CSE 21, Winter 2017, Section A00

Lecture 4 Notes

Class URL: http://vlsicad.ucsd.edu/courses/cse21-w17/
• HW2 due tomorrow 11:59PM
  – “Order.pdf” notes posted last night on Piazza, in case helpful…
• HW1 grades published in Gradescope – 72-hour window for regrade requests
• Discussion Worksheet submissions on Gradescope seem low (?)
  – Please don’t drop 5% of your grade. Collecting 7 out of 8 remaining sessions is very doable.
• Friday 7-8pm problem session was canceled… hopefully, this coming Friday!
  – Friday evening notes for Weeks 1, 2 are posted by Nathan, Andrew, Joseph
• Two more “fun problems” added …

• Today: Day 6 + (part of) Day 7 posted slides == Algorithm Design, Time Analysis, Introduction to Recursion

• Any logistic, other issues ?
Today’s Plan

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

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

If we blindly apply the recursive formula $C(n,k) = C(n,k-1) + C(n-1,k-1)$ to calculate binomial coefficients, this could take a long time! (Try calculating $C(50,25)$ this way!) Pascal’s Triangle shown to the above-right essentially tabulates binomial coefficients starting from $C(1,0) = 1$ and $C(1,1) = 1$ – allowing us to reuse calculations rather than performing them over and over again.

E.g., "one-time sorting"

If we blindly apply the recursive formula $C(n,k) = C(n,k-1) + C(n-1,k-1)$ to calculate binomial coefficients, this could take a long time! (Try calculating $C(50,25)$ this way!) Pascal’s Triangle shown to the above-right essentially tabulates binomial coefficients starting from $C(1,0) = 1$ and $C(1,1) = 1$ – allowing us to reuse calculations rather than performing them over and over again.
Summing Triples: WHAT

• A new problem...

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) \( (i,j,k) \)
C. No

\[ a_1 + a_3 = a_5 \]

\( i = 1, j = 3, k = 5 \)
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 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)

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

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

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

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

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

\[ \text{return false} \]

Describe all best-case inputs…?

\[ a_i = 0 \]

(See previous slide)
Summing Triples: HOW (1)

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

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

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)

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

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

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

return false

Can we do better? How?

- Redundant work
- Only look at \( j \geq i \)
- Looking for \((i, j, k)\) s.t. \(a_i + a_j = a_k\)
- Smarter bounds on \(j\) saves half
Summing Triples: HOW (2)

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

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

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

\[
\begin{align*}
\text{for } i & := 1 \text{ to } n \\
& \quad \text{for } j := i \text{ to } n \\
& \quad \quad \text{for } k := 1 \text{ to } n \\
& \quad \quad \quad \text{if } a_i + a_j = a_k \text{ then return } \text{true} \\
& \quad \quad \Rightarrow \quad \text{return } \text{false} \\
& \approx \frac{1}{2} n^2 \\
& \Rightarrow \quad O\left(n^3\right) \quad \left(\frac{1}{2} n^3\right)
\end{align*}
\]

What's the \textit{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)

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

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

Eliminate redundancy

Hmmmm...

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

Reframing what we did:

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

```
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
```

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

binary search?

Improvements??
Summing Triples: HOW (2)

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

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

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

\[ \text{SumTriples}3(a_1, \ldots, a_n : \text{real numbers}) \]

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

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)

