CS341: ALGORITHMS (W20)

Lecture 1: course overview and Bentley’s problem

Trevor Brown

trevor.brown@uwaterloo.ca
(http://tbrown.pro)
TABLE OF CONTENTS (FOR TODAY)

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

And administrivia...
COURSE MECHANICS

• **In lectures:** Ask lots of questions! Material **builds** over time...

• **Course website:** https://www.student.cs.uwaterloo.ca/~cs341/
  • Syllabus, calendar, policies, slides, assignments...
  • Read this and **stay up to date.** Mark important dates.

• **Piazza:** Join it. Ask questions. Answer other students’ questions. Read this and **stay up to date.**

• **Assignments:**
  • **Crowdmark:** submission/grading of **written** parts
  • **Marmoset:** submission/grading of **coding** parts
    • Need an account in **student.cs** environment!
GRADING SCHEME

- Midterm (25%)
  - Friday, Feb. 25, 2020, 7:00-8:50 PM
- Assignments (30%)
  - Plan is to have five assignments.
  - Solo assignments --- not group work
- Final (45%)

A1 should be due January 27th at 6pm.
ASSIGNMENTS

• All sections have same assignments, midterm and final
• Assignments due at 6:00 PM on the due date
  • No late submissions
  • Notify us long before the deadline of severe problems that will cause you to miss an assignment
• VIF form will be required; weight will be shifted to other assignments
**TEXTBOOK**

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

- Assignments
  - **High level discussion about solutions** is OK
  - **Don’t** simply give solutions to your classmates
  - (Don’t take written notes away from such a discussion)
- Midterm/final
  - No notes/aides. We know common cheating methods...
COURSE OVERVIEW
Sketching out the road ahead
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...
WHAT IS A **COMPUTATIONAL PROBLEM**?

- Informally: A description of input, and the **desired output**

WHAT IS AN **ALGORITHM**?

- Informally: A well-defined **procedure** (sequence of steps) to solve a **computational problem**
# Examples of Computational Problems

<table>
<thead>
<tr>
<th></th>
<th>Sorting</th>
<th>Matrix Multiplication</th>
<th>Traveling Salesman Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td>An array of integers</td>
<td>Two $n \times n$ matrices $A$, $B$</td>
<td>A set $S$ of cities, and distances between each pair of cities</td>
</tr>
<tr>
<td><strong>Desired output</strong></td>
<td>Same array of integers in <strong>increasing</strong> order</td>
<td>A matrix $C = A \times B$</td>
<td>Shortest possible path that visits each city, and returns to the origin city</td>
</tr>
</tbody>
</table>

**Input:** An array of integers (in arbitrary order)

**Desired output:** Same array of integers in **increasing** order

**Matrix Multiplication:**

\[
\begin{pmatrix}
2 & 1 & 5 \\
3 & 2 & 2 \\
1 & 4 & 6
\end{pmatrix}
\times
\begin{pmatrix}
1 & 3 & 4 \\
2 & 1 & 1 \\
3 & 7 & 2
\end{pmatrix}
= \begin{pmatrix}
19 & 41 & 18 \\
13 & 25 & 19 \\
27 & 49 & 20
\end{pmatrix}
\]

**Traveling Salesman Problem:**

- **Cities:** $C_1$, $C_2$, $C_3$, $C_4$, $C_5$
- **Distances:**
  - $C_1$ to $C_2$: 5 hours
  - $C_1$ to $C_3$: 2 hours
  - $C_1$ to $C_4$: 3 hours
  - $C_1$ to $C_5$: 1 hour
  - $C_2$ to $C_3$: 1 hour
  - $C_2$ to $C_4$: 2 hours
  - $C_2$ to $C_5$: 4 hours
  - $C_3$ to $C_4$: 3 hours
  - $C_3$ to $C_5$: 1 hour
  - $C_4$ to $C_5$: 2 hours
ANALYSIS OF ALGORITHMS

• Every software program uses resources
  • CPU instructions → we call this time
  • Memory (RAM) → 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)
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

This course mainly covers:
Serial, deterministic, exact
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): are there problems so hard that any solution must take exponential time on some “hard” inputs?
Topics to Cover

Fundamental (& Fast) Algorithms to Tractable Problems
- MergeSort
- Strassen’s MM
- BFS/DFS
- Dijkstra’s SSSP
- Kruskal’s MST
- Floyd Warshall APSP
- Topological Sort
- ...

