CS221: Algorithms and Data Structures

Analyzing Runtime

Alan J. Hu
(Borrowing many slides from Steve Wolfman)

Types of analysis

Orthogonal axes

- bound flavor
  - upper bound (O)
  - lower bound (Ω)
  - asymptotically tight (Θ)
- analysis case
  - worst case (adversary)
  - average case
  - best case
  - “common” case
- analysis quality
  - loose bound (any true analysis)
  - tight bound (no better bound which is asymptotically different)

Types of analysis

Orthogonal axes

- bound flavor
  - upper bound (O)
  - lower bound (Ω)
  - asymptotically tight (Θ)
- analysis case
  - worst case (adversary)
  - average case
  - best case
  - "common" case
- analysis quality
  - loose bound (any true analysis)
  - tight bound (no better bound which is asymptotically different)

WTF?!?

“Tight” Bounds

- Big-O and Big- are upper and lower bounds.
- *Any* upper or lower bound makes a true statement, e.g.,:
  - “Insertion sort runs in time $O(n^{1000})$.” is a true statement!
  - But it’s not very useful…
- We’d like a way to say that we have a ***good*** upper or lower bound. This is called a “tight” bound.

“Tight” Bound

There are at least three common usages for calling a bound “tight”:

1. Big-Theta, “asymptotically tight”
2. “no better bound which is asymptotically different”
3. Big-O upper bound on run time of an algorithm matches provable worst-case lower-bound on any solution algorithm.

“Tight” Bound – Def. 1

1. Big-Theta, “asymptotically tight”

This definition is formal and clear:

$T(n) \in \Theta(f(n))$ if $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$

but it is too rigid to capture practical intuition.

For example, what if T(n) = `(n%2==0)? n*n : 1`

Is $T(n) \in O(n^2)$?

Is $T(n) \in \Theta(n^2)$?

“Tight” Bound – Def. 2

2. “no better bound which is asymptotically different”

This is the most common definition, and captures what people usually want to say, but it’s not formal.

E.g., given same $T(n)$, we want $T(n) \in O(n^2)$ to be considered “tight”, but not $T(n) \in O(n^3)$

But, $T(n)$ is NOT $\Theta(n^2)$, so isn’t $T(n) \in O(T(n))$ a tighter bound?

“Tight” Bound – Def. 2

2. “no better \`reasonable’ bound which is asymptotically different”

This is the most common definition, and captures what people usually want to say, but it’s not formal.

E.g., given same T(n), we want $T(n) \in O(n^2)$ to be considered “tight”, but not $T(n) \in O(n^3)$

But, T(n) is NOT $\Theta(n^2)$, so isn’t $T(n) \in O(T(n))$ a tighter bound?

“Tight” Bound – Def. 2

2. “no better `reasonable’ bound which is asymptotically different”

This is the most common definition, and captures what people usually want to say, but it’s not formal.

“Reasonable” is defined subjectively, but it basically means a simple combination of normal, common functions, i.e., anything on our list of common asymptotic complexity categories (e.g., log n, n, n^k, 2^n, n!, etc.). There should be no lower-order terms, and no unnecessary coefficients.

“Tight” Bound – Def. 2

2. “no better `reasonable’ bound which is asymptotically different”

This is the most common definition, and captures what people usually want to say, but it’s not formal.

E.g., given same T(n), we want $T(n) \in O(n^2)$ to be considered “tight”, but not $T(n) \in O(n^3)$

This is the definition we’ll use in CPSC 221 unless stated otherwise.

“Tight” Bound – Def. 3

3. Big-O upper bound on run time of an algorithm matches provable lower-bound on any algorithm.

The definition used in more advanced, theoretical computer science:

- Upper bound is on a specific algorithm.
- Lower bound is on the problem in general.
- If the two match, you can’t get an asymptotically better algorithm.

This is beyond this course, for the most part.

