CSE 101

SS 2, 2017

Oliver Braun

Thanks to Andrew B. Kahng, Miles Jones, Russell Impagliazzo, Mohan Paturi.
About this course

CSE101: Design and Analysis of Algorithms

- Complete Enumeration
- Greedy-Algorithms
- Branch-And-Bound
- Dynamic Programming
- Backtracking
- Divide-And-Conquer
- Approximation Algorithms
- Heuristics
- NP-Completeness
- Mathematical Programming for Optimization problems
Welcome Message

Welcome to CSE101. Algorithmic problems arise in every area of the real world and computer science. This course exposes you to a variety of algorithms for problems from various domains. You will learn how the speed of algorithms greatly impact their utility. And you learn elementary mathematical techniques to analyze algorithms for correctness, time complexity, and memory use. You will become familiar with the most efficient algorithms for many classical and also new problems. You will learn the most useful methods and concepts for designing efficient algorithms, and know how to apply them to new problems:

- Complete Enumeration
- Creedy-Algorithms
- Branch-And-Bound
- Dynamic Programming
- Backtracking
- Divide-And-Conquer
- Approximation Algorithms
- Heuristics
- NP-Completeness
- Mathematical Programming for Optimization problems
Structure of CSE 101

1. Algorithms: Algorithmic problems, Analyzing iterative algorithms: Correctness and Time analysis (Minsort), Analyzing recursive algorithms: Correctness and Time analysis (Merging sorted arrays), Divide-And-Conquer (Mergesort)
3. Bin Packing: Complete Enumeration, FirstFit, FirstFitDecreasing, Next Fit, Approximation algorithms and Worst-case analysis
4. Scheduling: Basics, Parallel processor scheduling: McNaughton, Approximation algorithms, Worst-case analysis, Single-processor scheduling: maximum lateness (Earliest-Due-Date), number of delayed jobs (Moore), sum of delays (Complete Enumeration and Branch-and-Bound)
5. Graph algorithms and Shortest Paths: DFS, BFS, Bellman (acyclic networks), Dijkstra (non-negative networks, PQ: Array, Binary Heap), Bellman/Ford
6. Linear Programming: Simplex, Duality
Class Meeting

<table>
<thead>
<tr>
<th></th>
<th>Weekday</th>
<th>Time</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lecture</td>
<td>Mon, Tue, Wed, Thu</td>
<td>2:00pm - 3:20pm</td>
<td>WLH 2205</td>
</tr>
<tr>
<td>Discussion Section</td>
<td>Fri</td>
<td>2:00pm - 4:50pm</td>
<td>WLH 2205</td>
</tr>
</tbody>
</table>

Discussion Sections: Students who cannot attend Friday's discussion or are after discussions still not sure about the solutions, are more than welcome to join Tutors and/or TAs weekly office hours. You can also setup separate office hours with the Tutors/TAs by appointment (Email or Piazza Private Message).

Contact Information and Office Hours

We will be communicating with you and making announcements through the online question and answer platform Piazza. We ask that when you have a question about the class that might be relevant to other students, you post your question on Piazza instead of emailing us. That way, everyone can benefit from the response.

<table>
<thead>
<tr>
<th>Name</th>
<th>Role</th>
<th>Email</th>
<th>Office Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oliver Braun</td>
<td>Instructor</td>
<td>olbraun (at) cs.ucsd.edu</td>
<td>Mon 3:30pm-4:30pm CSE 2136</td>
</tr>
<tr>
<td>Ping Yin</td>
<td>Teaching Assistant</td>
<td>piyin (at) ucsd.edu</td>
<td>Wed 4:30pm-5:30pm Atkinson Hall 4605</td>
</tr>
<tr>
<td>Bekhzo Slieev</td>
<td>Teaching Assistant</td>
<td>bslieev (at) ucsd.edu</td>
<td>Tue 5:00pm-6:00pm CSE Basement B240 A</td>
</tr>
<tr>
<td>Ashwin Ramesh</td>
<td>Tutor</td>
<td>a5ramesh (at) ucsd.edu</td>
<td>Thu 4:00pm-5:00pm CSE Basement</td>
</tr>
</tbody>
</table>
Grading

Exams will be given in class, with no make-ups allowed. You may not use calculators but can bring one page with hand-written notes. Please be sure to write your name on the back page. The notes must be submitted together with the exams. After your weighted average (Final Exam: 75%, Midterm: 25%) is calculated, letter grades will be assigned based on the following curved grading scale:

