CSE 101
SS 2, 2018

Oliver Braun

Thanks to Andrew B. Kahng, Miles Jones, Russell Impagliazzo, Mohan Paturi.
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
- Greedy 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 (Minsert), Analyzing recursive algorithms: Correctness and Time analysis (Merging sorted arrays), Divide-And-Conquer (Mergesort)
2. Assignment and Knapsack Problem: Greedy, Complete Enumeration, Branch-And-Bound, Dynamic Programming, Backtracking
4. Bin Packing: Complete Enumeration, FirstFit, FirstFitDecreasing, Next Fit, Approximation algorithms and Worst-case analysis
5. 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)
6. Graph algorithms and Shortest Paths: DFS, BFS, Bellman (acyclic networks), Dijkstra (non-negative networks, PQ: Array, Binary Heap), Bellman/Ford
7. Linear Programming: Simplex

Class Meeting

<table>
<thead>
<tr>
<th></th>
<th>Weekday</th>
<th>Time</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lectures</td>
<td>Tue, Thu</td>
<td>8:00am - 10:50am</td>
<td>WLH 2206</td>
</tr>
<tr>
<td>Discussions</td>
<td>Fri</td>
<td>8:00am - 9:50am</td>
<td>WLH 2115</td>
</tr>
</tbody>
</table>

**Discussions:** Students who cannot attend Friday’s discussion or are after discussions still not sure about the solutions, 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.
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 2:30pm-3:30pm CSE 2138</td>
</tr>
<tr>
<td>Siyang Wang</td>
<td>Teaching Assistant</td>
<td>silw030 (at) eng.ucsd.edu</td>
<td>Tue 12.00pm-1.00pm CSE B240A</td>
</tr>
<tr>
<td>Jatin Agrawal</td>
<td>Teaching Assistant</td>
<td>j1agrawa (at) ucsd.edu</td>
<td>Thu 12.00pm-1.00pm CSE B260A</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: 180/260, Midterm: 80/260) is calculated, letter grades will be assigned based on the following curved grading scale:

<table>
<thead>
<tr>
<th>Grade</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>A+</td>
<td>97</td>
</tr>
<tr>
<td>A</td>
<td>93</td>
</tr>
<tr>
<td>A-</td>
<td>89</td>
</tr>
<tr>
<td>B+</td>
<td>85</td>
</tr>
<tr>
<td>B</td>
<td>81</td>
</tr>
<tr>
<td>B-</td>
<td>77</td>
</tr>
<tr>
<td>C+</td>
<td>73</td>
</tr>
<tr>
<td>C</td>
<td>69</td>
</tr>
<tr>
<td>C-</td>
<td>65</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 (CSD). 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


You may also wish to look at the following literature as a supplementary resource:
1. Course notes from MIT's Mathematics for Computer Science
5. The Art and Craft of Problem Solving, P. Zeitz, John Wiley & Sons, 2007, Read online
6. Problems on Algorithms, I. Parberry, Download (donation based)
## 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>Exams</th>
<th>Homework</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSE101-001-100 pdf</td>
<td></td>
<td>2017 Midterm</td>
<td>Hw 1 (Q)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2017 Final</td>
<td>Hw 1 (S)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Hw 2 (Q)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Hw 2 (S)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Hw 3 (Q)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Hw 3 (S)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Hw 4 (Q)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Hw 4 (S)</td>
</tr>
</tbody>
</table>
### Schedule

NOTE: This schedule is subject to change.

#### Week 1: 8/7/17 - 8/11/17

<table>
<thead>
<tr>
<th>Day</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Algorithms</strong></td>
<td></td>
</tr>
<tr>
<td>Tue</td>
<td>Algorithmic problems</td>
</tr>
<tr>
<td></td>
<td>Problem solving</td>
</tr>
<tr>
<td>Thu</td>
<td>Analyzing iterative and recursive algorithms</td>
</tr>
<tr>
<td></td>
<td>Divide-And-Conquer</td>
</tr>
<tr>
<td>Fri</td>
<td>Discussion</td>
</tr>
<tr>
<td></td>
<td>Two pointers, Counting inversions, Quicksort</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><strong>2. Assignment and Knapsack Problem</strong></td>
<td></td>
</tr>
<tr>
<td>Tue</td>
<td>Assignment: Greedy, Complete Enumeration, Branch-And-Bound</td>
</tr>
<tr>
<td></td>
<td>Knapsack: Greedy, Complete Enumeration, Branch-And-Bound</td>
</tr>
<tr>
<td><strong>3. Algorithmic complexity and NP-hard problems</strong></td>
<td></td>
</tr>
<tr>
<td>Thu</td>
<td>Combinatorial explosion, NP-hard problems, NP-completeness, Optimization vs. Decision problems, P, NP and reducibility, Approximation algorithms, Heuristics</td>
</tr>
<tr>
<td>Fri</td>
<td>Discussion</td>
</tr>
<tr>
<td></td>
<td>Dynamic Programming</td>
</tr>
</tbody>
</table>
### Week 3: 8/21/17 - 8/25/17

