CS341: ALGORITHMS (W21)
Lecture 1: course overview and Bentley’s problem
Readings: CLRS Chapter 1
Trevor Brown (co-taught with Anna Lubiw)
https://www.student.cs.uwaterloo.ca/~cs341/
trevor.brown@uwaterloo.ca

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…
• Piazza: For questions and announcements.

ASSESSMENTS
• All sections have same assignments, midterm and final
• Notify us soon 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

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

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: Is it possible to solve such problems in polynomial time?

Topics to Cover

Fundamental (& Fast) Algorithms for Tractable Problems
  • MergeSort
  • QuickSort
  • Hashing
  • RSA CRYPTO
  • Depth-first Search
  • Bellman-Ford (for n nodes)
  • Floyd-Warshall APSP
  • Topological Sort
  ... 

Common Algorithm Design Paradigms
  • Divide-and-Conquer
  • Backtrack
  • Greedy
  • Dynamic Programming
  • Exhausitve search / brute force

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

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

BENTLEY’S PROBLEM

A worked example to demonstrate algorithm design

CS341: Before → After

1. Fundamental Algorithms
2. Fundamental Design Paradigms
3. Tractability/Intractability
4. Math Techniques for Algorithm Analysis

Bentley’s Problem (introductory example)

Given an array of n integers, A[1], ..., 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>Index</th>
<th>1</th>
<th>1</th>
<th>7</th>
<th>4</th>
<th>0</th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</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>
<td>9</td>
</tr>
</tbody>
</table>

Solution: 19
(take all of A[1..6])

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>2</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</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>
<td>9</td>
</tr>
</tbody>
</table>

Solution: 0
(take no elements of A)

Example 3

<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>3</th>
<th>7</th>
<th>4</th>
<th>0</th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</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>
<td>9</td>
</tr>
</tbody>
</table>

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

Design: brute force

```
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;
```

Time: 

Try all combinations of \( i, j \) and for each combination, sum over \( k = i \ldots j \).

Bentley’s Problem: Solution 2

Design: slightly better brute force

Avoid summing over \( k = i \ldots j \).

```
max := 0;
for i := 1 to n do
    for j := i 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;
```

Time: 

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 \( max = \maxL, maxR, maxM \).

```
maxL := 10
maxR := 3
maxI := 5
maxJ := 0
maxM := maxI + maxJ = 5
Return max( 10, 3, 5 ) = 10
```

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.

```
maxL := 10
maxR := 3
maxI := 5
maxJ := 0
maxM := maxI + maxJ = 5
Return max( 10, 3, 5 ) = 10
```

Let’s see how…

### Case 1: optimal sol’n is entirely in L

### Case 2: optimal sol’n is entirely in R

### Case 3: optimal sol’n crosses the partition

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

WHY \( A[i \ldots j] \) IS MAXIMAL

- Suppose not for contradiction
- Then some \( A[i' \ldots j'] \) that crosses the partition has a larger sum

```
maxL := 10
maxR := 3
maxI := 5
maxJ := 0
maxM := maxI + maxJ = 5
Return max( 10, 3, 5 ) = 10
```

This sum is bigger!
So either \( i' > i \) or \( j' > j \).
**Bentley’s Problem: Solution 4**

- **Define: include(j) = maximum sum of consecutive entries in array A[1..j]**
  - If the sum must **include** A[j],
- **Define: exclude(j) = maximum sum of consecutive entries in array A[1..j]**
  - If the sum must **exclude** A[j],
- **Observe:** If we could solve for include(j), exclude(j) for all j, then the solution to our problem would be \( \max(\text{include}(n), \text{exclude}(n)) \)

**Example:** computing these recurrences relations with two arrays

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

**Recall the definition:**
- include(j) = max solution in A[1.. j] that includes A[j],
- exclude(j) = max solution in A[1.. j] that excludes A[j],

**Base case:** exclude(1) = 0 : include(1) = A[1]

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

**Let’s turn these recurrences into code...**

**Example:**

```plaintext
function solveDP(A)
    define arrays exclude[1..n], include[1..n]
    exclude[1] = 0
    for j = 2..n
        include[j] = max( include[j-1], A[j] + include[j-1] )
    return max( exclude[n], include[n] )
```

**Recurrence relations, recursion tree methods, master theorem...**

**This result is really quite good, but can we do asymptotically better?**
At this time, include contains exactly \[ \text{include}[j-1] \]
And similarly for exclude...

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

Same running time, but only \( O(1) \) space (besides the input array)

cant be done

\( \text{Pentium II (circa 1997)} \)

\( \text{AMD Threadripper 3970x (2020)} \)

<table>
<thead>
<tr>
<th>Problem</th>
<th>( n )</th>
<th>( 0.001 )</th>
<th>( 0.012 )</th>
<th>( 0.112 )</th>
<th>( 1.124 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{columns} )</td>
<td>100</td>
<td>3.582</td>
<td>12 hours</td>
<td>3700 years</td>
<td>3.7M years</td>
</tr>
<tr>
<td>( \text{columns} )</td>
<td>1000</td>
<td>0.002</td>
<td>3.582</td>
<td>12 hours</td>
<td>3700 years</td>
</tr>
<tr>
<td>( \text{columns} )</td>
<td>10000</td>
<td>0.036</td>
<td>3.582</td>
<td>12 hours</td>
<td>3700 years</td>
</tr>
<tr>
<td>( \text{columns} )</td>
<td>1 million</td>
<td>0.42s</td>
<td>3.582</td>
<td>12 hours</td>
<td>3700 years</td>
</tr>
<tr>
<td>( \text{columns} )</td>
<td>10 million</td>
<td>4.2s</td>
<td>3.582</td>
<td>12 hours</td>
<td>3700 years</td>
</tr>
</tbody>
</table>

\( \text{Time to solve a problem of size} \)

\( \text{Max size problem solved}\) | \( \text{Latency} \) | \( \text{Latency} \) | \( \text{Latency} \) | \( \text{Latency} \) | \( \text{Latency} \) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Problem size} )</td>
<td>( 100 )</td>
<td>0.001</td>
<td>0.012</td>
<td>0.112</td>
<td>1.124</td>
</tr>
<tr>
<td>( \text{Problem size} )</td>
<td>( \text{seconds} )</td>
<td>0.001</td>
<td>0.012</td>
<td>0.112</td>
<td>1.124</td>
</tr>
<tr>
<td>( \text{Problem size} )</td>
<td>( \text{minutes} )</td>
<td>0.001</td>
<td>0.012</td>
<td>0.112</td>
<td>1.124</td>
</tr>
<tr>
<td>( \text{Problem size} )</td>
<td>( \text{hours} )</td>
<td>0.001</td>
<td>0.012</td>
<td>0.112</td>
<td>1.124</td>
</tr>
</tbody>
</table>

\( \text{How about a more modern system?} \)

\( \text{ytd.therea/PCsOfTheDay} \)

**BONUS**

- Trevor’s study-song of the day
- Tool · Descending
- [youtube](https://www.youtube.com/watch?v=PCsOfTheDay)