<table>
<thead>
<tr>
<th>Grade</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>A+</td>
<td>95</td>
</tr>
<tr>
<td>A</td>
<td>90</td>
</tr>
<tr>
<td>A-</td>
<td>85</td>
</tr>
<tr>
<td>B+</td>
<td>80</td>
</tr>
<tr>
<td>B</td>
<td>75</td>
</tr>
<tr>
<td>B-</td>
<td>70</td>
</tr>
<tr>
<td>C+</td>
<td>65</td>
</tr>
<tr>
<td>C</td>
<td>60</td>
</tr>
<tr>
<td>C-</td>
<td>55</td>
</tr>
</tbody>
</table>

In addition, you must pass the final exam with at least a 50% in order to pass the course. Requests for regrades should be made immediately by returning your exam on the day you get it back, before ever leaving the room with your exam.

Academic Integrity

The Jacobs School of Engineering code of Academic Integrity is here. You should make yourself aware of what is and is not acceptable by reading this document. Academic integrity violations will be taken seriously and reported immediately. Ignorance of the rules will not excuse you from any violations.

Accommodations

Students requesting accommodations for this course due to a disability must provide a current Authorization for Accommodation (AFA) letter issued by the Office for Students with Disabilities (OSD). If you have an AFA letter, please schedule an appointment with your instructor within the first three days of class to ensure that reasonable accommodations can be arranged. For more information, see here.
Textbook

The textbook for this course is S. Dasgupta, C. Papadimitriou and U. Vazirani: *Algorithms*, McGraw Hill, 2008. The textbook's companion website has extra practice problems and resources. Read online [here](http://cseweb.ucsd.edu/classes/su17_2/cse101-a/).

Note that the free online version has subtle differences from the hardcopy version.

You may also wish to look at the following literature as a supplementary resource:

1. [Course notes](http://cseweb.ucsd.edu/classes/su17_2/cse101-a/) from MIT's *Mathematics for Computer Science*

Course Material

NOTE: Slides, Extra Reading and Homework will be uploaded as the lecture proceeds.

<table>
<thead>
<tr>
<th>Slides</th>
<th>Extra Reading</th>
<th>Homework</th>
</tr>
</thead>
<tbody>
<tr>
<td>001-045.pdf</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Schedule**

NOTE: This schedule is subject to change.

<table>
<thead>
<tr>
<th>Week 1: 8/7/17 - 8/11/17</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Day</strong></td>
</tr>
<tr>
<td><strong>1. Algorithms</strong></td>
</tr>
<tr>
<td>Mon (L1)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Tue (L2)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Wed (L3)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>2. Assignment and Knapsack Problem</strong></td>
</tr>
<tr>
<td>Thu (L4)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Fri (T1)</td>
</tr>
</tbody>
</table>
## Week 2: 8/14/17 - 8/18/17

<table>
<thead>
<tr>
<th>Day</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td>Knapsack Problem: Greedy, Complete Enumeration, Branch-And-Bound</td>
</tr>
<tr>
<td>Tue</td>
<td>Algorithmic complexity and NP-hard problems Dynamic Programming and Backtracking</td>
</tr>
<tr>
<td>Wed</td>
<td>Complete Enumeration Approximation algorithms</td>
</tr>
<tr>
<td>Thu</td>
<td>Worst-case analysis</td>
</tr>
<tr>
<td>Fri</td>
<td>TA session</td>
</tr>
</tbody>
</table>

### 3. Bin Packing
Week 3: 8/21/17 - 8/25/17

<table>
<thead>
<tr>
<th>Day</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon (E1)</td>
<td>Midterm Exam 2:00pm-3:20am</td>
</tr>
<tr>
<td>Tue (L9)</td>
<td>Basics</td>
</tr>
<tr>
<td>Wed (L10)</td>
<td>Parallel processor scheduling</td>
</tr>
<tr>
<td>Wed (L11)</td>
<td>Single-processor scheduling</td>
</tr>
<tr>
<td>Fri (T3)</td>
<td>TA session</td>
</tr>
</tbody>
</table>

