CSE 250 - Data Structures: Function Analysis

Andrew Hughes

SUNY at Buffalo
Computer Science and Engineering

Fall 2019

Outline

How do we measure the “cost” of code?

Code cost can be modeled by:

- the number of steps executed, or
- the number bytes allocated.

Cost modeled as a function of input size.

Asymptotics used to compare cost functions.

- How long does my function take to run?
- How much memory is used by my function?
- Is there a better solution?

Growth Function Basics

Compare the following functions:

$$f(n) = 2^{\frac{n}{2}}, g(n) = \frac{1}{2}\log(n^2), h(n) = \log(n), S(n) = 5n,$$

$$T(n) = \begin{cases} 2T(\frac{n}{2}) \text{ if } n > 1 \\ 1 \text{ if } n = 1 \end{cases}, \quad U(n) = \begin{cases} 2U(n-1) \text{ if } n > 1 \\ 1 \text{ if } n = 1 \end{cases}$$

Ranked by growth:

Fastest growth	$U(n) = \begin{cases} 2U(n-1) \text{ if } n > 1 \\ 1 \text{ if } n = 1 \end{cases}$, or $U(n) = 2^n$
	$f(n) = 2^{\frac{n}{2}}$
	$S(n) = 5n$
	$T(n) = \begin{cases} 2T(\frac{n}{2}) \text{ if } n > 1 \\ 1 \text{ if } n = 1 \end{cases}$, or $T(n) \approx n$
Slowest growth	$g(n) = \frac{1}{2}\log(n^2) = \log(n) = h(n)$

Growth Function Basics

Growth functions must be of the form:

$$f : \mathbb{Z}^+ \to \mathbb{R}^+$$

- Inputs to programs are all discrete values.
  - $\mathbb{Z}^+ = \{1, 2, \ldots\}$ (positive integers).
- Measurements can be fractional.
  - $\mathbb{R}^+ = \{x \mid x > 0\}$ (positive real numbers).

Log Refresher

$\log$ is usually base 2 in this course: $\log(n) = \log_2(n)$.

- Different base may appear in recursive analysis.
  (based on number of subproblem divisions)

Revisit log properties if you have forgotten.

- Let $a, b, c, n > 0$.
- Exponent rule: $\log(n^a) = a\log(n)$
- Product rule: $\log(an) = \log(a) + \log(n)$
- Division rule: $\log(\frac{n}{a}) = \log(n) - \log(a)$
- Change of base from b to c: $\log_b(n) = \frac{\log_c(n)}{\log_c(b)}$
  - Different base is only off by a constant factor.
- Logs/exponentiation are inverses: $b^{\log_b(n)} = \log_b(b^n) = n$

Architecture Independent

Compare two algorithms Algorithm 1 and Algorithm 2:

- Algorithm 1 performs $f_1(n)$ ops given an input of size n where:
  - $f_1(n) = 10000n^2$
- Algorithm 2 performs $f_2(n)$ ops given an input of size n where:
  - $f_2(n) = 10n^2$.

Which algorithm is better?

Algorithm 2 runs 1000 times faster.

- This might matter in practice.
- This is less relevant theoretically.
  - Both are "the same" for large enough n.

Algorithm 1 may have other practical benefits.

Behaviors When n Gets Large

Two functions grow "the same" if they behave the same when n is large.

$\frac{1}{100}n^3 + 10n + 1000000\log n$ behaves the same as n^3.

- As seen by applying limits:

$$\lim_{n\to\infty} \frac{\frac{1}{100}n^3 + 10n + 1000000\log n}{n^3} = \frac{1}{100}$$

- $\frac{1}{100}n^3$ dominates the other terms on top.
- Discarding constant factors gives us n^3.

$5 \cdot 2^n + n^{1000} + 2^{\log n}$ behaves the same as $5 \cdot 2^n$ (and 2^n).

- Dominating term is the fastest growing term.

Order to remember: exponential $>>$ polynomial $>>$ log

Practical View of Dominating Terms

Consider a 2 GHz processor (executes $2 * 10^9$ instructions per second).