(Example: Sorting…)

"Tight (Def. 3)" Bound for Sorting

- We'll see later that you can sort n numbers in $O(n \log n)$ time. Is it possible to do better?
- The answer is no (if you know nothing about the numbers and rely only on comparisons):
  - How many different ways can you arrange n numbers?
  - A sorting algorithm must distinguish between these n! choices (because any of them ***might*** be the input).
  - Each comparison can cut the set of possibilities in half.
  - So, to distinguish which of the n! orders you were input requires $\lg(n!)$ comparisons.
  - $\lg(n!)$ is $\Theta(n \log n)$

"Tight" Bound – Def. 2

2. "no better `reasonable' bound which is asymptotically different"

This is the most common definition, and captures what people usually want to say, but it's not formal.

E.g., given same T(n), we want $T(n) \in O(n^2)$ to be considered "tight", but not $T(n) \in O(n^3)$

This is the definition we'll use in CPSC 221 unless stated otherwise.

Analyzing Code

- C++ operations - constant time
- consecutive stmts - sum of times
- conditionals - max/sum of branches, plus condition
- loops - sum of iterations
- function calls - cost of function body

Above all, use your head!

Analyzing Code

```
// Linear search
find(key, array)
  for i = 1 to length(array) - 1 do
    if array[i] == key
      return i
  return -1
```

Step 1: What's the input size **n**?

Analyzing Code

```
// Linear search
find(key, array)
  for i = 1 to length(array) - 1 do
    if array[i] == key
      return i
  return -1
```

Step 2: What kind of analysis should we perform?
Worst-case? Best-case? Average-case?

Expected-case, amortized, …

Analyzing Code

```
// Linear search
find(key, array)
  for i = 1 to length(array) - 1 do
    if array[i] == key
      return i
  return -1
```

Step 3: How much does each line cost? (Are lines the right unit?)

Analyzing Code

```
// Linear search
find(key, array)
  for i = 1 to length(array) - 1 do
    if array[i] == key
      return i
  return -1
```

Step 4: What's `T(n)` in its raw form?

Analyzing Code

```
// Linear search
find(key, array)
  for i = 1 to length(array) - 1 do
    if array[i] == key
      return i
  return -1
```

Step 5: Simplify **T(n)** and convert to order notation. (Also, which order notation: O, Θ, Ω?)

Analyzing Code

```
// Linear search
find(key, array)
  for i = 1 to length(array) - 1 do
    if array[i] == key
      return i
  return -1
```

Step 6: Casually name-drop the appropriate terms in order to sound bracingly cool to colleagues: "Oh, linear search? That's tractable, polynomial time. What polynomial? Linear, duh. See the name?! I hear it's sub-linear on quantum computers, though. Wild, eh?"

Analyzing Code

```
// Linear search
find(key, array)
  for i = 1 to length(array) - 1 do
    if array[i] == key
      return i
  return -1
```

Step 7: **Prove** the asymptotic bound by finding constants $\mathbf{c}$ and $\mathbf{n_0}$ such that for all $\mathbf{n \geq n_0}$, $\mathbf{T(n) \leq cn}$.

You usually won't do this in practice.

More Examples Than You Can Shake a Stick At (#0)

```
// Linear search
find(key, array)
  for i = 1 to length(array) - 1 do
    if array[i] == key
      return i
  return -1
```

Here's a whack-load of examples for us to:

1. find a function **`T(n)`** describing its runtime
2. find **`T(n)`**'s asymptotic complexity
3. find **`c`** and $\mathbf{n_0}$ to prove the complexity

METYCSSA (#1)

```
for i = 1 to n do
  for j = 1 to n do
    sum = sum + 1
```

Time complexity:

a. $O(n)$
b. $O(n \lg n)$
c. $O(n^2)$
d. $O(n^2 \lg n)$
e. None of these

METYCSSA (#2)

```
i = 1
while i < n do
  for j = i to n do
    sum = sum + 1
  i++
```

Time complexity:

a. O(n)
b. O(n lg n)
c. $O(n^2)$
d. $O(n^2 \lg n)$
e. None of these

Three METYCSSA2 Approaches: Pure Math

`i = 1`	`takes "1" step`
`while i < n do`	`i varies 1 to n-1`
`  for j = i to n do`	`j varies i to n`
`    sum = sum + 1`	`takes "1" step`
`  i++`	`takes "1" step`