<table>
<thead>
<tr>
<th>Day (E1)</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tue</td>
<td>Midterm Exam 8:00am-9:20am</td>
</tr>
</tbody>
</table>

#### 3. Bin Packing

<table>
<thead>
<tr>
<th>Day (L9)</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tue</td>
<td>Complete Enumeration, Approximation algorithms, Worst-case analysis</td>
</tr>
</tbody>
</table>

#### 4. Scheduling

<table>
<thead>
<tr>
<th>Day (L10-L11)</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thu</td>
<td>Parallel processor scheduling (with and without preemptions) Worst-case (competitive) analysis of approximation algorithms (Ron Graham's List Scheduling and LPT)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Day (D3)</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fri</td>
<td>Discussion</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Day (L12-L13)</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tue</td>
<td>Flow-Shop scheduling and Single-processor scheduling (Greedy Heuristics and Branch-And-Bound)</td>
</tr>
</tbody>
</table>

#### 5. Graph algorithms

<table>
<thead>
<tr>
<th>Day (L14-L15)</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wed</td>
<td>DFS, BFS, Minimum Spanning Tree, Topological Sorting Single-source-shortest-paths: acyclic networks, non-negative networks All-pairs-shortest-paths</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Day (D4)</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fri</td>
<td>Discussion</td>
</tr>
</tbody>
</table>
### Week 5: 9/4/17 - 9/8/17

<table>
<thead>
<tr>
<th>Day</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tue (L16-L17)</td>
<td>Solving graph problems</td>
</tr>
</tbody>
</table>
| Thu (L18-L19)  | **6. Linear Programming**  
Simplex algorithm, Interpretation of dual values |
| Fri (D5)    | Discussion and Final preparation             |
| Sat (E2)    | Final Exam  
8:00am-11:00am                              |
| 1 Algorithms | 1.1 Algorithmic problems  
1.2 Analyzing iterative algorithms: Correctness and Time analysis (Minsort)  
1.3 Analyzing recursive algorithms: Correctness and Time analysis (Merging sorted arrays)  
1.4 Divide-And-Conquer (Mergesort) |
|---|---|
| 2 Assignment and Knapsack problem | 2.1 Assignment problem: Greedy heuristic, Complete Enumeration, Branch-And-Bound  
2.2 Knapsack problem: Greedy heuristic, Complete Enumeration, Branch-And-Bound  
2.3 Algorithmic complexity and NP-hard problems: Combinatorial explosion, NP-hard problems, NP-Completeness, Optimization vs. Decision problems, P, NP and reducibility, Approximation algorithms, Heuristics  
2.4 Dynamic Programming and Backtracking |
| 3 Bin Packing | 3.1 Complete Enumeration, First Fit, First Fit Decreasing, Next Fit  
3.2 Worst-case analysis of the approximation algorithms |
| 4 Scheduling | 4.1 Basics and 3-fields-notation  
4.2 Parallel processor scheduling: McNaughton, Approximation algorithms, Worst-case analysis of approximation algorithms, Ron Graham  
4.3 Single-processor scheduling: maximum lateness (EDD), number of delayed jobs (Moore), sum of delays (Complete Enumeration, Branch-and-Bound) |
| 5 Graph algorithms | 5.1 Depth First Search, Breath First Search  
5.2 SSSP: Bellman (acyclic networks), Dijkstra (non-negative networks), Arrays, Priority Queues (Binary Heaps), Bellman/Ford  
5.3 APSP: Floyd |
| 6 Linear Programming | 6.1 Simplex algorithm  
6.2 Duality |
| 7 Heuristics | 7.1 Simulated Annealing (SA)  
7.2 Variable-Neighborhood-Search (VNS) |
<table>
<thead>
<tr>
<th></th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Algorithms</td>
</tr>
<tr>
<td>1.1</td>
<td>Algorithmic problems</td>
</tr>
<tr>
<td>1.2</td>
<td>Analyzing iterative algorithms: Correctness and Time analysis (Minsort)</td>
</tr>
<tr>
<td>1.3</td>
<td>Analyzing recursive algorithms: Correctness and Time analysis (Merging sorted arrays)</td>
</tr>
<tr>
<td>1.4</td>
<td>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>
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

