## Algorithm Design and Time Analysis

<table>
<thead>
<tr>
<th>Lecture A</th>
<th>Tiefenbruck</th>
<th>MWF 9-9:50am</th>
<th>Center 212</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lecture B</td>
<td>Jones</td>
<td>MWF 2-2:50pm</td>
<td>Center 214</td>
</tr>
<tr>
<td>Lecture C</td>
<td>Tiefenbruck</td>
<td>MWF 11-11:50am</td>
<td>Center 212</td>
</tr>
</tbody>
</table>

[http://cseweb.ucsd.edu/classes/wi16/cse21-abc/](http://cseweb.ucsd.edu/classes/wi16/cse21-abc/)

January 15, 2016
Today’s Plan

Analyzing algorithms that solve other problems (besides sorting and searching)

Designing better algorithms
  • pre-processing
  • re-use of computation
Summing Triples: WHAT

Given a list of real numbers

\[ a_1, a_2, \ldots, a_n \]

look for three indices, i, j, k (each between 1 and n) such that

\[ a_i + a_j = a_k \]

Does the list 3, 6, 5, 7, 8 have a summing triple?

A. Yes: 1, 2, 3
B. Yes: 1, 3, 5
C. No
Given a list of real numbers

\[ a_1, a_2, \ldots, a_n \]

look for three indices, i, j, k (each between 1 and n) such that

\[ a_i + a_j = a_k \]

Design an algorithm to look for summing triples
Summing Triples: HOW (1)

\[ \text{SumTriples1}(a_1, \ldots, a_n : \text{real numbers}) \]

\[
\begin{array}{c}
\text{for } i := 1 \text{ to } n \\
\quad \text{for } j := 1 \text{ to } n \\
\quad \quad \text{for } k := 1 \text{ to } n \\
\quad \quad \quad \text{if } a_i + a_j = a_k \text{ then return true} \\
\text{return false}
\end{array}
\]

What's the order of the runtime of this algorithm?
A. O(1) 
B. O(n) 
C. O(n²) 
D. O(n³) 
E. None of the above

Summing Triples: HOW (1)

\[ SumTriples1(a_1, \ldots, a_n : \text{real numbers}) \]

\[
\begin{align*}
&\text{for } i := 1 \text{ to } n \\
&\quad \text{for } j := 1 \text{ to } n \\
&\quad\quad \text{for } k := 1 \text{ to } n \\
&\quad\quad\quad \text{if } a_i + a_j = a_k \text{ then return } \text{true} \\
&\text{return } \text{false}
\end{align*}
\]
Summing Triples: HOW (2)

\(\text{SumTriples1}(a_1, \ldots, a_n : \text{real numbers})\)

for \(i := 1\) to \(n\)

\[\text{Eliminate redundancy}\]

for \(j := 1\) to \(n\)

for \(k := 1\) to \(n\)

\[\text{if } a_i + a_j = a_k \text{ then return true}\]

return false
Summing Triples: HOW (2)

SumTriples2(a₁, ..., aₙ : real numbers)

```
for i := 1 to n
    for j := i to n
        for k := 1 to n
            if aᵢ + aⱼ = aₖ then return true

return false
```

What's the order of the runtime of this algorithm?
A. O(1)
B. O(n)
C. O(n²)
D. O(n³)
E. None of the above
Summing Triples: HOW (2)