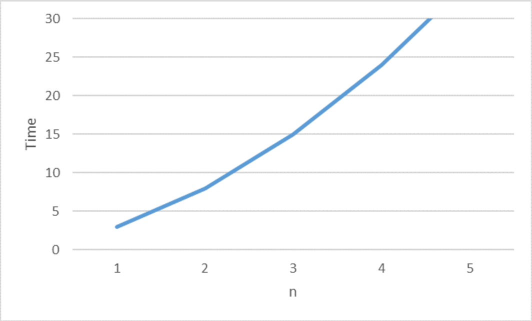
$f(n)$	Input Size (n) 10	20	50	100	1000
$\log \log n$	0.87 ns	1.06 ns	1.25 ns	1.37 ns	1.66 ns
$\log n$	1.66 ns	2.16 ns	2.82 ns	3.32 ns	4.98 ns
n	5 ns	10 ns	25 ns	50 ns	500 ns
$n \log n$	16.61 ns	43.22 ns	141.10 ns	332.19 ns	4.98 μs
n^2	50 ns	200 ns	1.25 μs	5 μs	500 μs
n^5	50 μs	1.6 ms	156.25 ms	5 s	138.9 h
2^n	512 ns	524.29 μs	6.5 d	$2 * 10^{13}$ y	$1.7 * 10^{284}$ y
$n!$	1.81 ms	38.57 y	$4.8 * 10^{47}$ y	$1.48 * 10^{141}$ y	:(

Time to execute $f(n)$ instructions using 2GHz processor

Asymptotic Analysis

Idea: Classify functions based on growth

Consider $T(n) = n^2 + 2n$.



What function f generalizes $T(n)$?

- $f(n) = n^2$ suffices:

$$f(n) \leq T(n) \leq 3f(n).$$

Big-O: Formal Definition

Big-O is an upper bound.

Big-O (big oh)

For any two functions $f, g : \mathbb{Z}^+ \to \mathbb{R}^+$,

$$f(n) = O(g(n)) \text{ iff } \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,\ f(n) \leq cg(n).$$

- $O(g(n))$: the set of all functions satisfying the condition above.
- $f(n) = O(g(n))$ meaning –
  - $f(n) \in O(g(n))$.
  - $f(n)$ is **bounded above** by a constant scaling of $g(n)$ for large n.
  - $f(n)$ **grows no faster than** $g(n)$.
- Examples:
  - $2n^2 + 4n = O(n^2)$.
  - $2n^2 + 4n = O(n^4 + 8n^3)$.
  - $n\log(n) + 5n = O(n^2 + 5n)$.

Big-Ω: Formal Definition

Big-Ω is a lower bound.

Big-Ω (big omega)

For any two functions $f, g : \mathbb{Z}^+ \to \mathbb{R}^+$,

$$f(n) = \Omega(g(n)) \text{ iff } \exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,\ f(n) \geq cg(n)$$

- $\Omega(g(n))$: the set of all functions satisfying the condition above.
- $f(n) = \Omega(g(n))$ meaning –
  - $f(n) \in \Omega(g(n))$.
  - $f(n)$ is **bounded below** by a constant scaling of $g(n)$ for large n.
  - $f(n)$ **grows no slower than** $g(n)$.
- Examples:
  - $2n^2 + 4n = \Omega(n^2 + 5)$.
  - $2n^2 + 4n = \Omega(\log(n))$.
  - $n\log(n) + 5n = \Omega(n\log(n))$.

Big-Θ: Formal Definition

Big-Θ is an upper and lower bound.

Big-Θ (big theta)

For any two functions $f, g : \mathbb{Z}^+ \to \mathbb{R}^+$,

$$f(n) = \Theta(g(n)) \text{ iff } [f(n) = O(g(n))] \wedge [f(n) = \Omega(g(n))].$$

- $\Theta(g(n))$: the set of all functions satisfying the condition above.
- $f(n) = \Theta(g(n))$ meaning –
  - $f(n) \in \Theta(g(n))$.
  - $f(n)$ is **bounded above and below** by constant scalings of $g(n)$ for large n.
  - Asymptotically speaking, $f(n)$ runs the same as $g(n)$.
- Examples:
  - $2n^2 + 4n = \Theta(n^2)$.
  - $2n^2 + 4n \neq \Theta(n)$.
  - $n\log(n) + 5n = \Theta(n\log(n))$.