- In this course, we will solve optimization problems:
  - **Optimization:** Determine the factorization with the minimum / maximum number of factors
    - Find the shortest path in a network
    - Find the schedule with minimum makespan
    - Find the minimum possible number of bins
    - Find the production program that leads to minimum costs

- Other 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
Algorithmic problems

You will learn about the following important concepts in Computer Science:

- **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”
Can you solve the following problem?
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 at least 10.00 cubic meters volume, and their length must be the double of the height (see diagram).

Material costs for the BASE PLATE are $10.00 per square meter, the SIDEWALLS cost $6.00 per square meter.

- Determine the **optimal** length and breath (width) of a container (rounded to meters and centimeters, e.g. 1.25m).
- How much does that minimal cost container cost?

Remember: A problem is defined by

- input domain / problem instance \( \text{volume}=10, \ c_{\text{Base}}=10, \ c_{\text{Side}}=6 \)
- output specification \( h, b \) (in meters) such that the container has at least 10.00m\(^3\) and the total costs are minimized

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

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

Remark: limited to this domain just for display reasons.

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

### Analytical Solution:

<table>
<thead>
<tr>
<th>h</th>
<th>b = 5/(h^2)</th>
<th>Vol = 2(h^2)b</th>
<th>c(h, b)</th>
<th>Actual Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.49</td>
<td>2.25215</td>
<td>10,00000</td>
<td>160,66495 €</td>
<td></td>
</tr>
</tbody>
</table>
### Container example

**Analytical Solution:**

- \( h = 1.49 \)
- \( b = 2.25215 \)
- \( \text{Vol} = 2(h^2)b \)
- \( c(h,b) \)
- \( 10,00000 \)
- \( 160,66495 \text{ €} \)