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

```plaintext
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.

*Something is wrong!*

$a_1, \ldots, a_n$ are not necessarily sorted!
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? No, unless/until $a_i$'s are in sorted order.
Summing Triples: HOW (4)

This algorithm works! How long does it take?

Preprocessing step

\[ \text{SumTriples4}(a_1, \ldots, a_n : \text{real numbers}) \]
\[ \text{MinSort}(a_1, \ldots, a_n) \]
\[ \text{SumTriples3}(a_1, \ldots, a_n) \]

\[ O(n^2) \]
\[ O(n^2 \log n) \]

aka SortedSumTriples

Overall

\[ O(n^2 \log n) \]

\[ O(n^2) \]

max-complexity step dominates big-O complexity
Summing Triples: HOW (4)

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

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

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

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

\[ \text{SumTriples4}(a_1, \ldots, a_n : \text{real numbers}) \]
\[ \text{MinSort}(a_1, \ldots, a_n) \]
\[ \text{SumTriples3}(a_1, \ldots, a_n) \]

- \text{SumTriples4} does better than O(n^3).
- Using a faster sort won't help overall.
- Fastest known algorithm: O(n^2)

\[ O(n^2) \rightarrow O(n \log n) \quad \text{(e.g., Mergesort)} \]

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

Lesson: preprocessing helps!
“Tight”?\[\text{To know that we've actually made improvements, need to make sure our original analysis was not overly pessimistic.} \]

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

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

\[\Omega(n^3) \quad \Omega(n^2) \quad \Omega(n) \quad \Omega(1)\]

\[\Omega() \approx \text{“at least”}\]

What's a \textbf{lower bound} on the \textbf{worst-case} runtime of this algorithm?

A. $\Omega(1)$ (not interesting/useful)
B. $\Omega(n)$
C. $\Omega(n^2)$
D. $\Omega(n^3)$
E. None of the above
For algorithms = our solutions to problems [ANALYSES of, e.g., worst-case runtime]

(1) For algorithms = our solutions to problems
    - (and, closing the gap between these)
      - sorting, searching, etc.

(2) For problems themselves
    - matrix mult
      - is $O(n^3)$
      - $\Omega(n^2)$ for matrix mult

- e.g., Sum Triples 2. Merge Sort
  - e.g., output size (writing it down)
Summing Triples: WHEN (1)

How to see $\Omega(n^3)$? (see 2 slides back)

$SumTriples_1(a_1, \ldots, a_n : \text{real numbers})$

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

return false

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

First try (not tight)

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

\[ \Omega(n) \left[ \Omega(i) \left[ \Omega(j) \left[ \Omega(k) \left[ \text{if } a_i + a_j = a_k \text{ then return true} \right] \right] \right] \right] = \Omega(n) \]

Second try: careful counting of times in j loop:

\[ n + (n-1) + (n-2) + \ldots + 2 + 1 = \Omega(n^2) \]

leads to \[ \Omega(n^2) \]

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

loose analysis

from more careful analysis
Summing Triples: WHEN (2)

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

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

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.
Summing Triples: WHEN (2)

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

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

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:

```
while (Guard Condition)  \( T_1 \) iterations
    Body of the Loop,  \( T_2 \) body
    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

\[ O(T_1 T_2) \]

(by Product Rule)

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

Product Rule may not be tight.
Intersecting sorted lists: WHAT

Given two sorted 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
\[
a_i = b_j
\]

Example: \( A = 1, 2, 4, 6 \)
\( B = 3, 5, 6, 7 \)
\( a_4 = b_3 \)
\( i = 4, j = 3 \)

Design an algorithm to look for indices of intersection

First try: \( \text{for } i = 1 \text{ to } n \) \( \text{for } j = 1 \text{ to } n \) \( \Rightarrow O(n^2) \) (try all \( a_i \)’s against all \( b_j \)’s)
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

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

\[ i := 1 \]

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

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

\[ i := i + 1 \]

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

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

return false

A = 1, 2, 4, 6
B = 3, 5, 6, 7

Comparisons:

\[ 1 \sim 3 \]
\[ 2 \sim 3 \]
\[ 4 \sim 3 \]
\[ 4 \sim 5 \]
\[ 6 \sim 5 \]
\[ 6 \sim 6 \]

\[ \Rightarrow \text{ advance } i \]
\[ \Rightarrow \text{ advance } j \]
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

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

\[ i := 1 \]

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

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

\[ i := i + 1 \]

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

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

return false

Key message of this slide: Product Rule not necessarily tight.

“body of for loop is executed \(O(n)\) times”

\(O(n)\) “Could advance \(i\) up to \(\sim n\) times”

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

Using product rule

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

\[ i := 1 \]

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

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

This is an upper bound.
Is it tight?

Total: \( O(n^2) \)

\( O(n) \) times through outer loop
\( \Rightarrow O(n^2) \) total

Really???
Intersecting sorted lists: WHEN

More careful analysis ...

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

\[ i := 1 \]

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

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

\[ i := i + 1 \]

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

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

return false

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

This can’t be \( O(n) \) in all iterations!

After each comparison, either advance \( i \) or advance \( j \)

\( \text{Note: } i, j \text{ can each advance only } O(n) \text{ times!} \)
Intersecting sorted lists: WHEN

More careful analysis ...

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

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

More careful analysis ...