4. Scheduling
## Week 4: 8/28/17 - 9/1/17

<table>
<thead>
<tr>
<th>Day</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td>Labor Day</td>
</tr>
<tr>
<td>Tue</td>
<td>DFS, BFS</td>
</tr>
<tr>
<td>Wed</td>
<td>SSSP</td>
</tr>
<tr>
<td>Wed</td>
<td>SSSP, APSP</td>
</tr>
<tr>
<td>Thu</td>
<td>Mathematical Formulation of Optimization Problems</td>
</tr>
<tr>
<td>Fri</td>
<td>TA session</td>
</tr>
</tbody>
</table>

5. **Graph algorithms**

6. **Linear Programming**
<table>
<thead>
<tr>
<th>Day</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td>Labor Day</td>
</tr>
<tr>
<td>Tue (L16)</td>
<td>Simplex algorithm</td>
</tr>
<tr>
<td>Wed (L17)</td>
<td>Duality</td>
</tr>
<tr>
<td>Thu (T5)</td>
<td>TA session 2:00pm-3:20pm</td>
</tr>
<tr>
<td>Thu (T6)</td>
<td>TA session (Final preparation) 4:00pm-6:00pm</td>
</tr>
<tr>
<td>Fri (E2)</td>
<td>Final Exam 3:00pm-6:00pm</td>
</tr>
</tbody>
</table>
### Content

<table>
<thead>
<tr>
<th></th>
<th>Algorithms</th>
<th>Assignments/Knapsack</th>
<th>Bin Packing</th>
<th>Scheduling</th>
<th>Graph algorithms</th>
<th>Linear Programming</th>
<th>Heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Algorithms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td>Algorithmic problems</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.2</td>
<td>Analyzing iterative algorithms: Correctness and Time analysis (Minsort)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>Analyzing recursive algorithms: Correctness and Time analysis (Merging sorted arrays)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>Divide-And-Conquer (Mergesort)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Assignment and Knapsack problem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.1</td>
<td>Assignment problem: Greedy heuristic, Complete Enumeration, Branch-And-Bound</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.2</td>
<td>Knapsack problem: Greedy heuristic, Complete Enumeration, Branch-And-Bound</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.3</td>
<td>Algorithmic complexity and NP-hard problems: Combinatorial explosion, NP-hard problems, NP-Completeness, Optimization vs. Decision problems, P, NP and reducibility, Approximation algorithms, Heuristics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.4</td>
<td>Dynamic Programming and Backtracking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Bin Packing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.1</td>
<td>Complete Enumeration, First Fit, First Fit Decreasing, Next Fit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.2</td>
<td>Worst-case analysis of the approximation algorithms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Scheduling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.1</td>
<td>Basics and 3-fields-notation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.2</td>
<td>Parallel processor scheduling: McNaughton, Approximation algorithms, Worst-case analysis of approximation algorithms, Ron Graham</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.3</td>
<td>Single-processor scheduling: maximum lateness (EDD), number of delayed jobs (Moore), sum of delays (Complete Enumeration, Branch-and-Bound)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Graph algorithms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.1</td>
<td>Depth First Search, Breath First Search</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.2</td>
<td>SSSP: Bellman (acyclic networks), Dijkstra (non-negative networks), Arrays, Priority Queues (Binary Heaps), Bellman/Ford</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5.3</td>
<td>APSP: Floyd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Linear Programming</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.1</td>
<td>Simplex algorithm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.2</td>
<td>Duality</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Heuristics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.1</td>
<td>Simulated Annealing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.2</td>
<td>Tabu Search</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Algorithms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>--------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>1.1 Algorithmic problems</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.2 Analyzing iterative algorithms: Correctness and Time analysis (Minsort)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.3 Analyzing recursive algorithms: Correctness and Time analysis (Merging sorted arrays)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.4 Divide-And-Conquer (Mergesort)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Assignment and Knapsack problem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Bin Packing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Scheduling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Graph algorithms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Linear Programming</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Heuristics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Algorithmic problems

Remember: An algorithm is a method for solving a problem (on a computer).