**Practical Solution:**

<table>
<thead>
<tr>
<th>( h )</th>
<th>( b = 5/(h^2) )</th>
<th>( b )</th>
<th>( \text{Vol} = 2(h^2)b )</th>
<th>( c(h,b) )</th>
<th>( \text{Cost} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.38</td>
<td>2.62550</td>
<td>2.63</td>
<td>10.02</td>
<td>161.85 €</td>
<td></td>
</tr>
<tr>
<td>1.39</td>
<td>2.58786</td>
<td>2.59</td>
<td>10.01</td>
<td>161.57 €</td>
<td></td>
</tr>
<tr>
<td>1.40</td>
<td>2.55102</td>
<td>2.56</td>
<td>10.04</td>
<td>161.73 €</td>
<td></td>
</tr>
<tr>
<td>1.41</td>
<td>2.51496</td>
<td>2.52</td>
<td>10.02</td>
<td>161.42 €</td>
<td></td>
</tr>
<tr>
<td>1.42</td>
<td>2.47967</td>
<td>2.48</td>
<td>10.00</td>
<td>161.08 €</td>
<td></td>
</tr>
<tr>
<td>1.43</td>
<td>2.44511</td>
<td>2.45</td>
<td>10.02</td>
<td>161.19 €</td>
<td></td>
</tr>
<tr>
<td>1.44</td>
<td>2.41127</td>
<td>2.42</td>
<td>10.04</td>
<td>161.28 €</td>
<td></td>
</tr>
<tr>
<td>1.45</td>
<td>2.37812</td>
<td>2.38</td>
<td>10.01</td>
<td>160.89 €</td>
<td></td>
</tr>
<tr>
<td>1.46</td>
<td>2.34566</td>
<td>2.35</td>
<td>10.02</td>
<td>160.95 €</td>
<td></td>
</tr>
<tr>
<td>1.47</td>
<td>2.31385</td>
<td>2.32</td>
<td>10.03</td>
<td>160.99 €</td>
<td></td>
</tr>
<tr>
<td>1.48</td>
<td>2.28269</td>
<td>2.29</td>
<td>10.03</td>
<td>161.02 €</td>
<td></td>
</tr>
<tr>
<td>1.49</td>
<td>2.25215</td>
<td>2.26</td>
<td>10.03</td>
<td>161.04 €</td>
<td></td>
</tr>
<tr>
<td>1.50</td>
<td>2.22222</td>
<td>2.23</td>
<td>10.04</td>
<td>161.04 €</td>
<td></td>
</tr>
<tr>
<td>1.51</td>
<td>2.19289</td>
<td>2.20</td>
<td>10.03</td>
<td>161.03 €</td>
<td></td>
</tr>
<tr>
<td>1.52</td>
<td>2.16413</td>
<td>2.17</td>
<td>10.03</td>
<td>161.00 €</td>
<td></td>
</tr>
<tr>
<td>1.53</td>
<td>2.13593</td>
<td>2.14</td>
<td>10.02</td>
<td>160.96 €</td>
<td></td>
</tr>
<tr>
<td>1.54</td>
<td>2.10828</td>
<td>2.11</td>
<td>10.01</td>
<td>160.90 €</td>
<td></td>
</tr>
<tr>
<td>1.55</td>
<td>2.08117</td>
<td>2.09</td>
<td>10.04</td>
<td>161.32 €</td>
<td></td>
</tr>
<tr>
<td>1.56</td>
<td>2.05457</td>
<td>2.06</td>
<td>10.03</td>
<td>161.24 €</td>
<td></td>
</tr>
<tr>
<td>1.57</td>
<td>2.02848</td>
<td>2.03</td>
<td>10.01</td>
<td>161.14 €</td>
<td></td>
</tr>
<tr>
<td>1.58</td>
<td>2.00288</td>
<td>2.01</td>
<td>10.04</td>
<td>161.54 €</td>
<td></td>
</tr>
<tr>
<td>1.59</td>
<td>1.97777</td>
<td>1.98</td>
<td>10.01</td>
<td>161.42 €</td>
<td></td>
</tr>
<tr>
<td>1.60</td>
<td>1.95313</td>
<td>1.96</td>
<td>10.04</td>
<td>161.79 €</td>
<td></td>
</tr>
</tbody>
</table>

**Continuous vs. Discrete Optimization**
<table>
<thead>
<tr>
<th></th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Algorithms</td>
</tr>
<tr>
<td></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>
Min Sort
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**

---

\[
\begin{array}{cccccccc}
29 & 25 & 3 & 49 & 9 & 37 & 21 & 43 \\
\end{array}
\]

- **Problem Specification**
  Sorting: Given an array
  rearrange the values in A so that
  \[ A[1], A[2], A[3], \ldots, A[n], \]

- 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.
Min Sort

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?  

**Min Sort**

|--------|---------------------|---------------|--------------|---------------------|---------------------|---------------|--------------|---------------|--------------|

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>29</td>
<td>3</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>k=1</th>
<th>3</th>
<th>29</th>
<th>49</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>k=2</td>
<td>3</td>
<td>25</td>
<td>49</td>
<td>29</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>k=3</td>
<td>3</td>
<td>25</td>
<td>29</td>
<td>49</td>
</tr>
</tbody>
</table>

Find the minimum of the elements at positions k..n (n-k+1 numbers) together with its position.
Min Sort

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

Min Sort

```
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.
```

```
| k=1 | 3 | 29 | 49 | 25 |
| k=2 | 3 | 25 | 49 | 29 |
| k=3 | 3 | 25 | 29 | 49 |
```

Swap the elements at positions index (where the min is) and k.
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

Min Sort

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
   min ← A[j]
   index ← j
6. 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? \( \rightarrow \) 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 \( \frac{n(n+1)}{2} \).

**Proof:**

We prove this using (weak) induction on \( n \).