Intersect(a₁, ..., aₙ, b₁, ..., bₙ)

\[ i := 1 \]

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

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

\[ i := i + 1 \]

if \( i > n \) then return false

if \( b_j = a_i \) then return true

return false

\( \text{Total: } \Theta(n) \)

Be careful: product rule isn't always tight!

This executes \( O(n) \) times total (across all iterations of for loop)
A Preview: **MERGING** Two Sorted Lists

Sorted List X: \(x[1], x[2], \ldots, x[m]\)

Sorted List Y: \(y[1], y[2], \ldots, y[n]\)

**Goal:** Merge two sorted lists into one sorted list

“pointer walking” …: if \(y[i] < x[j]\) then compare \(y[i+1]\) and \(x[j]\),
else compare \(y[i]\) and \(x[j+1]\)

After comparing \(x[1]\) and \(y[1]\), we advance in \(x[.]\) and \(y[.]\) a total of \((m-1) + (n-1)\) times

\[\Rightarrow m + n - 1\] comparisons (edges between red and blue) = linear time
MERGESORT Tree of Recursive Subproblems

DIVIDE

CONQUER, COMBINE

\( \log n \) (n comparisons per level) X \( \log n \) (levels) = \( n \log n \) runtime

List sizes at any level = \( n \)
Announcements

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

1-1 Signup  
Is available…!

See also the posted Friday 7pm notes!
End of Day 6 (MWF schedule)
Start of Day 7 (not repeated from Day 6)
(MWF schedule)
Recursion: Introduction and Correctness

CSE21 Winter 2017, Day 7 (B00), Day 4-5 (A00)

January 25, 2017

http://vlsicad.ucsd.edu/courses/cse21-w17
Today’s Plan

From last time:
● intersecting sorted lists and tight bounds

New topic: Recursion
● recursive algorithms
● correctness of recursive algorithms
● solving recurrence relations

e.g., “divide and conquer” algorithms
● matrix multiplication
● FFT
● sorting
● long-number multiplication
● closest pair
● median-finding
What is recursion?

- Self-referential
- Method that calls itself

```python
factorial(n)
  if (n==1) return 1
  result = factorial(n-1) * n
  return result
```
What is recursion?

Solving a problem by successively reducing it to the same problem with smaller inputs.

Rosen p. 360
Strings and substrings

A string is a finite sequence of symbols such as 0s and 1s, written as $b_1 b_2 b_3 \ldots b_n$.

A substring of length $k$ contains $k$ consecutive symbols of the string, $b_i b_{i+1} b_{i+2} \ldots b_{i+k-1}$.

In how many places can we find 010 as a substring of 0100101000?

A. 1
B. 2
C. 3
D. 4
Problem: Given a string of 0s and 1s

\[ b_1 \ b_2 \ b_3 \ \ldots \ b_n \]

count how many times the substring \(00\) occurs in the string.
Counting a pattern: HOW

Problem: Given a string of 0s and 1s

\[ b_1 \ b_2 \ b_3 \ \ldots \ b_n \]

count how many times the substring 00 occurs in the string.

*Design an algorithm to solve this problem*
Counting a pattern: HOW

An **Iterative Algorithm**:
Step through each position and see if pattern starts there.

```
procedure countDoubleIter(b₁, ..., bₙ : each 0 or 1)
    count := 0
    if n < 2 then return 0
    for i := 1 to n - 1
        if (bᵢ = 0 and bᵢ₊₁ = 0) then
            count := count + 1
    return count
```
A **Recursive Algorithm:**
Does pattern occur at the head? Then solve for the rest.

```plaintext
procedure countDoubleRec(b₁, . . . , bₙ : each 0 or 1)
    if n < 2 then return 0
    if (b₁ = 0 and b₂ = 0) then return 1 + countDoubleRec(b₂, . . . , bₙ)
    return countDoubleRec(b₂, . . . , bₙ)
```
Recursive vs. Iterative

This example shows that essentially the same algorithm can be described as iterative or recursive.

But describing an algorithm recursively can give us new insights and sometimes lead to more efficient algorithms.

It also makes correctness proofs more intuitive.
Induction and recursion

**Induction**

A proof strategy where we prove

- The base case
- How to prove the statement is true about \(n+1\) if we get to assume that it is true for \(n\).

**Recursion**

A way of solving a problem where we must give

- The base case
- How to solve a problem of size \(n+1\), assuming we can solve a problem of size \(n\).