Problem solving = “The Spirit of Computing”, driven by real-world necessity:

• Logistics
  • Scheduling: production planning, resource allocation, …
  • Bin Packing: storage on container ships, airline logistics, …
  • Shortest paths: warehouses, factory hall, distribution center, roads, …
• DNA Sequencing
  • Evolutionary Trees (edit distance, Steiner trees…)
  • Finding homologues, evolutionary significance (string-matching)
• Conformational Analysis
  • Drug Design (minimum-energy configuration)
• Autonomous Robotics, Vehicles
  • managing smart highways, collision avoidance / path planning, …
• Design of integrated circuits
  • placement, wiring, …
Algorithmic problems

- A problem is defined by:
  - input domain: integer \( n \)
  - output specification: find the factors of \( n \)

- A problem with the input specified \( n=258 \) is a **problem instance**

- An algorithm must be effective, i.e. give a correct answer and terminate.

- An algorithm should be efficient.
Algorithmic problems

- Types of Problems:
  - Decision: Yes or No answer - Does there exist a factor of n in the interval [2, n/4]?
  - Computation: How many different factorizations?
  - Construction: Construct (exhibit) a factorization, a factorization with only prime numbers, a factorization that has runtime less than x
  - Optimization: Determine the factorization with the minimum / maximum number of factors
    - Find the shortest path in a network
    - Find the schedule with minimum makespan

- Concepts
  - Upper bounds “at most this hard, at most this much effort”
  - Lower bounds “at least this hard, at least this much effort”
  - Reductions “solving this boils down to solving that”
  - Intractability “believed impossible to solve efficiently”
You are the operation manager in your company managing the re-organization of the container production. The containers have to be open on the top, must contain 10 cubic meters volume, and their length must be the double of the height (see diagram).

Material costs for the BASE PLATE are $10 per square meter, the SIDEWALLS cost $6 per square meter.

• Determine the optimal length and breath (width) of a container (in meters).
• How much does that minimal cost container cost?

Remember: A problem is defined by

• input domain / problem instance volume=10, c_Base=10, c_Side=6
• output specification h, b (in meters) such that the container has at least (?) 10m³ and the total costs are minimized
• An algorithm must be effective, i.e. give a correct answer and terminate.
• An algorithm should be efficient.
Real-world optimization problems and algorithms

(This slide is intentionally left empty for your notes.)
Solution

minimize \( c(h) = 24h^2 + \frac{160}{h} \), from 0 to 4

(Remark: limited to this domain just for display reasons)

\[
\min \left\{ 24h^2 + \frac{160}{h} \mid 0 \leq h \leq 4 \right\} = 24\sqrt{3} \cdot 10^{2/3} \text{ at } h = \sqrt[3]{\frac{10}{3}}
\]
## Content

<table>
<thead>
<tr>
<th></th>
<th>Algorithms</th>
<th>1.1 Algorithmic problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1.2 Analyzing iterative algorithms: Correctness and Time analysis (Minsort)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.3 Analyzing recursive algorithms: Correctness and Time analysis (Merging sorted arrays)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.4 Divide-And-Conquer (Mergesort)</td>
</tr>
<tr>
<td>2</td>
<td>Assignment and Knapsack problem</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Bin Packing</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Scheduling</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Graph algorithms</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Linear Programming</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Heuristics</td>
<td></td>
</tr>
</tbody>
</table>
Minsort
1. What problem are we solving?  
   problem specification  

2. How do we solve the problem?  
   algorithm description  

3. Why do these steps solve the problem?  
   proof of correctness  

4. When do we get an answer?  
   time analysis  

\[
29 25 3 49 9 37 21 43
\]

- **Problem Specification**

- We usually think of \( A \) as an integer array, but really \( A \) can contain any set of elements with an underlying (total) order.

- Remember, an algorithm must be well-defined, terminate, and produce the correct result.
Minsort

1. What problem are we solving?  
2. How do we solve the problem?  
3. Why do these steps solve the problem?  
4. When do we get an answer?

**MinSort  -- Selection Sort --**

Begin.
1. \textbf{For} k \leftarrow 1 \textbf{To} n-1
2. \texttt{min} \leftarrow A[k]
3. \texttt{index} \leftarrow k
4. \textbf{For} j \leftarrow k+1 \textbf{To} n
   \hspace{1em} \textbf{If} A[j] < \texttt{min} \textbf{Then}
   \hspace{2em} \texttt{min} \leftarrow A[j]
   \hspace{2em} \texttt{index} \leftarrow j
5. \texttt{A[index]} \leftarrow A[k]
6. \texttt{A[k]} \leftarrow \texttt{min}
End.
Minsort: Correctness

1. What problem are we solving?  
2. How do we solve the problem?  
3. Why do these steps solve the problem?  
4. When do we get an answer?