**Base case:** \( n = 1 \). \[ \sum_{k=1}^{1} k = 1 = \frac{1(1+1)}{2} \]

**Induction hypothesis:** Suppose that the Lemma is true for \( n - 1 \):

\[
\sum_{k=1}^{n-1} k = \frac{(n-1)n}{2}
\]

**Induction step:**

\[
\sum_{k=1}^{n} k = \left( \sum_{k=1}^{n-1} k \right) + \left( n \right)
\]

By the induction hypothesis we get:

\[
\left( \frac{(n-1)n}{2} \right) + \left( n \right)
\]

After combining terms we have:

\[
\frac{n^2 + n}{2} = \frac{n(n+1)}{2}
\]
Lemma: Every amount of postage of 12 cents or more can be formed using just 4-cent and 5-cent stamps.

Proof: We prove this using (strong) induction on any postage of \( n \geq 12 \) cents. Let \( P(n) \) be the statement that that a postage of \( n \) cents can be formed using only 4-cent and 5-cent stamps.

Base cases:
\( P(12) \): Use three 4-cent stamps.
\( P(13) \): Use two 4-cent stamps and one 5-cent stamp.
\( P(14) \): Use one 4-cent stamp and two 5-cent stamps.
\( P(15) \): Use three 5-cent stamps.

Induction hypothesis: Suppose \( P(n) \) is true for \( n \geq 12 \), i.e. that we can form postage of \( k \) cents where \( 12 \leq k \leq n \).

Induction step: To form postage of \( n+1 \) cents, use the stamps to form postage of \( n - 3 \) cents together with a 4-cent stamp.
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?

Min Sort

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
      min ← A[j]
      index ← j
9. A[k] ← min
End.

Loop invariant:

After the k\text{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 for-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)

$k-1 \Rightarrow k$:

We distinguish between small numbers (all sorted) and big numbers (not sorted). All big numbers are $\geq$ to all small numbers.

Loop Invariant: After the $k$th iteration through the outer for-loop, the first $k$ positions $A[1]$ through $A[k]$ contain the $k$ smallest array elements in sorted order.

Induction Base: $k=0$. Before we enter the outer for-loop for the first time, all numbers are big numbers, i.e. unsorted.

Induction Hypothesis: Same as Loop Invariant for $k-1$.

Induction Step:

- Lines 2..7: During the $k$-th run through the outer for-loop, the smallest number out of the $n-k+1$ big numbers (lets call it $x$) is selected.
- Lines 8..9: This number is moved to the front of the big numbers, i.e. to position $A[k]$ ($x$ is swapped with the number at position $A[k]$).
- Because $x$ is the smallest of the big numbers, $x$ is obviously $\leq$ to all big numbers.
- As all big numbers are $\geq$ all small numbers, $x$ is obviously $\geq$ all small numbers.

Result:

- The set of small numbers has been increased by 1 ($A[k]=x$) and the first $k$ numbers $A[1..k]$ are in sorted order.
- The set of big numbers has been decreased by 1.
Min Sort: 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

Min Sort

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.

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
Min Sort: 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

Min Sort

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 Min Sort, 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 Min Sort

MinSort(a₁, a₂, ..., aₙ: real numbers with n >=2 )
for i := 1 to n-1
  m := i
  for j:= i+1 to n
    if ( aⱼ < aₘ ) then m := j
    interchange aᵢ and aₘ

{a₁, ..., aₙ is in increasing order}
Exercise: Which is true?

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

<table>
<thead>
<tr>
<th>( O )</th>
<th>( \leq )</th>
<th>( O(n^2) )</th>
<th>( O(n^3) )</th>
<th>( O(n^4) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( o )</td>
<td>( &lt; )</td>
<td>( o(n^2) )</td>
<td>( o(n^3) )</td>
<td>( o(n^4) )</td>
</tr>
<tr>
<td>( \Theta )</td>
<td>( = )</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n^3) )</td>
<td>( \Theta(n^4) )</td>
</tr>
<tr>
<td>( \Omega )</td>
<td>( \geq )</td>
<td>( \Omega(n^2) )</td>
<td>( \Omega(n^3) )</td>
<td>( \Omega(n^4) )</td>
</tr>
<tr>
<td>( \omega )</td>
<td>( &gt; )</td>
<td>( \omega(n^2) )</td>
<td>( \omega(n^3) )</td>
<td>( \omega(n^4) )</td>
</tr>
</tbody>
</table>
# Running time analysis for Min Sort: Upper Bound

