TABLE OF CONTENTS

- Course mechanics
- **Overview** of course material
- Worked example: Bentley’s problem
  - Multiple solutions, demonstrating different algorithm design techniques

COURSE MECHANICS

- **Course website**: https://www.student.cs.uwaterloo.ca/~cs341/
  - Syllabus, calendar, policies, slides, assignments...
  - Read this and mark important dates.
- **Keep up with the lectures**: Material builds over time...
- **Plaza**: For questions and announcements.

ASSESSMENTS

- **All sections** have **same** assignments, midterm and final
  - Notify us **long before** the deadline of severe problems that will cause you to miss an assignment
  - Midterm and final are to be take-home exams
  - See website for grading scheme

TEXTBOOK

- Introduction to Algorithms, Third Edition
  - Cormen, Leiserson, Rivest and Stein
  - Available for free via library website!
  - You are expected to know
    - entire textbook sections, as listed on course website
  - **all the material presented in lectures** (unless we explicitly say you aren’t responsible for it)
ACADEMIC OFFENSES

- Beware plagiarism
- **High level discussion** about solutions with individual students is **OK**
- Don’t take written notes away from such discussions
- Class-wide discussion of solutions is **not** OK [until the deadline]

WHY IS CS341 IMPORTANT FOR YOU?

- Algorithms is the heart of CS
- It appears often in later courses
- It dominates technical interviews
- Master this material... make your interviews easy!
- Designing algorithms is **creative** work
- Useful for some of the more interesting jobs out there
- And, you want to graduate...

CS 341 is a required course for all CS

EXAMPLES OF COMPUTATIONAL PROBLEMS

<table>
<thead>
<tr>
<th>Input</th>
<th>Sorting</th>
<th>Matrix Multiplication</th>
<th>Traveling Salesman Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>An array of integers (in arbitrary order)</td>
<td>Two ( n \times n ) matrices ( A, B )</td>
<td>A set ( S ) of cities, and distances between each pair of cities</td>
<td></td>
</tr>
<tr>
<td>A set ( S ) of cities, and distances between each pair of cities</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2 1 5 1 3 4
3 2 2 1 1 1
1 4 6 3 7 2

19 41 18
13 25 19
27 49 20

ANALYSIS OF ALGORITHMS

- Every software program uses **resources**
- CPU instructions \( \rightarrow \) we call this **time**
- Memory (RAM) \( \rightarrow \) we call this **space**
- Others: I/O, network bandwidth/messages, locks... (not covered in this course)
- **Analysis** is the study of how many resources an algorithm uses
- Usually using big-O notation (to ignore constant factors)
This course mainly covers:
Serial, deterministic, exact

**TAXONOMY OF ALGORITHMS**
- Serial vs Parallel
  - Serial: One instruction at a time
  - Parallel: Multiple instructions at once
- Deterministic vs Randomized
  - D: On multiple runs on same input, always do same thing
  - R: On multiple runs on same input, may do different things
    Example: flip a coin, and base your next action on the result
- Exact vs Approximate
  - Exact: exact solution to the problem
  - Approximate: produce something “close” to a solution

**TRACTABILITY: DO ALL PROBLEMS HAVE FAST SOLUTIONS?**
- For some problems, such as the traveling salesman problem, we have only found exponential time algorithms.
- These algorithms take exponentially longer to solve the problem as the number of cities increases!
- Informally: adding one city doubles the runtime
- This severely limits our ability to solve “real world” inputs...
- Is there a way around this limitation? Or should we stop trying?
- Open question (P vs NP): is it possible to solve such problems in polynomial time?

**Topics to Cover**
- **Fundamental (& Fast) Algorithms for Tractable Problems**
  - MergeSort
  - Strassen’s MMM
  - BFS/DFS
  - Dijkstra’s SSSP
  - MST (Kruskal or Prim)
  - Topological Sort
  - ...  
- **Common Algorithm Design Paradigms**
  - Divide-and-Conquer
  - Greedy
  - Dynamic Programming
  - Exhaustive search / brute force
- **Math Techniques for Algorithm Analysis**
  - Big-oh notation
  - Recursion Tree
  - Master method
  - Substitution method
  - Exchange Arguments
- **Bentley’s Problem (introductory example)**
  Given an array of n integers, A1, ..., An, find the maximum sum of consecutive entries of A (return 0 if all entries of A are negative).

**CS341: Before → After**

**Example 1**
```
| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
```
Solution: 19
(take all of A[1..8])

**Example 2**
```
| Index | -1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
```
Solution: 0
(take no elements of A)

**Example 3**
```
| Index | 1 | -7 | 4 | 5 | 2 | 1 | 3 | 8 |
```
Solution: 8
(take A[3..7])

**Bentley’s Problem**
- A worked example to demonstrate algorithm design:
Bentley’s Problem: Solution 1

Try all combinations of i, j
And for each combination,
sum over k = i..j

Design: brute force
Avoid summing over $k = i..j$

Design: slightly better brute force

WHY $A[i \ldots j]$ IS MAXIMAL

• Suppose not for contradiction
• Then some $(i' \ldots j')$ that crosses the partition has a larger sum

But both are impossible!