MinSort

Begin.
1. For k ← 1 To n-1 
2. min ← A[k]  
3. index ← k  
4. For j ← k+1 To n 
5. If A[j] < min Then 
6. min ← A[j]  
7. index ← j  
9. A[k] ← min  
End.

Loop invariant:

After the kth time through the outer loop, the first k positions A[1] through A[k] contain the k smallest array elements in order.

- How can we show that this loop invariant is true? → Induction on the number of times through the loop.
- Base case: k=0, before the loop.
- Induction hypothesis: Suppose the invariant holds after k-1 times through the loop.
- Inductive step: Show that the invariant holds after k times through the loop.
Weak induction

**Lemma:** The sum of all integers from 1 to n is equal to \( n(n+1)/2 \).

**Proof (by weak induction).**

**Basis step:**
\( n=1 \) is true (as the sum of the first \( n=1 \) numbers is 1).

**Inductive step:** (If the Lemma is true for \( n-1 \), then it is also true for \( n \))
Using the inductive hypothesis that the sum of all integers from 1 to \( n-1 \) is equal to \( (n-1)n/2 \), we can conclude:
\[
(n-1)n/2 + n = (n^2+n)/2 = n(n+1)/2.
\]
This completes the inductive step.
**Strong induction**

**Lemma:** Every amount of postage of 12 cents or more can be formed using just 4-cent and 5-cent stamps.

**Proof (by strong induction).**

**Basis step:**  
Inductive step (If the Lemma is true for n-4, then it is also true for n):

- \( P(12) = 4 + 4 + 4 \) \( P(16) + 4 \)
- \( P(13) = 4 + 4 + 5 \) \( P(17) + 4 \)
- \( P(14) = 4 + 5 + 5 \) \( P(18) + 4 \)
- \( P(15) = 5 + 5 + 5 \) \( P(19) + 4 \)

We use the inductive hypothesis that \( P(n-4) \) is true for any \( n \geq 16 \).

To form postage of \( n \) cents, we need only add another 4-cent stamp to the stamps we used to form postage of \( n-4 \) cents.

This completes the inductive step.
Minsort: Correctness

1. What problem are we solving? problem specification
2. How do we solve the problem? algorithm description
3. Why do these steps solve the problem? proof of correctness
4. When do we get an answer? time analysis

MinSort

Begin.
1. For k ← 1 To n-1
2. min ← A[k]
3. index ← k
4. For j ← k+1 To n
   If A[j] < min Then
   5. min ← A[j]
   6. index ← j
9. A[k] ← min
End.

Loop invariant:
After the $k^{th}$ time through the outer loop, the first $k$ positions $A[1]$ through $A[k]$ contain the $k$ smallest array elements in order

- How can we show that this loop invariant is true? → Induction on the number of times through the loop.
- **Base case:** $k=0$, before the loop.
- **Induction hypothesis:** Suppose the invariant holds after $k-1$ times through the loop.
- **Inductive step:** Show that the invariant holds after $k$ times through the loop.
MinSort: Correctness

After the $k^{th}$ time through the outer loop, the first $k$ positions $A[1]$ through $A[k]$ contain the $k$ smallest array elements in order.

$k=0$: (before we enter the for loop for the first time)


| big numbers (unsorted) |

$k-1 \rightarrow k$:


| small numbers (sorted) | big numbers (unsorted) |


| small numbers (sorted) | big numbers (unsorted) |
Minsort: Time analysis (1/2)

1. What problem are we solving?  
   problem specification
2. How do we solve the problem?  
   algorithm description
3. Why do these steps solve the problem?  
   proof of correctness
4. When do we get an answer?  
   time analysis

MinSort

Begin.
1. For k ← 1 To n-1
2. min ← A[k]
3. index ← k
4. For j ← k+1 To n
   If A[j] < min Then
   5. min ← A[j]
   6. index ← j
9. A[k] ← min
End.

Time analysis:
To determine how long an algorithm takes to sort an array, we typically measure the number of comparisons of array entries.