Common Runtimes

- $\Theta(1)$ is **constant time.**
  - $T(n) = c$, some constant $c > 0$. $T(n) = \Theta(1)$.
- $\Theta(\log(n))$ is **logarithmic time.**
  - $T(n) = c\log(n)$, some constant $c > 0$. $T(n) = \Theta(\log(n))$.
- $\Theta(n)$ is **linear time.**
  - $T(n) = an + b$, some constants a, b, where $a > 0$. $T(n) = \Theta(n)$.
- $\Theta(n^2)$ is **quadratic time.**
  - $T(n) = an^2 + bn + c$, some constants a, b, c, where $a > 0$. $T(n) = \Theta(n^2)$.
- $\Theta(n^k)$ is **polynomial time** (for $k \in \mathbb{Z}^+$).
  - $T(n) = a_k n^k + \ldots + a_1 n + a_0$, for constants $a_0, \ldots, a_k$, where $a_k > 0$. $T(n) = \Theta(n^k)$.
- $\Theta(c^n)$ is **exponential time.**
  - $T(n) = c^n$, some constant $c > 1$. Then $T(n) = \Theta(c^n)$.

Why do constants matter?

Reducing lines of code can shave constant-factor off runtime.

- Consider the body of a for loop

```
for(i <- 0 until n) {...}
```

  - Original body: 10 statements (all with equal cost).
  - You reduce the statements by 3.
  - New runtime?
    - 7/10 of old runtime (30% faster).

Simplifying non-repeated code saves additive constant.

Why do constants matter?

Why do we care about the constant c or n_0?

- Consider $T_1(n) = 100n$ vs $T_2(n) = n^2$.
  - $c = 1$, $n_0 = 100$: $\forall n \geq 100, T_1(n) \leq T_2$.
  - Until inputs of size 100 or more, n^2 is small (i.e., faster) than $100n$.

Asymptotically slower runtime can be better.

- Algorithm with runtime $T_2(n)$ is best if input size < 100.
- Algorithm with runtime $T_2(n)$ might be easier to implement.
- No solution with runtime $T_1(n)$ might exist.

Runtime to sort a sequence?

BubbleSort Algorithm

```
bubbleSort(seq: a sequence of items)
1.  n ← seq length.
2.  for i ← n-2 to 0.
3.      for j ← 0 to i.
4.          if seq(j+1) < seq(j):
5.              swap seq(j) and seq(j+1).
```

- Runtime $T(n)$:
  - -- n is the length of the sequence.
  - -- at each step of j, pass through (4-5) $i+1$ times.
  - -- at each i, there are n values of j.

$$\sum_{i=0}^{n-2}(i+1) = O(n^2)?$$

- What are the requirements of the input `seq`?

Runtime to sort a sequence?

BubbleSort for mutable sequence.

```
def sort(seq: mutable.Seq[Int]): Unit = {
  val n = seq.length
  for (i <- n-2 to 0 by -1; j <- 0 to i) {
    if (seq(j+1) < seq(j)) {
      val temp = seq(j+1)
      seq(j+1) = seq(j)
      seq(j) = temp
    }
  }
}
```

- Is the runtime $T(n) = O(n^2)$?
  - -- What is the cost of `seq(j+1) < seq(j)`?
  - -- What is the cost of each `seq(k)`?

Runtime to sort a sequence?

BubbleSort for immutable sequence.

```scala
def sort(seq: Seq[Int]): Seq[Int] = {
  val newSeq = seq.toArray
  val n = seq.length
  for (i <- n-2 to 0 by -1; j <- 0 to i) {
    if (newSeq(j+1) < newSeq(j)) {
      val temp = newSeq(i);
      newSeq(i) = newSeq(j);
      newSeq(j) = temp
    }
  }
  newSeq.toList
}
```

- Is the runtime $T(n) = O(n^2)$?
  - What is the cost of `toArray`?
  - What is the cost of `toList`?

Searching Sequences

What is the cost of finding a value in a sequence?

```
def indexOf[T](seq: Seq[T], value: T, from: Int): Int = {
  for (i <- from until seq.length) {
    if (seq(i) == value) return i
  }
  -1
}
```

- Expected runtime is $T(n) = O(n)$.

What about counting all occurrences of a value in a sequence?

```
def count[T](seq: Seq[T], value: T): Int = {
  var count = 0; var i = indexOf(seq, value, 0)
  while (i != -1) {
    count += 1; i = indexOf(seq, value, i+1)
  }
  count
}
```

- Runtime is $T(n) = \Theta(n)$.

Searching Sequences

Problem

How can we search a sorted sequence?

- If random access is available: Binary Search
  To search the range [`begin`,`end`) for `value`:
  1. Compare `value` to `middle` element.
  2. If `value` is found, return index.
  3. If `value` is smaller than `middle`, search [`begin`,`middle`)
  4. If `value` is larger than `middle`, search (`middle`,`end`)
  5. Continue until range is empty or `value` found.
- If no random access?