Now, we write a function T(n) that adds all of these up, summing over the iterations of the two loops:

$$T(n) = 1 + \sum_{i=1}^{n-1} \left(1 + \sum_{j=i}^{n} 1 \right)$$

Three METYCSSA2 Approaches: Pure Math

Here's our function for the runtime of the code:

$$T(n) = 1 + \sum_{i=1}^{n-1}\left(1 + \sum_{j=i}^{n} 1\right)$$

Summing 1 for *j* from *i* to *n* is just going to be 1 added together (n-i+1) times, which is (n-i+1):

$$T(n) = 1 + \sum_{i=1}^{n-1} 1 + n - i + 1 = 1 + \sum_{i=1}^{n-1} n - i + 2$$

Three METYCSSA2 Approaches: Pure Math

Here's our function for the runtime of the code:

$$T(n) = 1 + \sum_{i=1}^{n-1} 1 + n - i + 1 = 1 + \sum_{i=1}^{n-1} n - i + 2$$

The n and 2 terms don't change as i changes. So, we can pull them out (and multiply by the number of times they're added):

$$T(n) = 1 + n(n-1) + 2(n-1) - \sum_{i=1}^{n-1} i$$

And, we know that $\sum_{i=1}^{k} i = k(k+1)/2$, so:

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

Three METYCSSA2 Approaches: Pure Math

Here's our function for the runtime of the code:

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

$$= n^2 + n - 1 - \frac{n^2}{2} + \frac{n}{2} = \frac{n^2}{2} + \frac{n}{2} - 1$$

So, $T(n) = \frac{n^2}{2} + \frac{n}{2} - 1$.

Drop low-order terms and the ½ coefficient, and we find:
$T(n) \in \Theta(n^2)$.

Yay!!!

Three METYCSSA2 Approaches: Faster Code/Slower Code

```
i = 1                              takes "1" step
while i < n do                     i varies 1 to n-1
  for j = i to n do                j varies i to n
    sum = sum + 1                  takes "1" step
  i++                              takes "1" step
```

This code is "too hard" to deal with. So, let's find *just* an upper bound.

In which case we get to change the code so in any way that makes it run no faster (even if it runs slower).

We'll let j go from 1 to n rather than i to n. Since $i \geq 1$, this is no *less* work than the code was already doing…

Three METYCSSA2 Approaches: Faster Code/Slower Code

```
i = 1                         takes "1" step
while i < n do                goes n-1 times
  for j = 1 to n do           goes n times
    sum = sum + 1             takes "1" step
  i++                         takes "1" step
```

Now, each iteration of each loop body takes the same amount of time as the next iteration, and we get:

$$T(n) = 1 + (n-1)(1+n) = 1 + n^2 - 1 = n^2$$

Clearly, $T(n) \in O(n^2)$!

BUT, that's just an upper-bound (big-O), since we changed the code, possibly making it run **slower**.

Three METYCSSA2 Approaches: Faster Code/Slower Code

```
i = 1                          takes "1" step
while i < n do                 i varies 1 to n-1
  for j = i to n do            j varies i to n
    sum = sum + 1              takes "1" step
  i++                          takes "1" step
```

Let's do a lower-bound, in which case we can make the code run *faster* if we want. The trouble is that j starts at i. If it started at $n - 1$, we wouldn't have to worry about i… but we'd get an $\Omega(n)$ bound, which is lower than we'd like.

We can't start j at something nice like $n/2$ because i grows larger than $n/2$. So, let's keep i from growing so large!

Three METYCSSA2 Approaches: Faster Code/Slower Code

```
i = 1                              takes "1" step
while i < n/2 + 1 do               goes n/2 times
  for j = i to n do                j varies i to n
    sum = sum + 1                  takes "1" step
  i++                              takes "1" step
```

We used $n/2 + 1$ so the outer loop will go exactly $n/2$ times.

Now we can start j at $n/2 + 1$, knowing that j will never get that large, so we're certainly not making the code slower!