For each value of k, compare
(n-k) pairs of elements:
(n-1) + (n-2) + … + (1) =
n(n-1)/2
Minsort: Time analysis (2/2)

1. What problem are we solving?  
   problem specification
2. How do we solve the problem?  
   algorithm description
3. Why do these steps solve the problem?  
   proof of correctness
4. When do we get an answer?  
   time analysis

MinSort

Begin.
1. For k ← 1 To n-1
2. min ← A[k]
3. index ← k
4. For j ← k+1 To n
   If A[j] < min Then
   5. min ← A[j]
   6. index ← j
9. A[k] ← min
End.

Time analysis:
To determine how long an algorithm takes to sort an array, we typically measure the number of comparisons of array entries.

For MinSort, what is the maximum number of times we might have to compare the values of a pair of array elements?
- Worst Case
- Best Case
- Average Case
Running time analysis for Minsort

MinSort\((a_1, a_2, \ldots, a_n): \text{real numbers with } n \geq 2\)

\[
\text{for } i := 1 \text{ to } n-1 \\
\quad m := i \\
\quad \text{for } j := i+1 \text{ to } n \\
\qquad \text{if } (a_j < a_m) \text{ then } m := j \\
\text{interchange } a_i \text{ and } a_m \\
\]

\(\{a_1, \ldots, a_n \text{ is in increasing order}\}\)
Exercise: Which is true?

\[ f(n) = 4n^3 + 17n^2 + 46 \]

<table>
<thead>
<tr>
<th>O</th>
<th>( O(n^2) )</th>
<th>( O(n^3) )</th>
<th>( O(n^4) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( o )</td>
<td>( o(n^2) )</td>
<td>( o(n^3) )</td>
<td>( o(n^4) )</td>
</tr>
<tr>
<td>( \Theta )</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n^3) )</td>
<td>( \Theta(n^4) )</td>
</tr>
<tr>
<td>( \Omega )</td>
<td>( \Omega(n^2) )</td>
<td>( \Omega(n^3) )</td>
<td>( \Omega(n^4) )</td>
</tr>
<tr>
<td>( \omega )</td>
<td>( \omega(n^2) )</td>
<td>( \omega(n^3) )</td>
<td>( \omega(n^4) )</td>
</tr>
</tbody>
</table>
Running time analysis for Minsort: Upper Bound

MinSort(a₁, a₂, ..., aₙ: real numbers with n ≥ 2)

1. for i := 1 to n-1
   2. \( m := i \) \( O(1) \) Basic Operation
   3. for j := i+1 to n
      4. if \( a_j < a_m \) then \( m := j \) \( O(n-i) = O(n) \) Simple Loop
      5. interchange \( a_i \) and \( a_m \) \( O(1) \) Basic Operation

We work from the inside out, going from the body of the inside loop to the main algorithm.

The inner-most Line 4 is defined in terms of a fixed number of basic operations: a comparison, some logic, some variable writes. It is thus \( O(1) \).

Line 3 is a loop, with constant time line 4 inside. It repeats \( n-i \) times, so the total time is \( O(n-i) \). This ranges from constant time when \( i \) reaches \( n-1 \) to \( O(n) \) when \( i=1 \). So the worst-case is \( O(n) \).

Line 2 and 5 are constant time, so the body of the FOR loop in line 1 takes \( O(1+n+1) = O(n) \) total.

Finally, line 1 is a loop whose body is \( O(n) \) and gets repeated \( n-1 < n \) times. So the whole algorithm is \( O(n^2) \).
Running time analysis for Minsort: Lower Bound

MinSort(a_1, a_2, ..., a_n: real numbers with n >=2 )

1 for i := 1 to n-1
2 m := i
3 for j:= i+1 to n
4 if ( a_j < a_m ) then m := j
5 interchange a_i and a_m

O is an upper bound, not always tight. We can ask: is the running time also lower bounded by a quadratic, or is there a smaller upper bound? We don't need to find the “worst-case input” or give an exact formula to answer this question, just show that sometimes the algorithm performs at least on the order of n^2 operations of some kind.

Look at the first n/2 times we run the loop in line 3. Then i ≤ n/2, so n-i ≥ n/2.

Thus, we run it at least n/2 x n/2 = n^2/4 times total. This is Ω(n^2).

Thus, the time is both O(n^2) and Ω(n^2), so our analysis is tight, and the time is Φ(n^2).