Common Algorithm Design Paradigms
- Divide-and-Conquer
- Greedy
- Dynamic Programming
- Exhaustive search / brute force

Mathematical Tools to Analyze Algorithms
- Big-oh notation
- Recursion Tree
- Master method
- Substitution method
- Exchange Arguments
- Greedy-stays-ahead Arguments

Intractable Problems
- P vs NP
- Poly-time Reductions
- Undecidability

Other (Last Lecture)
- Parallel Algorithms?
CS341: Before ➔ After

1. Fundamental Algorithms
2. Fundamental Design Paradigms
3. Tractability/Intractability

Math Techniques for Algorithm Analysis
BENTLEY’S PROBLEM

A worked example to demonstrate algorithm design
Bentley’s Problem (introductory example)

Given an array of \( n \) integers, \( A[1], \ldots, A[n] \), find the maximum sum of consecutive entries of \( A \) (return 0 if all entries of \( A \) are negative).

**Example 1**

<table>
<thead>
<tr>
<th>Array index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Solution: 19  
(take all of \( A[1..8] \))

**Example 2**

<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1</td>
<td>-7</td>
<td>-4</td>
<td>-1</td>
<td>-2</td>
<td>-1</td>
<td>-3</td>
<td>-1</td>
</tr>
</tbody>
</table>

Solution: 0  
(take no elements of \( A \))

**Example 3**

<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>-7</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>-1</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>

Solution: 8  
(take \( A[3..7] \))
Bentley’s Problem: Solution 1

max := 0;
for i := 1 to n do
    for j := i to n do
        // compute A[i] + ... + A[j]
        sum := 0;
        for k := i to j do
            sum := sum + A[k];
        // compare to maximum sum observed so far
        if sum > max then max := sum;

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

Design: brute force

Time: Delete me to reveal answer
Bentley’s Problem: Solution 2

max := 0;
for i := 1 to n do
    // for each j, compute A[i] + ... + A[j]
    sum := 0;
    for j := i to n do
        // update sum by adding the next entry A[j]
        sum := sum + A[j];
        // compare to maximum sum observed so far
        if sum > max then max := sum;

Avoid summing over $k = i .. j$

Design: slightly better brute force
Bentley’s Problem: Solution 3

Divide-and-Conquer can also be used here:
Divide an array into two equally-sized parts.
Our solution must either be entirely in the left part, or entirely in
the right part, or it must be crossing the partition line.

Case: optimal sol’n is entirely in L

Case: optimal sol’n is entirely in R

Case: optimal sol’n crosses the 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 \(\Theta(n)\).

The solution is \(\max \, maxL, maxR, maxM\).

Find \(i\) that maximizes the sum over \(i\) \(\ldots\) \(n/2\).

\(A[i \ldots j]\) is the maximum subarray going over the middle partition!

Find \(j\) that maximizes the sum over \(\left(\frac{n}{2} + 1\right) \ldots j\).

Proving this is the tricky part. We will learn more when we study greedy algos.

This is the "greedy" part of the algorithm.
function solveDnC(A)
    let n = sizeof(A)

    // base case
    if n == 1 then return max(0, A[1])

    // recursive case
    maxL = solveDnC(A[1 .. n/2])
    maxR = solveDnC(A[n/2+1 .. n])

    // compute maxM
    tempSum = 0
    maxI = 0
    for i = n/2 .. 1
        tempSum = tempSum + A[i]
        if tempSum > maxI then maxI = tempSum
    tempSum = 0
    maxJ = 0
    for j = n/2+1 .. n
        tempSum = tempSum + A[j]
        if tempSum > maxJ then maxJ = tempSum
    maxM = maxI + maxJ
    return max(maxL, maxR, maxM)
function solveDnC(A)
    let n = sizeof(A)
    // base case
    if n == 1 then return max(0, A[1])
    // recursive case
    maxL = solveDnC(A[1 .. n/2])
    maxR = solveDnC(A[n/2+1 .. n])
    // compute maxM
    tempSum = 0
    maxI = 0
    for i = n/2 .. 1
        tempSum = tempSum + A[i]
        if tempSum > maxI then maxI = tempSum
    tempSum = 0
    maxJ = 0
    for j = n/2+1 .. n
        tempSum = tempSum + A[j]
        if tempSum > maxJ then maxJ = tempSum
    maxM = maxI + maxJ
    return max( maxL, maxR, maxM )
How do we analyze this running time?
Need new mathematical techniques!
Recurrence relations, recursion tree methods, master theorem...
This result is really quite good...
but can we do asymptotically better?
Bentley’s Problem: Solution 4

