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)

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

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

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
Summing Triples: HOW (1)

\[ \text{SumTriples1}(a_1, \ldots, a_n : \text{real numbers}) \]
\[
\text{for } i := 1 \text{ to } n \\
\quad \text{for } j := 1 \text{ to } n \\
\quad 
\quad \text{for } k := 1 \text{ to } n \\
\quad 
\quad \quad \text{if } a_i + a_j = a_k \text{ then return } \text{true} \\
\]
\[
\text{return } \text{false} \\
\]

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

SumTriples1(\(a_1, \ldots, a_n\) : real numbers)

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

What's the \textbf{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)

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

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

Can we do better? How?
Summing Triples: HOW (2)

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

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

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

\begin{verbatim}
for i := 1 to n
    for j := i to n
        for k := 1 to n
            if \( a_i + a_j = a_k \) then return true

return false
\end{verbatim}

What's the \textbf{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 (2)

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

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

Eliminate redundancy

Hmmm...

Can we do better? How?
Summing Triples: HOW (2)

Reframing what we did:

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

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

\[
\text{for } k := 1 \text{ to } n \\
\quad \text{do linear search to find it} \\
\quad \quad \text{if } a_i + a_j = a_k \text{ then return true}
\]

return false

Improvements??
Summing Triples: HOW (2)

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

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

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

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

SumTriples3(a₁, ..., aₙ : real numbers)

```plaintext
for i := 1 to n
    for j := i to n
        For each candidate sum aᵢ+aⱼ,
        if BinarySearch(aᵢ + aⱼ; a₁, ..., aₙ)
            then return true
        return false
```

Worst-case runtime?
A. O(n³)
B. O(n²)
C. O(n² log n)
D. O(n log n)
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.

Something is wrong!
Summing Triples: HOW (3)

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

for $i := 1$ to $n$

for $j := i$ to $n$

if $BinarySearch(a_i + a_j; a_1, \ldots, a_n)$

then return $true$

return $false$

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?

Summing Triples: HOW (4)

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

$MinSort(a_1, \ldots, a_n)$

$SumTriples3(a_1, \ldots, a_n)$

aka $SortedSumTriples$

Preprocessing step
Summing Triples: HOW (4)

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

SumTriples4 worst-case complexity is \text{max} of these: \text{O}(n^2 \log n)
Summing Triples: HOW (4)

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

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

\[ \text{SumTriples3}(a_1, \ldots, a_n) \quad \text{O}(n^2 \log n) \]

Max of these: \text{O}(n^2 \log n)

- \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)

```
SumTriples1(a_1, \ldots, a_n : \text{real numbers})
  for i := 1 to n
    for j := 1 to n
      for k := 1 to n
        if a_i + a_j = a_k then return true
  return 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)

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

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

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

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

for $i := 1$ to $n$

for $j := i$ to $n$

for $k := 1$ to $n$

if $a_i + a_j = a_k$ then return $true$

return $false$

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 (2)

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

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

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

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

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

\[
\text{return false}
\]
Summing Triples: WHEN (2)

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

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

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:

\[
\text{while (Guard Condition)}
\]

Body of the Loop,
May contain other loops, etc.

\[
\sum_{k=1}^{T_1} t_k
\]

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

\[O(T_1T_2)\]

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
Intersecting sorted lists: HOW

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
Intersecting sorted lists: HOW

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

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

\[ i := 1 \]

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

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

\[ \quad \quad i := i + 1 \]

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

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

\[ \text{return } false \]

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

Using product rule

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

\[
i := 1
\]

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

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

\[
i := i + 1
\]

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

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

\[
\text{return false}
\]

\[ O(n) \]

\[ O(1) \]
Intersecting sorted lists: WHEN

Using product rule

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

\[
i := 1
\]

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

\[
\text{return } false
\]

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 ...

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

\[ i := 1 \]

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

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

\[ i := i + 1 \]

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

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

\[ \text{return } false \]

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

More careful analysis ...

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

\[
i := 1
\]

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

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

\[
i := i + 1
\]

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

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

\[
\text{return false}
\]

Total: $\Theta(n)$

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

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

1-1 Signup

Is available...!

HW2 Due tomorrow!
(Tues 1/24, 11:59PM)

Practice with Order Notation on Khan Academy

e.g., https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation