T \left( \frac{n}{2^m} \right) + c[1 + 2 + \cdots + 2^{m-1}] \]
    \[ \leq 2^{\log n} + c \left[ 1 + 2 + \cdots + 2^{\log n - 1} \right] \]
    \[ \leq n + c O(n) = O(n) \]
  - Previous algorithm (recursion) will not give a linear time algorithm.
  - If \( k = \frac{n}{2} \), this is the median problem

- **The median**
  - Can be found by sorting the numbers and taking the \( \left( \frac{n}{2} \right) \)-th element. But can we do it in linear time?
  - **The selection problem**
    - Given a set \( A \) of \( n \) (distinct) numbers and \( 1 \leq i \leq n \), find the element \( x \) that is larger than exactly \( i - 1 \) other elements of \( A \).
    - The median is a special case of the selection problem.
Selecting the \( i \)th largest element in linear time:

- Divide the \( n \) elements into \( \frac{n}{5} \) groups of 5 elements each.
  (And at most one group with the remaining elements)
- Find the median of the groups
  - Sort each (vertical) group of size 5 in constant time
- Recursively find \( x \), the median of the \( \frac{n}{5} \) medians
- Partition the initial array around \( x \), the median of the medians
  (re-arrange groups around the median-of-the-medians)
- Using the partitioning, count the elements to the left of \( x \). Let \( k - 1 \) be the number of elements smaller than \( x \), so that \( x \) is the \( k \)th smallest element in the re-arranged array.
  - If \( i = k \), we’re done, return \( x \).
  - Otherwise if \( i < k \), then recursively find the \( i \)th largest element on the low partition, or the \( (i - k) \)th element on the high partition
  - A bad case is where the partitioning around \( x \) is not so good. So with every recursive call, we do not get rid of enough elements. But this cannot be the case.
- Elements here are larger than \( x \), and elements here are smaller than \( x \).
- The high partition has at least all the blue elements and low partition has at least all the yellow elements. The partitioning is good.
  - Each of the yellow and blue parts have at least \( 3 \left( \frac{n}{2} \right) - 2 \) \( \geq \frac{3n}{10} - 6 \) elements.
  - Every recursive call discards at least this many elements, leaving only \( \frac{7n}{10} + 6 \).
  - So the number of steps required is \( T(n) \leq T \left( \frac{n}{5} \right) + T \left( \frac{7n}{10} + 6 \right) + O(n) \)
  - Solution: \( T(n) = O(n) \)
- In linear time:
  - Break up in groups of 5 – \( O(n) \)
• Sort each group – $O(n)$
• Find median of the medians – $T\left(\frac{n}{5}\right)$
• Partition around the median of the medians – $O(n)$
• If not done, continue recursively on one side of the partition – $T\left(\frac{7n}{10} + 6\right)$

- The master theorem
  - $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ where $a \geq 1$, $b \geq 1$
  - $f(n) = O(n^{\log_b a - \epsilon})$  
    $T(n) = \Theta\left(n^{\log_b a}\right)$
  - $f(n) = \Theta\left(n^{\log_b a}\right)$  
    $T(n) = \Theta\left(n^{\log_b a \log n}\right)$
  - $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$  
    $T(n) = \Theta\left(f(n)\right)$  
    if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ and large $n$

**Randomised analysis of quicksort**

- Quick sort reminder
  - Choose a pivot element $p$
  - Partition array into two: the “smaller than $p$” partition and the “larger than $p$” partition
  - Recursively quick sort the two partitions
  - No merging step required, but if the pivot element leads to unbalanced partitions, then the running time is quadratic.
  - Worst case $O(n^2)$, but in practice, expected running time is fast $O(n \log n)$

- **Sorting.** Given a set of $n$ distinct elements $S$, rearrange them in ascending order.

  ```
  RandomizedQuickSort(S) { 
    if |S| = 0 return
    choose a splitter $a_i$ uniformly at random
    foreach $(a \in S)$ { 
      if $(a < a_i)$ put $a$ in $S'$
      else if $(a > a_i)$ put $a$ in $S''$
    } 
    RandomizedQuickSort($S'$)
    output $a_i$
    RandomizedQuickSort($S''$)
  }
  ```

- **Remark.** Can implement in-place ($\log n$ extra space)

- **Running time.**
  - Best case. Select the median element as the splitter: quicksort makes $\Theta(n \log n)$ comparisons
  - Worst case. Select the smallest element as the splitter: quicksort makes $\Theta(n^2)$ comparisons

- Randomise. Protect against worst case by choosing splitter at random.

- **Intuition.** If we always select an element that is bigger than 25% of the elements and smaller than 25% of the elements, then quicksort makes $\Theta(n \log n)$ comparisons.

- **Notation.** Label elements so that $x_1, x_2 < ... < x_n$
• Splitting.

- BST representation. Draw recursive BST of splitters
- Observation. Element only compared with its ancestors and descendants
  - $x_2$ and $x_7$ are compared if their lca = $x_2$ or $x_7$
  - $x_2$ and $x_7$ are not compared if their lca = $x_3$, $x_4$, $x_5$ or $x_6$

- Claim. $P[x_i$ and $x_j$ are compared] = $\frac{2}{j - (i + 1)}$

- Expected number of comparisons
  - Theorem. Expected number of comparisons is $O(n \log n)$
  - Pf. (Summation of probability that i and j are compared)
    $$\sum_{1 \leq i < j \leq n} \frac{2}{j - i + 1} = 2 \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{j} \leq 2n \sum_{j=1}^{n} \frac{1}{j} \approx 2n \int_{x=1}^{n} \frac{1}{x} dx = 2n \ln n$$
  - Theorem. [Knuth 1973] Stddev of number of comparisons is $\sim 0.65n$
    - Ex. if $n = 1$ million, the probability that randomised quicksort takes less than $4n \ln n$ comparisons is at least 99.94%
  - Chebyshev’s inequality. $P[|X - \mu| \geq k\delta] \leq \frac{1}{k^2}$

- Quicksort analysis
  - Linearity of expectation:
    $$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$$
  - What is the probability that two numbers are compared?
    - i and j are compared only if either is chosen as the first pivot from the set $Y_{ij} = \{y_i, ..., y_j\}$
    - otherwise the two elements will be split in two partitions and will never be compared
o Pivot choice is uniform so all elements in $Y^l$ are equally likely to be chosen
o Probability that $y_i$ or $y_j$ is chosen: $\frac{2}{j-i+1}$
o Let $k = j - i + 1$. All done.

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k}$$

$$= \sum_{k=2}^{n} \sum_{i=1}^{n+1-k} \frac{2}{k}$$

$$= \sum_{k=2}^{n} (n + 1 - k) \frac{2}{k}$$

$$= \left( n + 1 \sum_{k=2}^{n} \frac{2}{k} \right) - 2(n - 1)$$

$$= (2n + 2) \sum_{k=1}^{n} \frac{1}{k} - 4n$$

$$= (2n + 2)H(n) - 4n$$

$$= 2n \log n + O(n)$$