So in this example, our first analysis is the best possible.
### Bubble Sort

**Idea:** Compare the first two numbers, and if the first is bigger, keep comparing it to the next number in the array until we find a larger one. Repeat until the array is sorted.

```
Begin.
1. For i ← 1 To n-1
2.   For j ← 1 To n-i
End.
```

```
29 25 3 49 9 37 21 43
```
Bubble Sort

(this page is intentionally left empty for your notes)
**Insertion Sort**

**Idea:** Take an element of A and find where it belongs relative to the elements before it. Shift everything back to make room and put the element in its proper place. Now that this element has been inserted where it belongs, do the same for the next element of A.

```
Begin.
1. for j := 2 to n
   i := 1
2. while a_j > a_i
   i := i+1
3. m := a_j
4. for k := 0 to j-i-1
   a_j-k := a_j-k-1
5. a_i := m
End.
```

29 25 3 49 9 37 21 43
Insertion Sort

(this page is intentionally left empty for your notes)
## Content

<table>
<thead>
<tr>
<th></th>
<th>Algorithms</th>
<th>1.1 Algorithmic problems</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1.2 Analyzing iterative algorithms: Correctness and Time analysis (Minsort)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.3 Analyzing recursive algorithms: Correctness and Time analysis (Merging sorted arrays)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.4 Divide-And-Conquer (Mergesort)</td>
</tr>
<tr>
<td></td>
<td>Assignment and Knapsack problem</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bin Packing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scheduling</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Graph algorithms</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Linear Programming</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Heuristics</td>
<td></td>
</tr>
</tbody>
</table>
Merging sorted arrays
Merging sorted arrays

In the merge problem, we are given two sorted arrays $A[1..n]$ and $B[1..m]$ and want to produce a sorted array containing the union of both lists. While this is interesting in its own right, it will also be a key sub-procedure in the recursive sorting algorithm MergeSort.

\[
A = \begin{bmatrix} 2 & 7 & 9 & 11 \end{bmatrix} \quad B = \begin{bmatrix} 6 & 12 & 13 \end{bmatrix}
\]

\[
C = \begin{bmatrix} 2 & 6 & 7 & 9 & 11 & 12 & 13 \end{bmatrix}
\]
Recursive Merge

**Definition**

Let $v \circ C[1, \ldots, m]$ denote an array of length $m + 1$ whose first element is $v$ and the rest is $C[1, \ldots, m]$.

$$RMerge(A[1, \ldots, k], B[1, \ldots, \ell]: \text{sorted arrays})$$

1. IF $k = 0$ return $B[1, \ldots, \ell]$
2. IF $\ell = 0$ return $A[1, \ldots, k]$
4. ELSE return $B[1] \circ RMerge(A[1, \ldots, k], B[2, \ldots, \ell])$
Recursive Merge: Correctness

\[
R\text{Merge}(A[1, \ldots, k], B[1, \ldots, \ell]): \text{ sorted arrays } \\
1. \text{IF } k = 0 \text{ return } B[1, \ldots, \ell] \\
2. \text{IF } \ell = 0 \text{ return } A[1, \ldots, k] \\
\quad A[1] \circ R\text{Merge}(A[2, \ldots, k], B[1, \ldots, \ell]) \\
4. \text{ELSE return } B[1] \circ R\text{Merge}(A[1, \ldots, k], B[2, \ldots, \ell])
\]

We want to show that \( R\text{Merge}(A[1, \ldots, k], B[1, \ldots, \ell]) \) is a sorted array containing all elements from either array. We'll prove this by induction on \( n = k + \ell \), the total input size. (left as an Exercise)
(this page is intentionally left empty for your notes)
Recursive Merge: Time analysis

\[ R\text{Merge}(A[1, \ldots, k], B[1, \ldots, \ell]): \text{sorted arrays} \]

1. IF \( k = 0 \) return \( B[1, \ldots, \ell] \)
2. IF \( \ell = 0 \) return \( A[1, \ldots, k] \)
4. ELSE return \( B[1] \circ R\text{Merge}(A[1, \ldots, k], B[2, \ldots, \ell]) \)