SumTriples2(a_1, \ldots, a_n : \text{real numbers})

\begin{align*}
\text{for } i & := 1 \text{ to } n \\
\text{for } j & := i \text{ to } n \\
\text{for } k & := 1 \text{ to } n \\
& \quad \text{if } a_i + a_j = a_k \text{ then return } true \\
& \quad \text{return } false
\end{align*}
Reframing what we did:

\[ \text{SumTriples2}(a_1, \ldots, a_n : \text{real numbers}) \]

\[
\begin{align*}
\text{for } i & := 1 \text{ to } n \\
\text{for } j & := i \text{ to } n \\
& \quad \text{For each candidate sum } a_i + a_j,
\end{align*}
\]

\[
\begin{align*}
\text{for } k & := 1 \text{ to } n \\
& \quad \text{do linear search to find it}
\end{align*}
\]

\[
\begin{align*}
\text{if } a_i + a_j = a_k & \text{ then return } \text{true} \\
\text{return } \text{false}
\end{align*}
\]
Summing Triples: HOW (2)

\[ SumTriples2(a_1, \ldots, a_n : \text{real numbers}) \]

\[
\text{for } i := 1 \text{ to } n \\
\quad \text{for } j := i \text{ to } n \\
\quad \quad \text{for } k := 1 \text{ to } n \\
\quad \quad \quad \text{if } a_i + a_j = a_k \text{ then return } true \\
\]

\[ \text{return false} \]

We have a faster search than linear search!
Summing Triples: HOW (3)

\[ \text{SumTriples3}(a_1, \ldots, a_n : \text{real numbers}) \]

\[
\begin{align*}
\text{for } i & := 1 \text{ to } n \\
\text{for } j & := i \text{ to } n \quad \text{For each candidate sum } a_i + a_j, \\
\text{if } \text{BinarySearch}(a_i + a_j; a_1, \ldots, a_n) & \\
\text{then return } true \\
\text{return } false \quad \text{do binary search to find it}
\end{align*}
\]

How long would this take?
A. O(n^3)
B. O(n^2)
C. O(n^2 \log n)
D. O(n \log n)
Does this algorithm really work???
Summing Triples: HOW (3)

\[ \text{SumTriples}_3(a_1, \ldots, a_n : \text{real numbers}) \]

\[
\begin{align*}
&\text{for } i := 1 \text{ to } n \\
&\quad \text{for } j := i \text{ to } n \\
&\quad \quad \text{if BinarySearch}(a_i + a_j; a_1, \ldots, a_n) \\
&\quad \quad \quad \text{then return } \text{true} \\
&\quad \text{return } \text{false}
\end{align*}
\]

For each candidate sum \(a_i + a_j\),

Does this algorithm really work???
Summing Triples: HOW (4)

$$SumTriples4(a_1, \ldots, a_n : \text{real numbers})$$

$$MinSort(a_1, \ldots, a_n)$$

$$SumTriples3(a_1, \ldots, a_n)$$

Preprocessing step

This algorithm works!
How long does it take?

aka SortedSumTriples
Summing Triples: HOW (4)

\[\text{SumTriples}_4(a_1, \ldots, a_n : \text{real numbers})\]

\[\text{MinSort}(a_1, \ldots, a_n) \quad \mathcal{O}(n^2)\]

\[\text{SumTriples}_3(a_1, \ldots, a_n) \quad \mathcal{O}(n^2 \log n)\]

Sum is maximum: \(\mathcal{O}(n^2 \log n)\)
Summing Triples: HOW (4)

\[ \text{SumTriples4}(a_1, \ldots, a_n : \text{real numbers}) \]
\[ \text{MinSort}(a_1, \ldots, a_n) \quad O(n^2) \]
\[ \text{SumTriples3}(a_1, \ldots, a_n) \quad O(n^2 \log n) \]

Sum is maximum: \( O(n^2 \log n) \)

Have we made progress? Can we do better?

- \( \text{SumTriples4} \) does better than \( O(n^3) \).
- Using a faster sort won’t help overall.
- But …. fastest known algorithm: \( O(n^2) \)
"Tight"?

To know that we've actually made improvements, need to make sure our original analysis was not overly pessimistic.

A **tight** bound for runtime is a function $g(n)$ so that the runtime is in $\Theta(g(n))$.

The big-O class for our algorithm : upper bound.

