Algorithm Design and Time Analysis

	Miles Jones	**MTThF 8:30-9:50~~pm~~** am	**CSE 4258**

August 5, 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

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)

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

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 the order of the 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)

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

$\quad$ **for** $i := 1$ **to** n

$\quad\quad$ **for** $j := 1$ **to** n

$\quad\quad\quad$ **for** $k := 1$ **to** n

$\quad\quad\quad\quad$ **if** $a_i + a_j = a_k$ **then return** $true$

$\quad$ **return** $false$

Improvements??

Summing Triples: HOW (2)

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

Eliminate redundancy

for $i := 1$ **to** n

  for $j := \cancel{1}\ i$ **to** n

    for $k := 1$ **to** n

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

return $false$

Summing Triples: HOW (2)

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

for $i := 1$ **to** n

for $j := i$ **to** n

Eliminate redundancy

$O(n^2)$

~~**for** $k := 1$ **to** n~~

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

$O(n)$

return $false$

$n + n-1 + n-2 + \cdots + 2 + 1 = \frac{n(n+1)}{2}$

What's the order of the 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)

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

    for $i := 1$ **to** n **Eliminate redundancy**

        for $j := i$ **to** n

            for $k := 1$ **to** n

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

    return $false$

Improvements??

Summing Triples: HOW (2)

Reframing what we did:

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

$\quad$ **for** $i := 1$ **to** n

$\quad\quad$ **for** $j := i$ **to** n

For each candidate sum $a_i + a_j$,

$O(n^2)$

$\quad\quad\quad$ **for** $k := 1$ **to** n

do linear search to find it

$\quad\quad\quad\quad$ **if** $a_i + a_j = a_k$ **then return** $true$

$O(n)$

$\quad$ **return** $false$

Improvements??

Summing Triples: HOW (2)

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

$\quad$ **for** $i := 1$ **to** n

$\quad\quad$ **for** $j := i$ **to** n

For each candidate sum a_i+a_j,

$\quad\quad\quad$ **for** $k := 1$ **to** n

do linear search to find it

$\quad\quad\quad\quad$ **if** $a_i + a_j = a_k$ **then return** $true$

$\quad$ **return** $false$

We have a faster search than linear search!

Summing Triples: HOW (3)

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

for $i := 1$ **to** n

for $j := i$ **to** n

For each candidate sum a_i+a_j,

$O(n^2)$

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

then return $true$

return $false$

do binary search to find it

$O(\log n)$

How long would this take?

A. $O(n^3)$

B. $O(n^2)$

C. $O(n^2 \log n)$

D. $O(n \log n)$

Summing Triples: HOW (3)

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

for $i := 1$ **to** n

for $j := i$ **to** n

For each candidate sum a_i+a_j,

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

then return $true$

return $false$

do binary search to find it

Does this algorithm really work???

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???

Summing Triples: HOW (4)

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

$MinSort(a_1, \ldots, a_n)$ → $O(n^2)$

$SumTriples3(a_1, \ldots, a_n)$

Preprocessing step

aka *SortedSumTriples*

$O(n^2 \log n)$

This algorithm works!
How long does it take?

total runtime $O(\cancel{n^2} + \boxed{n^2 \log n}) = O(n^2 \log n)$

Summing Triples: HOW (4)

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

$\quad MinSort(a_1, \ldots, a_n)$ **O(n^2)**

$\quad SumTriples3(a_1, \ldots, a_n)$ **O(n^2 log n)**

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

Summing Triples: HOW (4)

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

$\quad MinSort(a_1, \ldots, a_n)$ **O(n^2)**

$\quad SumTriples3(a_1, \ldots, a_n)$ **O(n^2 log n)**

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

Have we made progress?
Can we do better?

- *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-Ω : lower bound.

Summing Triples: WHEN (1)

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

$\quad$ **for** $i := 1$ **to** n

$\quad\quad$ **for** $j := 1$ **to** n

$\quad\quad\quad$ **for** $k := 1$ **to** n

$\quad\quad\quad\quad$ **if** $a_i + a_j = a_k$ **then return** $true$

$\quad$ **return** $false$

upper bound

$O(n^3)$

lower bound

$\Omega(n^3)$

$\Theta(n^3)$

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 (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 — $\Omega(n)$

      if $a_i + a_j = a_k$ **then return** $true$ — $\Omega(1)$

return $false$

Strategy: work from the inside out

Summing Triples: WHEN (2)

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

$\frac{n(n+1)}{2}$

for $i := 1$ **to** n

  for $j := i$ **to** n

n+

For the first two loops, we iterate (n-1)+(n-2)+…+2+1 times. For each one of these iterations, the third loop iterates n times.

n

    for $k := 1$ **to** n

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

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)

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

$\quad \textbf{for } i := 1 \textbf{ to } n$

$\quad\quad \textbf{for } j := i \textbf{ to } n$

$\quad\quad\quad \textbf{for } k := 1 \textbf{ to } n$

$\quad\quad\quad\quad \textbf{if } a_i + a_j = a_k \textbf{ then return } true$

$\quad \textbf{return } false$

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.

O(T_2)

$$\sum_{k=1}^{T_1} t_k$$

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

$$O(T_1 T_2)$$

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

Intersecting sorted lists: HOW

Given two sorted lists

$$a_1, \ a_2, \ ..., \ a_n \text{ and } b_1, \ b_2, \ ..., \ b_n$$

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

a_1, a_2, a_3, a_4, a_5

11, 13, 15, 18, 19

10, 12, 16, 17, 20

b_1, b_2, b_3, b_4, b_5

$Intersect(a_1, \ldots, a_n, b_1, \ldots, b_n)$

$i := 1$

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 $false$

$i=1$
$j=1$
$b_1 > a_1$
$j=2$
$b_2 > a_1$
$i=2$
$b_2 > a_2$
$i=3$
$b_3 > a_2$

$j=3$
$b_3 > a_3$
$i=4$
$b_3 > a_4$
$j=4$
$b_4 > a_4$
$b_4 > a_5$
$b_5 > a_5$
$i=6$

★ Exam Intersecting sorted lists: WHY

$Intersect(a_1, \ldots, a_n, b_1, \ldots, b_n)$

$i := 1$

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 $false$

To practice: trace examples & generalize argument for correctness

Intersecting sorted lists: WHEN

Using product rule

$b_1 > a_1$

$b_1 > a_2$

$\vdots$

$b_1 > a_n$

$Intersect(a_1, \ldots, a_n, b_1, \ldots, b_n)$

$\quad i := 1$

$\quad$ **for** $j := 1$ **to** n

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

$\quad\quad\quad i := i + 1$

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

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

$\quad$ **return** $false$

worst case

Intersecting sorted lists: WHEN

Using product rule

$$Intersect(a_1, \ldots, a_n, b_1, \ldots, b_n)$$

$i := 1$

for $j := 1$ **to** n

O(n)

Body
worst case

return $false$

Total: O(n²)

Intersecting sorted lists: WHEN

More careful analysis …

$Intersect(a_1, \ldots, a_n, b_1, \ldots, b_n)$

$i := 1$

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 $false$

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)

Intersecting sorted lists: WHEN

More careful analysis …

$Intersect(a_1, \ldots, a_n, b_1, \ldots, b_n)$

$i := 1$

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 $false$

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

Intersecting sorted lists: WHEN

More careful analysis …

$Intersect(a_1, \ldots, a_n, b_1, \ldots, b_n)$

$i := 1$

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 $false$

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

Total: O(n)

product rule analysis wasn't tight in this case!