Design: dynamic programming

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

• Define: \(\text{exclude}(j) = \text{maximum sum}\) of consecutive entries in array \(A[1..j]\) if the sum must \textbf{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 \(\max\{ \text{include}(n), \text{exclude}(n) \}\)

• We can define \textbf{recurrence relations} to solve
  - Base case: \(\text{include}(1) = A[1]\)
  - Base case: \(\text{exclude}(1) = 0\)
  - \(\text{include}(j) = \max\{ A[j], A[j] + \text{include}(j - 1) \}\)
  - \(\text{exclude}(j) = \max\{ \text{include}(j - 1), \text{exclude}(j - 1) \}\)
Example: computing these recurrence relations with two arrays

- **Base case:** $include(1) = A[1]$
- $include(j) = \max\{ A[j], A[j] + include(j - 1) \}$

<table>
<thead>
<tr>
<th>$A$</th>
<th>1</th>
<th>-7</th>
<th>4</th>
<th>0</th>
<th>2</th>
<th>-1</th>
<th>3</th>
<th>-9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>$include$</td>
<td>1</td>
<td>-6</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>8</td>
<td>-1</td>
</tr>
</tbody>
</table>

- **Base case:** $exclude(1) = 0$
- $exclude(j) = \max\{ include(j - 1), exclude(j - 1) \}$

| $exclude$ | 0  | 1  | 1  | 4  | 4  | 6  | 6  | 8  |

Recall the definition:

- $include(1) = \text{"max solution in } A[1..1] \text{ that includes } A[1]...$
- $include(2) = \text{"max solution in } A[1..2] \text{ that includes } A[2]...$
- $include(3) = \text{"max solution in } A[1..3] \text{ that includes } A[3]...$

- $exclude(1) = \text{"max solution in } A[1..1] \text{ that excludes } A[1]...$
- $exclude(2) = \text{"max solution in } A[1..2] \text{ that excludes } A[2]...$
- $exclude(3) = \text{"max solution in } A[1..3] \text{ that excludes } A[3]...$

Full solution is $\max$ of these two: 8
Base case: \( \text{exclude}(1) = 0 \); \( \text{include}(1) = A[1] \)

Recursive case:

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

Let's turn these recurrences into code...

```
function solveDP(A)
    define arrays exclude[1..n], include[1..n]

    exclude[1] = 0

    for j = 2..n
        exclude[j] = max( include[j-1], exclude[j-1] )
        include[j] = max( A[j], A[j] + include[j-1] )

    return max( exclude[n], include[n] )
```
At this time, include contains exactly \texttt{include[j-1]}

And similarly for exclude...

And these contain exactly \texttt{exclude[n]} and \texttt{include[n]}
BENTLEY’S PROBLEM: TIME CONSTRAINTS

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

<table>
<thead>
<tr>
<th>Time to solve a problem of size:</th>
<th>Sol.4 $\Theta(n)$</th>
<th>Sol.3 $\Theta(n \log n)$</th>
<th>Sol.2 $\Theta(n^2)$</th>
<th>Sol.1 $\Theta(n^3)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td>50</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td>100</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td>1000</td>
<td>$\varepsilon$</td>
<td>0.01s</td>
<td>2.1s</td>
<td>4.5s</td>
</tr>
<tr>
<td>10000</td>
<td>0.04s</td>
<td>0.12s</td>
<td>3.5m</td>
<td>75m</td>
</tr>
<tr>
<td>1 mil.</td>
<td>0.42s</td>
<td>1.4s</td>
<td>5.8h</td>
<td>142yrs.</td>
</tr>
<tr>
<td>10 mil.</td>
<td>4.2s</td>
<td>16.1s</td>
<td>24.3d</td>
<td>1400000yrs.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Max size problem solved in time if $n$ increases:</th>
<th>1s</th>
<th>2.3 mil.</th>
<th>740000</th>
<th>6900</th>
<th>610</th>
</tr>
</thead>
<tbody>
<tr>
<td>1m</td>
<td>140 mil.</td>
<td>34 mil.</td>
<td>53000</td>
<td>2400</td>
<td></td>
</tr>
<tr>
<td>1d</td>
<td>200 bil.</td>
<td>35 bil.</td>
<td>2 mil.</td>
<td>26000</td>
<td></td>
</tr>
</tbody>
</table>