Now want matching big-$\Omega$ : lower bound.
Summing Triples: WHEN (1)

Let $SumTriples1(a_1, \ldots, a_n : \text{real numbers})$

for $i := 1$ to $n$

for $j := 1$ to $n$

for $k := 1$ to $n$

\[ \text{if } a_i + a_j = a_k \text{ then return } \text{true} \]

return $\text{false}$

What's the lower bound order of the worst case runtime of this algorithm?

A. $\Omega(1)$ also correct
B. $\Omega(n)$
C. $\Omega(n^2)$
D. $\Omega(n^3)$ most meaningful
E. None of the above
Summing Triples: WHEN (1)

```
SumTriples1(a_1, \ldots, a_n : \text{real numbers})

\text{for } i := 1 \text{ to } n

\text{for } j := 1 \text{ to } n

\text{for } k := 1 \text{ to } n \quad \Omega(n)

\text{if } a_i + a_j = a_k \text{ then return true } \Omega(1)

\text{return false}
```

Strategy: work from the inside out
Summing Triples: WHEN (2)

$SumTriples2(a_1, \ldots, a_n : \text{real numbers})$

\[ \text{for } i := 1 \text{ to } n \]
\[ \quad \text{for } j := i \text{ to } n \]
\[ \quad \quad \text{for } k := 1 \text{ to } n \]
\[ \quad \quad \text{if } a_i + a_j = a_k \text{ then return } true \]

\text{return } false

What's the lower bound order of the worst case runtime of this algorithm?
A. $\Omega(1)$
B. $\Omega(n)$
C. $\Omega(n^2)$
D. $\Omega(n^3)$
E. None of the above
Summing Triples: WHEN (2)

\[ \text{SumTriples2}(a_1, \ldots, a_n : \text{real numbers}) \]

\[ \text{for } i := 1 \text{ to } n \quad \Omega \left( \frac{n}{2} \right) \]

\[ \text{for } j := i \text{ to } n \quad \Omega \left( \frac{n}{2} \right) \]

\[ \text{for } k := 1 \text{ to } n \quad \Omega(n) \]

\[ \text{if } a_i + a_j = a_k \text{ then return true} \]

\[ \text{return false} \]

For at least \( n/2 \) values of \( i \) (1 \ldots n/2), we do inner for loop (k) at least n/2 times, each taking n steps.

What's the lower bound order of the worst case runtime of this algorithm?

A. \( \Omega(1) \)
B. \( \Omega(n) \)
C. \( \Omega(n^2) \)
D. \( \Omega(n^3) \)
E. None of the above
Observe: in both these examples, the product rule for calculating the nested loop runtime gave us tight upper bounds … is that always the case?
When is the product rule for nested loops tight?

Nested code:

\[
\text{while (Guard Condition)} \\
\text{Body of the Loop,} \\
\text{May contain other loops, etc.}
\]

If Guard Condition is \(O(1)\) and body of the loop has runtime \(O(T_2)\) in the worst case and run at most \(O(T_1)\) iterations, then runtime is

\[
O(T_1 T_2)
\]

**Example:**

\[
\frac{1}{n} + \frac{1}{n} + \ldots + \frac{1}{n} = \text{linear for } O(n) = 1 + n
\]

But what if many \(t_k\) are much better than the worst case?
Given two lists

\[ a_1, a_2, \ldots, a_n \text{ and } b_1, b_2, \ldots, b_n \]

determine if there are indices i,j such that

\[ a_i = b_j \]

Design an algorithm to look for indices of intersection
Intersecting sorted lists: HOW

Given two lists

\[ a_1, a_2, \ldots, a_n \] and \[ b_1, b_2, \ldots, b_n \]

determine if there are indices i,j such that

\[ a_i = b_j \]