Find $i$ that maximizes the sum over $i \ldots m/2$

Find $j$ that maximizes the sum over $(m/2+1) \ldots j$

We can prove $A[i \ldots j]$ is the maximum subarray going over the middle partition!

Therefore: Find the maximum subarray for left part ($maxL$) and right part ($maxR$) (done by recursive call).
Find the maximum subarray “going over the middle partition line” ($maxM$).
This can be done in linear time $O(n)$.
The solution is max $maxL$, $maxR$, $maxM$. 

WHY $A[i \ldots j]$ IS MAXIMAL

• Suppose not for contradiction
• Then some $(i' \ldots j')$ that crosses the partition has a larger sum

But both are impossible!
Bentley’s Problem: Solution 4

- Define: \( \text{include}(j) = \text{maximum sum} \) of consecutive entries in array \( A[1..j] \)
  If the sum must include \( A[j] \)

- Define: \( \text{exclude}(j) = \text{maximum sum} \) of consecutive entries in array \( A[1..j] \)
  If the sum must exclude \( A[j] \)

- Observe: if we could solve for \( \text{include}(j) \), \( \text{exclude}(j) \) for all \( j \), then the solution to our problem would be \( \max(\text{include}(n), \text{exclude}(n)) \)

Example: computing these recurrence relations with two arrays

- Base case: \( \text{include}(1) = A[1] \)
- \( \text{include}(j) = \max(\text{include}(j), A[j] + \text{include}(j-1)) \)
- \( \text{exclude}(j) = \max(\text{exclude}(j), \text{include}(j-1)) \)
- Base case: \( \text{exclude}(1) = 0 \)

Recall the definition:

- \( \text{include}(1) = \max \) solution in \( A[1..1] \) that includes \( A[1] \)
- \( \text{include}(j) = \max \) solution in \( A[1..j] \) that includes \( A[j] \)
- \( \text{exclude}(1) = \max \) solution in \( A[1..1] \) that excludes \( A[1] \)
- \( \text{exclude}(j) = \max \) solution in \( A[1..j] \) that excludes \( A[j] \)

Example: computing these recurrence relations with two arrays

- Base case: \( \text{include}(1) = A[1] \)
- \( \text{include}(j) = \max(\text{include}(j), A[j] + \text{include}(j-1)) \)
- \( \text{exclude}(j) = \max(\text{exclude}(j), \text{include}(j-1)) \)
- Base case: \( \text{exclude}(1) = 0 \)

Recall:

Base case: \( \text{exclude}(1) = 0 ; \text{include}(1) = A[1] \)

Recursive case:
- \( \text{include}(j) = \max(\text{include}(j-1), \text{exclude}(j-1)) \)
- \( \text{exclude}(j) = \max(\text{include}(j-1), \text{exclude}(j-1)) \)

Let's turn these recurrences into code...

```
function solveDE(A)
    let n = size(A)
    // base case
    if n == 1, then return max(0, A[1])
    // recursive case
    for j = 2 to n
        include[j] = max(include[j-1], exclude[j-1])
        exclude[j] = max(A[j], include[j-1] + exclude[j-1])
    return max(exclude[n], include[n])
```
At this time, include contains exactly \[\text{include}[j-1]\] And similarly for exclude...

And these contain exactly \[\text{exclude}[n]\] and \[\text{include}[n]\]

HOW ABOUT A MORE MODERN SYSTEM? 😊

BONUS

• Trevor’s study-song of the day
• Tool - Descending
• youtu.be/PcSoLwFisaw

BENTLEY’S PROBLEM: TIME CONSTRAINTS

- Consider solutions implemented in C
- Some values measured (on a Pentium II)
- Some estimated from other measurements
- \(\epsilon\) represents time under 0.01s

<table>
<thead>
<tr>
<th>Max size problem</th>
<th>1X</th>
<th>100X</th>
<th>1000X</th>
<th>10000X</th>
<th>100000X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to solve a</td>
<td>1s</td>
<td>2.3 ms</td>
<td>34 ms</td>
<td>3.56 s</td>
<td>2.08 ms</td>
</tr>
<tr>
<td>Space (besides the input array)</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(n^2)</td>
<td>O(n^3)</td>
<td>O(n^4)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AMD Threadripper 3970x (2020)</th>
<th>Pentium II (circa 1997)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to solve a</td>
<td>2 minutes</td>
</tr>
<tr>
<td>Space (besides the input array)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

Pentium II (circa 1997)

| Time to solve a | 2 minutes | 33 hours |
| Space (besides the input array) | O(1) | 640 GB |

Pentium II (circa 1997)

| Time to solve a | 2 minutes | 33 hours |
| Space (besides the input array) | O(1) | 640 GB |