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/](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 **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...
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>Input</th>
<th>Desired output</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>Same array of integers in <strong>increasing</strong> order</td>
<td>An array of integers (in arbitrary order)</td>
<td>Two $n \times n$ matrices $\mathbf{A}$, $\mathbf{B}$</td>
<td>A set $\mathcal{S}$ of cities, and distances between each pair of cities</td>
</tr>
<tr>
<td>Two $n \times n$ matrices $\mathbf{A}$, $\mathbf{B}$</td>
<td>A matrix $\mathbf{C} = \mathbf{A} \times \mathbf{B}$</td>
<td>A matrix $\mathbf{C} = \mathbf{A} \times \mathbf{B}$</td>
<td>Shortest possible path that visits each city, and returns to the origin city</td>
<td></td>
</tr>
</tbody>
</table>

![Graph showing the Traveling Salesman Problem](image)
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
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 MM
- BFS/DFS
- Dijkstra’s SSSP
- MST (Kruskal or Prim)
- 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
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).

<table>
<thead>
<tr>
<th>Example 1</th>
<th>Solution: 19</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array index</td>
<td>1 7 4 0 2 1 3 1</td>
</tr>
<tr>
<td>Index</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example 2</th>
<th>Solution: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>-1 -7 -4 -1 -2 -1 -3 -1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example 3</th>
<th>Solution: 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>1 -7 4 0 2 -1 3 -1</td>
</tr>
</tbody>
</table>
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;

Design: brute force

Try all combinations of \(i, j\)
And for each combination, sum over \(k = i \ldots j\)
Bentley’s Problem: Solution 2

max := 0;
for i := 1 to n do
    // for each j, compute $A[i] + \ldots + A[j]$
    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 $k = i \ldots j$

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 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
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:** maximum subarray going over the middle partition

<table>
<thead>
<tr>
<th>Index</th>
<th>(1)</th>
<th>(-7)</th>
<th>(4)</th>
<th>(0)</th>
<th>(2)</th>
<th>(-1)</th>
<th>(3)</th>
<th>(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>(\ldots)</td>
<td>(n/2)</td>
<td>(n/2+1)</td>
<td>(\ldots)</td>
<td>(n)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

This sum is bigger

So either $\sum L' > \sum L$ or $\sum R' > \sum R$

But both are impossible!
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 )

A = [9, -3, 4, -5, -2, -5, 3, -1]

Index
1 2 3 4 5 6 7 8
\[ \begin{array}{cccccc}
9 & -3 & 4 & -5 & -2 & -5 & 3 & -1 \\
\end{array} \]

maxL = 10
maxR = 3
maxM = maxL + maxJ = 5
maxI = 5
maxJ = 0

Return max( 10, 3, 5 ) = 10
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 \textit{asymptotically} better?
Bentley’s Problem: Solution 4

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

- Define: \( \text{exclude}(j) = \text{maximum sum} \) of consecutive entries in array \( A[1..j] \) if the sum must \text{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 \( \max\{ \text{include}(n), \text{exclude}(n) \} \)
Bentley’s Problem: Solution 4

• We can define recurrence relations to solve for include and exclude

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

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

  “Max sum in A[1..1] if we must include A[1]”

  If including \( A[j] \), there are two possibilities: either start a new sum of consecutive entries at \( A[j] \), or extend the best sum that ends at \( A[j - 1] \)

  If excluding \( A[j] \), the best we can do in \( A[1..j] \) is simply the best we can do in \( A[1..j - 1] \)
Example: computing these recurrence relations with two arrays

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

<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:** \( \text{exclude}(1) = 0 \)
- \( \text{exclude}(j) = \max\{ \text{include}(j - 1), \text{exclude}(j - 1) \} \)

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

Recall the definition:

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

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

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

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

Let's turn these recurrences into code...

```python
1  function solveDP(A):
2      define arrays exclude[1..n], include[1..n]
3  
4      exclude[1] = 0
6      for j = 2..n
7          exclude[j] = max( include[j-1], exclude[j-1] )
8          include[j] = max( A[j], A[j] + include[j-1] )
9  
10     return max(exclude[n], include[n])
```

Recall: Do we actually need these entire arrays? Only really care about the last entry of each...
At this time, include contains exactly "include[j-1]"

And similarly for exclude...

And these contain exactly "exclude[n]" and "include[n]"

Same running time, but only $O(1)$ space (besides the input array)
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</th>
<th>Sol. 3</th>
<th>Sol. 2</th>
<th>Sol. 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n \log n)$</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^3)$</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>
<tr>
<td>Max size problem solved in time if n increases:</td>
<td>1s</td>
<td>2.3 mil.</td>
<td>740000</td>
<td>6900</td>
</tr>
<tr>
<td></td>
<td>1m</td>
<td>140 mil.</td>
<td>34 mil.</td>
<td>53000</td>
</tr>
<tr>
<td></td>
<td>1d</td>
<td>200 bil.</td>
<td>35 bil.</td>
<td>2 mil.</td>
</tr>
</tbody>
</table>
HOW ABOUT A MORE MODERN SYSTEM? 😊
### AMD Threadripper 3970x (2020)

<table>
<thead>
<tr>
<th>N</th>
<th>Sol.4</th>
<th>Sol.3</th>
<th>Sol.2</th>
<th>Sol.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1,000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.12</td>
</tr>
<tr>
<td>10,000</td>
<td>0</td>
<td>0</td>
<td>0.036</td>
<td>2 minutes</td>
</tr>
<tr>
<td>100,000</td>
<td>0</td>
<td>0.002</td>
<td>3.582</td>
<td>33 hours</td>
</tr>
<tr>
<td>1M</td>
<td>0.001</td>
<td>0.017</td>
<td>6 minutes</td>
<td>4 years</td>
</tr>
<tr>
<td>10M</td>
<td>0.012</td>
<td>0.195</td>
<td>12 hours</td>
<td>3700 years</td>
</tr>
<tr>
<td>100M</td>
<td>0.112</td>
<td>2.168</td>
<td>50 days</td>
<td>3.7M years</td>
</tr>
<tr>
<td>1 billion</td>
<td>1.124</td>
<td>24.57</td>
<td>1.5 years</td>
<td>&gt; age of life</td>
</tr>
<tr>
<td>10 billion</td>
<td>19.15</td>
<td>5 minutes</td>
<td>150 years</td>
<td>&gt; age of universe</td>
</tr>
</tbody>
</table>

### Pentium II (circa 1997)

<table>
<thead>
<tr>
<th>Sol.4</th>
<th>Sol.3</th>
<th>Sol.2</th>
<th>Sol.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td>ε</td>
<td>ε</td>
<td>0.02s</td>
<td>4.5s</td>
</tr>
<tr>
<td>ε</td>
<td>0.01s</td>
<td>2.1s</td>
<td>75m</td>
</tr>
<tr>
<td>0.04s</td>
<td>0.12s</td>
<td>3.5m</td>
<td>52d</td>
</tr>
<tr>
<td>0.42s</td>
<td>1.4s</td>
<td>5.8h</td>
<td>142yrs.</td>
</tr>
<tr>
<td>4.2s</td>
<td>16.1s</td>
<td>24.3d</td>
<td>1400000yrs.</td>
</tr>
</tbody>
</table>
BONUS

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