**High-level description:**
- Use linear search to see if \( b_1 \) is anywhere in first list, using early abort
- Since \( b_2 > b_1 \), start the search for \( b_2 \) where the search for \( b_1 \) left off
- And in general, start the search for \( b_j \) where the search for \( b_{j-1} \) left off
Intersect \((a_1, \ldots, a_n, b_1, \ldots, b_n)\)

\[
i := 1
\]

\[
\text{for } j := 1 \text{ to } n
\]

\[
\text{while } (b_j > a_i \text{ and } i \leq n)
\]

\[
i := i + 1
\]

\[
\text{if } i > n \text{ then return } false
\]

\[
\text{if } b_j = a_i \text{ then return } true
\]

return false
Intersecting sorted lists: WHY

\[ \text{Intersect}(a_1, \ldots, a_n, b_1, \ldots, b_n) \]

\[ i := 1 \]

\[ \text{for } j := 1 \text{ to } n \]

\[ \text{while } (b_j > a_i \text{ and } i \leq n) \]

\[ i := i + 1 \]

\[ \text{if } i > n \text{ then return } false \]

\[ \text{if } b_j = a_i \text{ then return } true \]

\[ \text{return } false \]

To practice: trace examples & generalize argument for correctness
Intersecting sorted lists: WHEN

Using product rule

\[ \text{Intersect}(a_1, \ldots, a_n, b_1, \ldots, b_n) \]

\[ i := 1 \]

for \( j := 1 \) to \( n \)

while (\( b_j > a_i \) and \( i \leq n \)) \( O(n) \)

\[ i := i + 1 \]

if \( i > n \) then return false \( O(1) \)

if \( b_j = a_i \) then return true \( O(1) \)

return false
Intersecting sorted lists: WHEN

Using product rule

\[
\text{Intersect}(a_1, \ldots, a_n, b_1, \ldots, b_n)
\]

\[
i := 1
\]

\[
\text{for } j := 1 \text{ to } n
\]

\[
\text{return } \text{false}
\]

\[
\text{Total: } O(n^2)
\]
Intersecting sorted lists: WHEN

More careful analysis ...

\[ \text{Intersect}(a_1, \ldots, a_n, b_1, \ldots, b_n) \]
\[ i := 1 \]
\[ \text{for } j := 1 \text{ to } n \]
\[ \text{while } (b_j > a_i \text{ and } i \leq n) \]
\[ i := i + 1 \]
\[ \text{if } i > n \text{ then return } \text{false} \]
\[ \text{if } b_j = a_i \text{ then return } \text{true} \]
\[ \text{return } \text{false} \]

Every time this is executed (except last time in each iteration of for loop), i is incremented. If i ever reaches n+1, the program terminates (returns)

\[ O(n) \text{ in worst case but not in every case} \]
Intersecting sorted lists: WHEN

More careful analysis ...

\[ \text{Intersect}(a_1, \ldots, a_n, b_1, \ldots, b_n) \]

\[ i := 1 \]

\[ \text{for } j := 1 \text{ to } n \]

\[ \text{while } (b_j > a_i \text{ and } i \leq n) \]

\[ i := i + 1 \]

\[ \text{if } i > n \text{ then return false} \]

\[ \text{if } b_j = a_i \text{ then return true} \]

\[ \text{return false} \]

This executes \(O(2n)\) times total (across all iterations of for loop)
Intersecting sorted lists: WHEN

More careful analysis ...

$$\text{Intersect}(a_1, \ldots, a_n, b_1, \ldots, b_n)$$

$$i := 1$$

for $j := 1$ to $n$

while $(b_j > a_i$ and $i \leq n$)

$$i := i + 1$$

if $i > n$ then return false

if $b_j = a_i$ then return true

return false

Total: $O(n)$

This executes $O(2n)$ times total (across all iterations of for loop)

Linear time

Product rule analysis wasn't tight in this case!
Recursive algorithms (like Merge Sort and Bucket Sort)

- Design
- Analysis
Reminders

HW 3 due **Wednesday 11:59pm** via Gradescope.

**Monday is a holiday.** No lecture, discussion section, office hours.