Algorithm design and time analysis

<table>
<thead>
<tr>
<th>Lecture A</th>
<th>Tiefenbruck</th>
<th>Tu, Th 11am-12:20pm</th>
<th>Center 119</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lecture B</td>
<td>Tiefenbruck</td>
<td>Tu, Th 9:30am-10:50am</td>
<td>Center 119</td>
</tr>
<tr>
<td>Lecture C</td>
<td>Minnes</td>
<td>Tu, Th 3:30pm-4:50pm</td>
<td>WLH 2005</td>
</tr>
</tbody>
</table>

http://cseweb.ucsd.edu/classes/fa15/cse21-abc/

Oct 6, 2015
Today's plan

1. Analyze algorithms using asymptotic (order) notation

2. Find better algorithms
   • How much better?

3. Observe algorithm design principles
   • Pre-processing
   • Re-use of computation
Computing the big-O class of algorithms

How to deal with …

Basic operations

Consecutive (non-nested) code

Loops (simple and nested)

Subroutines
Computing the big-O class of algorithms

How to deal with …

**Basic operations** : operation whose time doesn't depend on input

**Consecutive (non-nested) code** : one algorithm followed by another

**Loops (simple and nested)** : while loops, for loops

**Subroutines** : method calls
Consecutive (non-nested) code: Run $\text{Prog}_1$ followed by $\text{Prog}_2$

If $\text{Prog}_1$ takes $O(f(n))$ time and $\text{Prog}_2$ takes $O(g(n))$ time, what's the big-O class of runtime for running them consecutively?

A. $O(f(n) + g(n))$
B. $O(f(n) \cdot g(n))$
C. $O(g(f(n)))$
D. $O(\max(f(n), g(n)))$
E. None of the above.
Computing the big-O class of algorithms

Simple loops: \texttt{while} (Guard Condition) \\
\hspace{1cm} Body of the Loop

What's the runtime?

A. Constant
B. Same order as the number of iterations through the loop.
C. Same order as the runtime of the guard condition
D. Same order as the runtime of the body of the loop.
E. None of the above.
Computing the big-O class of algorithms

Simple loops:

\[
\text{while (Guard Condition)} \\
\quad \text{Body of the Loop}
\]

If Guard Condition uses basic operations and body of the loop is constant time, then runtime is of the same order as the number of iterations.
Computing the big-O class of algorithms

Nested code:

```
while (Guard Condition) {
    Body of the Loop,
    May contain other loops, etc.
}
```

If Guard Condition uses basic operations and body of the loop has constant-time runtime $O(T_2)$ in the worst case, then runtime is

$$O(T_1T_2)$$

where $T_1$ is the bound on the number of iterations through the loop.

Product rule
Subroutine  Call method S on some part of the input.

If sub-routine S has runtime $T_S(n)$ and we call S at most $T_1$ times,

A. Total time for all uses of S is $T_1 + T_S(n)$
B. Total time for all uses of S is $\max(T_1, T_S(n))$
C. Total time for all uses of S is $T_1 T_S(n)$
D. None of the above
Subroutine Call method S on some part of the input.

If sub-routine S has runtime is $O \left( T_S(n) \right)$ and if we call S at most $T_1$ times, then runtime is

$$O \left( T_1 T_S(m) \right)$$

where $m$ is the size of biggest input given to S.

*Distinguish between the size of input to subroutine, $m$, and the size of the original input, $n$, to main procedure!*
Selection Sort (MinSort) Pseudocode

Before, we counted comparisons, and then went to big-O

procedure selection sort(a₁, a₂, ..., aₙ: real numbers with n >= 2 )
for i := 1 to n-1
    m := i
    for j := i+1 to n
        if (aⱼ < aᵢ) then m := j
    interchange aᵢ and aₘ

{ a₁, ..., aₙ is in increasing order}
procedure selection sort(a_1, a_2, ..., a_n: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( a_j < a_i ) then m := j
    interchange a_i and a_m

{ a_1, ..., a_n is in increasing order}
Selection Sort (MinSort) Pseudocode

Now, straight to big O

procedure selection sort(a₁, a₂, ..., aₙ: real numbers with n ≥ 2)
for i := 1 to n-1
    m := i
    for j := i+1 to n
        if (aⱼ < aᵢ) then m := j
    interchange aᵢ and aₘ

{ a₁, ..., aₙ is in increasing order}

Strategy: work from the inside out
procedure selection sort(a_1, a_2, ..., a_n: real numbers with n \geq 2)
for i := 1 to n-1
    m := i
    for j := i+1 to n
        if (a_j < a_i) then m := j
    interchange a_i and a_m
{ a_1, ..., a_n is in increasing order}

Strategy: work from the inside out
procedure selection sort(a_1, a_2, ..., a_n: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( a_j < a_i ) then m := j
    interchange a_i and a_m