Every step is constant time, except that we make one recursive call in either line 3 or line 4. Thus,

\[ T(1) = c \text{ for some constant } c \]
\[ T(n) = T(n-1) + c' \text{ for some constant } c' \cdot \]

\[ \Rightarrow O(n) \]
## Content

<table>
<thead>
<tr>
<th></th>
<th>Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.1 Algorithmic problems</td>
</tr>
<tr>
<td></td>
<td>1.2 Analyzing iterative algorithms: Correctness and Time analysis (Minsort)</td>
</tr>
<tr>
<td></td>
<td>1.3 Analyzing recursive algorithms: Correctness and Time analysis (Merging sorted arrays)</td>
</tr>
<tr>
<td></td>
<td>1.4 Divide-And-Conquer (Mergesort)</td>
</tr>
<tr>
<td>2</td>
<td>Assignment and Knapsack problem</td>
</tr>
<tr>
<td>3</td>
<td>Bin Packing</td>
</tr>
<tr>
<td>4</td>
<td>Scheduling</td>
</tr>
<tr>
<td>5</td>
<td>Graph algorithms</td>
</tr>
<tr>
<td>6</td>
<td>Linear Programming</td>
</tr>
<tr>
<td>7</td>
<td>Heuristics</td>
</tr>
</tbody>
</table>
Divide-And-Conquer: Basic Idea

Divide-and-conquer is a form of recursive strategy for designing algorithms.

1. **Divide** an instance into several smaller instances of the same problem.
2. **Recursively solve** each smaller instance.
3. **Conquer** by combining the solutions into the solution for the original instance.

Because Divide-And-Conquer creates at least two subproblems, a Divide-And-Conquer algorithm makes multiple recursive calls.
Mergesort
Divide-And-Conquer: Mergesort

11, 9, 7, 2, 13, 12, 6

11, 9, 7, 2

11, 9

9

divide

divide

conquer

9, 11

2, 7

combine

combine

11, 9, 7, 2

7, 2

2, 7

13, 12, 6

13, 12

12, 13

6

divide

divide

divide

divide

divide

divide

divide

divide

combine

combine

combine

combine

combine

combine

combine

combine

2, 7, 9, 11

2, 6, 7, 9, 11, 12, 13

2, 6, 7, 9, 11, 12, 13

2, 6, 7, 9, 11, 12, 13

2, 6, 7, 9, 11, 12, 13

2, 6, 7, 9, 11, 12, 13

2, 6, 7, 9, 11, 12, 13

2, 6, 7, 9, 11, 12, 13

2, 6, 7, 9, 11, 12, 13

2, 6, 7, 9, 11, 12, 13

2, 6, 7, 9, 11, 12, 13

2, 6, 7, 9, 11, 12, 13

2, 6, 7, 9, 11, 12, 13

2, 6, 7, 9, 11, 12, 13
Mergesort

\[ \text{MergeSort}(A[1, \ldots, n]) \]

1. IF \( n = 1 \) Return \( A \)
2. \( B[1, \ldots, n/2] \leftarrow \text{MergeSort}(A[1, \ldots, n/2]) \)
3. \( C[1, \ldots, n/2] \leftarrow \text{MergeSort}(A[n/2 + 1, \ldots, n]) \)
4. Return Merge(\( B[1, \ldots, n/2], C[1, \ldots, n/2] \))
Mergesort: Correctness

For Mergesort to be correct, it should return a sorted array, and that array should contain exactly the elements A[1],…,A[n].

Prove (strong induction on n).

(Remember:)

- In strong induction, you assume that the statement you want to show holds for all integers \( n' \) with \( k \leq n' \leq n \).
- Then you must show that under this inductive hypothesis your statement is also true for \( n \).
- We use strong induction whenever a recursive algorithm acting on an input of size \( n \) makes calls with inputs of size other than \( n-1 \).

(leave as an Exercise)
(this page is intentionally left empty for your notes)
Mergesort: Time analysis

MergeSort(A[1, ..., n])

1. IF n = 1 Return A
2. B[1, ..., n/2] ← MergeSort(A[1, ..., n/2])
3. C[1, ..., n/2] ← MergeSort(A[n/2 + 1, ..., n])
4. Return Merge(B[1, ..., n/2], C[1, ..., n/2])

\[
T(1) = c' \\
T(n) = 2 \cdot T \left( \frac{n}{2} \right) + cn \\
= 2^2 \cdot T \left( \frac{n}{2^2} \right) + 2 \cdot cn \\
= \ldots \\
= 2^k \cdot T \left( \frac{n}{2^k} \right) + k \cdot cn \\
= n \cdot T(1) + \log n \cdot c \cdot n \\
\in O(n \log n)
\]

\[\frac{n}{2^k} = 1\]
\[n = 2^k\]
\[k = \log_2 n\]