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\textbf{for } i := 1 \textbf{ to } n-1</td>
<td>(O(1)) Basic Operation</td>
</tr>
<tr>
<td>2</td>
<td>\hspace{1cm} m := i</td>
<td>(O(1)) Basic Operation</td>
</tr>
<tr>
<td>3</td>
<td>\hspace{1cm} \textbf{for } j := i+1 \textbf{ to } n</td>
<td>Simple Loop</td>
</tr>
<tr>
<td>4</td>
<td>\hspace{1cm} \textbf{if } (a_j &lt; a_m) \textbf{ then } m := j</td>
<td>(O(n-i) = O(n))</td>
</tr>
<tr>
<td>5</td>
<td>\hspace{1cm} \textbf{interchange } a_i \textbf{ and } a_m</td>
<td>(O(1)) Basic Operation</td>
</tr>
</tbody>
</table>

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 Min Sort: Lower 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 Simple Loop
4 if (aⱼ < aₘ) then m := j O(n-i) = O(n)
5 interchange aᵢ and aₘ O(1) Basic Operation


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 n² 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 × n/2 = n²/4 times total. This is Ω(n²).

Thus, the time is both O(n²) and Ω(n²), so our analysis is tight, and the time is Φ(n²).

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 $i \leftarrow 1$ To $n-1$
2. For $j \leftarrow 1$ To $n-i$
**End.**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>49</td>
<td>29</td>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>49</td>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>3</td>
<td>49</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>3</td>
<td>25</td>
<td>49</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>29</td>
<td>3</td>
<td>25</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>29</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>25</td>
<td>29</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>25</td>
<td>29</td>
<td></td>
</tr>
</tbody>
</table>

**Exercise:**
- Find a loop invariant that implies correctness.
- Prove the loop invariant.
**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.
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>29</td>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td>j=2</td>
<td>29</td>
<td>49</td>
<td>3</td>
</tr>
<tr>
<td>j=3</td>
<td>3</td>
<td>29</td>
<td>49</td>
</tr>
<tr>
<td>j=4</td>
<td>3</td>
<td>25</td>
<td>29</td>
</tr>
</tbody>
</table>

**Exercise:**
- Find a loop invariant that implies correctness.
- Prove the loop invariant.
<table>
<thead>
<tr>
<th></th>
<th>Algorithms</th>
<th>Assignment and Knapsack problem</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>1.1 Algorithmic problems</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 (Mergesort)</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>
</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 \( \nu \circ C[1, \ldots, m] \) denote an array of length \( m + 1 \) whose first element is \( \nu \) and the rest is \( C[1, \ldots, m] \).

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

1. If \( k = 0 \) return \( B[1, \ldots, \ell] \)
2. If \( \ell = 0 \) return \( A[1, \ldots, k] \)
   \[
   A[1] \circ R\text{Merge}(A[2, \ldots, k], B[1, \ldots, \ell])
   \]
4. Else return \( B[1] \circ R\text{Merge}(A[1, \ldots, k], B[2, \ldots, \ell]) \)
Recursive Merge: Correctness

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

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

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)
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 \( A[1, \ldots, k] \)
   \[
   A[1] \circ R\text{Merge}(A[2, \ldots, k], B[1, \ldots, \ell])
   \]
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,

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

\[\Rightarrow O(n)\]
| 1 Algorithms | 1.1 Algorithmic problems |
| | 1.2 Analyzing iterative algorithms: Correctness and Time analysis (Minsort) |
| | 1.3 Analyzing recursive algorithms: Correctness and Time analysis (Merging sorted arrays) |
| | 1.4 Divide-And-Conquer (Mergesort) |
| 2 Assignment and Knapsack problem |
| 3 Bin Packing |
| 4 Scheduling |
| 5 Graph algorithms |
| 6 Linear Programming |
| 7 Heuristics |
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.
Divide-And-Conquer: Merge Sort

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

11, 9, 7, 2

11, 9

9, 11

2, 7, 9, 11

2, 7

7, 2

13, 12

13, 12

12, 13

12, 13

6

6

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

divide

divide

divide

divide

divide

divide

divide

divide

conquer

combine

combine

combine

combine

combine

combine

combine

combine
Merge Sort

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

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 \( \text{Merge}(B[1, \ldots, n/2], C[1, \ldots, n/2]) \)
Mergesort: Correctness

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

**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 $n+1$.
- 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)
Mergesort: Time analysis

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

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 \( \text{Merge}(B[1, \ldots, n/2], C[1, \ldots, n/2]) \)

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

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