{ a_1, ..., a_n is in increasing order}

Strategy: work from the inside out

Now, straight to big O

O(n-i),
but i ranges from 1 to n-1
procedure selection sort(a_1, a_2, ..., a_n: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( a_j < a_i ) then m := j
    interchange a_i and a_m
{ a_1, ..., a_n is in increasing order}

Worst case: when i =1, O(n)

Strategy: work from the inside out
Selection Sort (MinSort) Pseudocode

Now, straight to big O

procedure selection sort \((a_1, a_2, \ldots, a_n: \text{real numbers with } n \geq 2)\)
for \(i := 1\) to \(n-1\)

\(m := i\)

\(O(1)\)

\(O(n)\)
interchange \(a_i\) and \(a_m\)

\(O(1)\)

\{ \(a_1, \ldots, a_n\) is in increasing order\}

Strategy: work from the inside out
Selection Sort (MinSort) Pseudocode

Now, straight to big O

procedure selection sort(a₁, a₂, ..., aₙ: real numbers with n >=2 )
for i := 1 to n-1

{ a₁, ..., aₙ is in increasing order}
Selection Sort (MinSort) Pseudocode

```plaintext
procedure selection sort(a_1, a_2, ..., a_n: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if (a_j < a_i) then
            m := j
    interchange a_i and a_m
{ a_1, ..., a_n is in increasing order}
```

Now, straight to big O

Strategy: work from the inside out

```plaintext
O(n)
Nested for loop, repeats O(n) times
Total: O(n^2)
```
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

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

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)

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

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

\text{return } \text{false}

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)

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

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

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

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
Summing Triples: HOW (2)

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

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

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 (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

Eliminate redundancy

Improvements??
Summing Triples: HOW (3)

Reframing what we did:

\[ \text{SumTriples}_2(a_1, \ldots, a_n : \text{real numbers}) \]

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

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

do linear search to find it

Improvements??
Summing Triples: HOW (3)

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

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

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)

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

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

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

Do binary search to find it.

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)

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

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

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

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 \\
\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?

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

aka SortedSumTriples

Preprocessing step
Summing Triples: HOW (4)

\[ \text{SumTriples}4(a_1, \ldots, a_n : \text{real numbers}) \]
\[ \text{MinSort}(a_1, \ldots, a_n) \quad O(n^2) \]
\[ \text{SumTriples}3(a_1, \ldots, a_n) \quad O(n^2 \log n) \]

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

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

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

$SumTriples3(a_1, \ldots, a_n)$ \hspace{1cm} $O(n^2 \log n)$

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.

The **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: How (1)

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

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

What's the lower bound order of the 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: How (1)

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

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

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

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

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

return false

\textbf{Strategy: work from the inside out}
Summing Triples: How (2)

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

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

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

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

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

return false

What's the lower bound order of the 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: How (2)

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

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

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

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: How (2)

\[ \text{SumTriples2}(a_1, \ldots, a_n : \text{real numbers}) \]
\[ \text{for } i := 1 \text{ to } n \]
\[ \text{for } j := i \text{ to } n \]
\[ \text{for } k := 1 \text{ to } n \]
\[ \text{if } a_i + a_j = a_k \text{ then return } \text{true} \]
\[ \text{return } \text{false} \]

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:**

```plaintext
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

$$O(T_1 T_2)$$

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

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
Example Intersecting sorted arrays: HOW

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
\[ a_i = b_j \]

**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
Example Intersecting sorted arrays: HOW

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

\(i := 1\)

\textbf{for} \(j := 1 \textbf{ to } n\)

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

\(i := i + 1\)

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

\textbf{if} \(b_j = a_i\) \textbf{ then return } true

\textbf{return } false
Example Intersecting sorted arrays: 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
Example Intersecting sorted arrays: WHEN

Using product rule

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

\[
i := 1
\]

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

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

\[
i := i + 1
\]

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

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

return false
Example Intersecting sorted arrays: WHEN

Using product rule

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

\[ i := 1 \]

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

\[ \text{O(n)} \]

\[ \text{return } false \]

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

More careful analysis ...

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

return \text{false}
Example Intersecting sorted arrays: WHEN

More careful analysis ...

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

\[i := 1\]

\[
\text{for } j := 1 \text{ to } n \\
\text{while } (b_j > a_i \text{ and } i \leq n) \\
\quad i := i + 1 \\
\text{if } i > n \text{ then return } false \\
\text{if } b_j = a_i \text{ then return } true \\
\text{return } false
\]
Example Intersecting sorted arrays: 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: \(O(n)\)

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

Product rule analysis wasn't tight in this case!
Next Time…

Recursive algorithms

- Design
- Analysis
Reminders

HW 3 due **Friday 11:59pm**.

Office hours all this week.

Assignment submission through **Gradescope** (see Piazza).