Three METYCSSA2 Approaches: Faster Code/Slower Code

`i = 1`	takes "1" step
`while i < n/2 + 1 do`	goes n/2 times
`  for j = n/2 + 1 to n do`	goes n/2 times
`    sum = sum + 1`	takes "1" step
`  i++`	takes "1" step

Again, the loop bodies take the same amount of time from one iteration to the next, and we get:

$$T(n) = 1 + \frac{n}{2}\left(\frac{n}{2} + 1\right) = 1 + \frac{n^2}{4} + \frac{n}{2}$$

Droping low-order terms and constant coefficients:

$T(n) \in \Omega(n^2)$

Yay!!!

Three METYCSSA2 Approaches: Pretty Pictures!

```
i = 1                        takes “1” step
while i < n do               i varies 1 to n-1
  for j = i to n do          j varies i to n
    sum = sum + 1            takes “1” step
  i++                        takes “1” step
```

Imagine drawing one point for each time the gets executed. In the first iteration of the outer loop, you’d draw *n* points. In the second, *n*-1. Then *n*-2, *n*-3, and so on down to (about) 1. Let’s draw that picture…

Three METYCSSA2 Approaches: Pretty Pictures!

```
* * * * * * * * * *
  * * * * * * * * *
    * * * * * * * *
      * * * * * * *
        * * * * * *
          * * * * *
            * * * *
              * * *
                * *
                  *
```

It’s a triangle, and its area is proportional to runtime

Three METYCSSA2 Approaches: Pretty Pictures!

about n columns

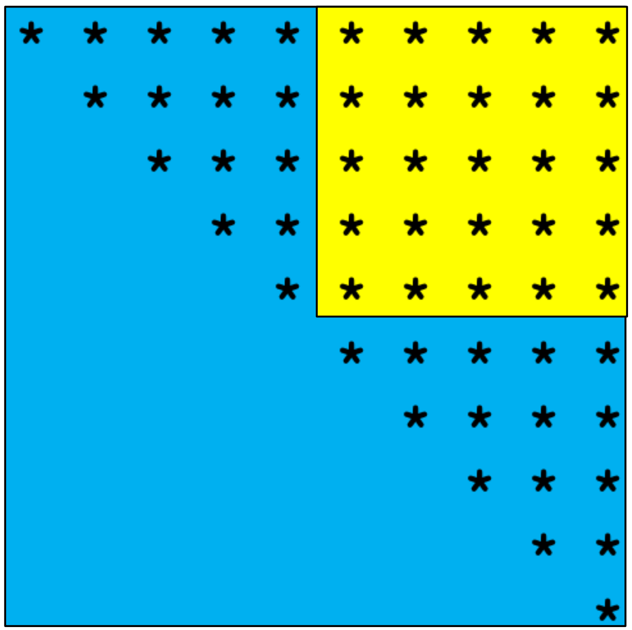
```
* * * * * * * * * *
  * * * * * * * * *
    * * * * * * * *
      * * * * * * *
        * * * * * *
          * * * * *
            * * * *
              * * *
                * *
                  *
```

about n rows

It's a triangle, and its area is proportional to runtime:

$$T(n) = \frac{base * height}{2} = \frac{n^2}{2} \in \Theta(n^2)$$

Note: Pretty Pictures and Faster/Slower are the Same(ish)



Both the overestimate (upper-bound) and underestimate (lower-bound) are squares with sides proportional to n (area proportional to n^2). So, it's $\Theta(n^2)$

METYCSSA (#3)

```
i = 1
while i < n do
  for j = 1 to i do
    sum = sum + 1
  i += i
```

Time complexity:

a. O(n)
b. O(n lg n)
c. $O(n^2)$
d. $O(n^2 \lg n)$
e. None of these

METYCSSA (#4)

- Conditional

  if C **then** S_1 **else** S_2

- Loops

  while C **do** S

METYCSSA (#5)

- Recursion almost always yields a *recurrence*
- Recursive max:

```
if length == 1: return arr[0]
else: return larger of arr[0] and max(arr[1..length-1])
```

T(1) <= b

T(n) <= c + T(n - 1) *if* **n > 1**

- Analysis

T(n) <= c + c + T(n - 2) (by substitution)

T(n) <= c + c + c + T(n - 3) (by substitution, again)

