CSE21 Fall 2016, Day 6

October 5, 2016

http://cseweb.ucsd.edu/classes/fa16/cse21-ab/
Today’s Plan

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

Designing better algorithms
- pre-processing
- re-use of computation
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
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

\[ a_i + a_j = a_k \]

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

SumTriples1(a₁, . . . , aₙ : real numbers)

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

return false

What's the best-case 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)

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

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

What's the best-case runtime of this algorithm?
A. O(1)  
B. O(n)  
C. O(n^2)  
D. O(n^3)  
E. None of the above

Describe all best-case inputs
Summing Triples: HOW (1)

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

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

What's the worst-case runtime of this algorithm?
A. O(1)  
B. O(n)  
C. O(n^2)  
D. O(n^3)  
E. None of the above
Summing Triples: HOW (1)

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

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

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

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

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

return false

Improvements??
Summing Triples: HOW (2)

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

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

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

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

Reframing what we did:

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

```plaintext
for i := 1 to n
    for j := i to n
        For each candidate sum \(a_i + a_j\),
        do linear search to find it
        if \(a_i + a_j = a_k\) then return true

return false
```

Improvements??
**Summing Triples: HOW (2)**

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

\[
\begin{align*}
\text{for } i & := 1 \text{ to } n \\
\text{for } j & := i \text{ to } n \quad \text{For each candidate sum } a_i + a_j, \\
\text{for } k & := 1 \text{ to } n \quad \text{do linear search to find it} \\
\text{if } a_i + a_j = a_k & \quad \text{then return } \text{true} \\
\text{return } \text{false}
\end{align*}
\]

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

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

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

For each candidate sum \(a_i + a_j\),
do binary search to find it.

Worst-case runtime?
A. \(O(n^3)\)
B. \(O(n^2)\)
C. \(O(n^2 \log n)\)
D. \(O(n \log n)\)
Summing Triples: HOW (3)

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

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

Something is wrong!
Summing Triples: HOW (3)

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

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

For each candidate sum \(a_i + a_j\), do binary search to find it

Does this algorithm really work?
This algorithm works! How long does it take?

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

\[ \text{MinSort}(a_1, \ldots, a_n) \]

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

Preprocessing step

aka SortedSumTriples
Summing Triples: HOW (4)

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

$\text{MinSort}(a_1, \ldots, a_n)$ \hspace{1cm} O(n^2)

$\text{SumTriples}_3(a_1, \ldots, a_n)$ \hspace{1cm} O(n^2 \log n)

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

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

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

Have we made progress? Can we do better?

- \text{SumTriples4} does better than \text{O}(n^3).
- Using a faster sort won't help overall.
- Fastest known algorithm: \text{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 $\Theta(g(n))$.

- **Big-O**: upper bound.
- **Big-Ω**: lower bound.
Summing Triples: WHEN (1)

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

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

What's a lower bound on 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 (1)

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

\begin{align*}
\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 \quad \Omega(1) \\
\text{return } false
\end{align*}

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

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

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

What's a lower bound on 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)

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

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

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

```
while (Guard Condition)

Body of the Loop,
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

\[
O(T_1 T_2)
\]

But what if many $t_k$ are much better than the worst case?
Intersecting sorted lists: WHAT

Given two sorted 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
Given two sorted 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

\[ a_i = b_j \]

**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\((a_1, \ldots, a_n, b_1, \ldots, b_n)\)

\[ i := 1 \]

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

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

\[ i := i + 1 \]

if \( i > n \) then return \text{false}

if \( b_j = a_i \) then return true

return \text{false}
Intersect sorted lists: WHY

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

\[ i := 1 \]

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

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

\[ i := i + 1 \]

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

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

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

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

Using product rule

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

\begin{align*}
i &:= 1 \\
\text{for } j &:= 1 \text{ to } n \\
\text{while } (b_j > a_i \text{ and } i \leq n) &:
\begin{align*}
i &:= i + 1
\end{align*}
\text{if } i > n \text{ then return false} \\
\text{if } b_j = a_i \text{ then return true}
\end{align*}

return false
Intersecting sorted lists: WHEN

Using product rule

\[ \text{Intersect}(a_1, \ldots, a_n, b_1, \ldots, b_n) \]
\[ i := 1 \]
\[ \text{for } j := 1 \text{ to } n \]

return false

Total: \( O(n^2) \)
Intersecting sorted lists: WHEN

More careful analysis ...

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

Every time the while loop condition is true, \( i \) is incremented. If \( i \) ever reaches \( n+1 \), the program terminates (returns)
Intersecting sorted lists: WHEN

More careful analysis ...

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

\[ i := 1 \]

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

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

\[ i := i + 1 \]

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

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

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

\begin{align*}
\text{while } (b_j > a_i \text{ and } i \leq n) \\
i := i + 1
\end{align*}

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

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

\text{return false}

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

Total: \(O(n)\)

Be careful: product rule isn't always tight!
Announcements

- HW2 Due Sunday
- Practice with Order Notation on Khan Academy
- Participation Points on TritonEd

Please verify, and if you see an issue, make a private Piazza post to let us know.