T(n) <= kc + T(n - k) (extrapolating **0 < k** ≤ **n**)

T(n) <= (n – 1)c + T(1) = (n – 1)c + b (for **k = n - 1**)

- **T(n)** ∈

METYCSSA (#6): Mergesort

- Mergesort algorithm
  - split list in half, sort first half, sort second half, merge together
- $T(1) \le b$

  $T(n) \le 2T(n/2) + cn$ *if* $n > 1$
- Analysis

$$
\begin{aligned}
T(n) &\le 2T(n/2) + cn \\
&\le 2(2T(n/4) + c(n/2)) + cn \\
&= 4T(n/4) + cn + cn \\
&\le 4(2T(n/8) + c(n/4)) + cn + cn \\
&= 8T(n/8) + cn + cn + cn \\
&\le 2^k T(n/2^k) + kcn && \text{(extrapolating } 1 < k \le n) \\
&\le nT(1) + cn \lg n && \text{(for } 2^k = n \text{ or } k = \lg n)
\end{aligned}
$$

- $T(n) \in$

METYCSSA (#7): Fibonacci

- Recursive Fibonacci:

```
int Fib(n)
  if (n == 0 or n == 1) return 1
  else return Fib(n - 1) + Fib(n - 2)
```

- *Lower* bound analysis
- `T(0), T(1) >= b`

  $T(n) \geq T(n-1) + T(n-2) + c$ *if* $n > 1$
- Analysis

  let ϕ be $(1 + \sqrt{5})/2$ which satisfies $\phi^2 = \phi + 1$

  show by induction on n that $T(n) \geq b\phi^{n-1}$

Example #7 continued

- Basis: $\mathbf{T(0)} \geq \mathbf{b} > \mathbf{b}\phi^{-1}$ and $\mathbf{T(1)} \geq \mathbf{b} = \mathbf{b}\phi^{0}$
- Inductive step: Assume $\mathbf{T}(m) \geq \mathbf{b}\phi^{m-1}$ for all $m < \mathbf{n}$

$$
\begin{aligned}
\mathbf{T}(n) &\geq \mathbf{T}(n-1) + \mathbf{T}(n-2) + \mathbf{c} \\
&\geq \mathbf{b}\phi^{n-2} + \mathbf{b}\phi^{n-3} + \mathbf{c} \\
&\geq \mathbf{b}\phi^{n-3}(\phi + 1) + \mathbf{c} \\
&= \mathbf{b}\phi^{n-3}\phi^{2} + \mathbf{c} \\
&\geq \mathbf{b}\phi^{n-1}
\end{aligned}
$$

- $\mathbf{T(n)} \in$
- Why? Same recursive call is made numerous times.

Example #7: Learning from Analysis

- To avoid recursive calls
  - store all basis values in a table
  - each time you calculate an answer, store it in the table
  - before performing any calculation for a value ***n***
    - check if a valid answer for ***n*** is in the table
    - if so, return it
- This strategy is called "*memoization*" and is closely related to "*dynamic programming*"

- How much time does this version take?

Final Concrete Example (#8): Longest Common Subsequence

- Problem: given two strings (*m* and *n*), find the longest sequence of characters which appears in order in both strings
  - lots of applications, DNA sequencing, blah, blah, blah

- Example:
  - "**`search me`**" and "**`insane method`**" = "**`same`**"

Abstract Example (#9): It's Log!

Problem: find a tight bound on **`T(n) = lg(n!)`**

Time complexity:

a. $O(n)$
b. $O(n \lg n)$
c. $O(n^2)$
d. $O(n^2 \lg n)$
e. None of these

“Tight (Def. 3)” Bound for Sorting

- We’ll see later that you can sort n numbers in O(n log n) time. Is it possible to do better?
- The answer is no (if you know nothing about the numbers and rely only on comparisons):
  - How many different ways can you arrange n numbers?
  - A sorting algorithm must distinguish between these n! choices (because any of them ***might*** be the input).
  - Each comparison can cut the set of possibilities in half.
  - So, to distinguish which of the n! orders you were input requires lg(n!) comparisons.
  - lg(n!) is Θ(n log n)