CSE 250

Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu

Dr. Oliver Kennedy
okennedy@buffalo.edu

212 Capen Hall

Day 07

Runtime Analysis with Examples

Textbook Ch. 7.3-7.4

Announcements

- I'm back (obviously…)
- PA1 due on Friday at 11:59pm
  - Be wary of availability after 5:00pm…

Recap of Runtime Complexity

Big-Θ

- Growth functions are in the **same** complexity class
- If f(n) ∈ **Θ**(g(n)) then an algorithm taking f(n) steps is as "exactly" as fast as one that takes g(n) steps.

Big-O

- Growth functions in the **same or smaller** complexity class.
- If f(n) ∈ O(g(n)), then an algorithm that takes f(n) steps is *at least as fast* as one taking g(n) (but it may be even faster).

Big-Ω

- Growth functions in the **same or bigger** complexity class
- If f(n) ∈ **Ω**(g(n)), then an algorithm that takes f(n) steps is *at least as slow* as one that takes g(n) steps (but it may be even slower)

Recap of Runtime Complexity

Big-Θ — Tight Bound

- Growth functions are in the **same** complexity class
- If f(n) ∈ **Θ**(g(n)) then an algorithm taking f(n) steps is as "exactly" as fast as one that takes g(n) steps.

Big-O — Upper Bound

- Growth functions in the **same or smaller** complexity class.
- If f(n) ∈ O(g(n)), then an algorithm that takes f(n) steps is *at least as fast* as one taking g(n) (but it may be even faster).

Big-Ω — Lower Bound

- Growth functions in the **same or bigger** complexity class
- If f(n) ∈ **Ω**(g(n)), then an algorithm that takes f(n) steps is *at least as slow* as one that takes g(n) steps (but it may be even slower)

Common Runtimes (in order of complexity)

Constant Time: $\Theta(1)$

Logarithmic Time: $\Theta(\log(n))$

Linear Time: $\Theta(n)$

Quadratic Time: $\Theta(n^2)$

Polynomial Time: $\Theta(n^k)$ for some $k > 0$

Exponential Time: $\Theta(c^n)$ (for some $c \geq 1$)

Common Runtimes (in order of complexity)

Constant Time: $\Theta(1)$ $T(n) = c$

Logarithmic Time: $\Theta(\log(n))$ $T(n) = c \log(n)$

Linear Time: $\Theta(n)$ $T(n) = c_1 n + c_0$

Quadratic Time: $\Theta(n^2)$ $T(n) = c_2 n^2 + c_1 n^1 + c_0$

Polynomial Time: $\Theta(n^k)$ for some $k > 0$ $T(n) = c_k n^k + \ldots + c_1 n + c_0$

Exponential Time: $\Theta(c^n)$ (for some $c \geq 1$) $T(n) = c^n$

Constants vs Asymptotics

Given the following pseudocode:

```
for (i ← 0 until n) { /* do work */ }
```

If the `/* do work */` portion of the code originally takes 10 steps…

But we optimize it to now take 7 steps…

Our total runtime goes from 10n steps to 7n steps: 30% faster!

…but still **Θ(n)**

c and n_0

Compare the two runtimes:

$$T_1(n) = 100n$$

$$T_2(n) = n^2$$

- $100n = O(n^2)$ (T_2 is the slower runtime)
- …but $c_{hig} = 1, n_0 = 100$
- Until our input size reaches 100 or more, T_2 is the **faster** runtime

Takeaways

Asymptotically slower runtimes *can* be better in real-world situations.

- An algorithm with runtime T_2 is better on small inputs
- An algorithm with runtime T_2 might be easier to implement/maintain
- An algorithm with runtime T_1 might not exists

Takeaways

Asymptotically slower runtimes *can* be better in real-world situations.

- An algorithm with runtime T_2 is better on small inputs
- An algorithm with runtime T_2 might be easier to implement/maintain
- An algorithm with runtime T_1 might not exists

(sometimes this is provable…see CSE 331)

Takeaways

The important thing is learning the tools to *reason* about the different algorithms and *why* you might choose one over the other!

Takeaways

The important thing is learning the tools to *reason* about the different algorithms and *why* you might choose one over the other!

...But for this class, we can assume that if $T_2(n)$ is in a bigger complexity class, then $T_1(n)$ is better/faster/stronger.

Now some examples…

…and common pitfalls

Bubble Sort

```
bubblesort(seq: Seq[Int]):
  n ← seq length
  for i ← n-2 to 0, by -1:
    for j ← i to n-1:
      if seq(j+1) < seq(j):
        swap seq(j) and seq(j+1)
```

What is the runtime complexity class for bubblesort?

Helpful Summation Rules

1. $\sum_{i=j}^{k} c = (k - j + 1)c$

2. $\sum_{i=j}^{k}(cf(i)) = c\sum_{i=j}^{k} f(i)$

3. $\sum_{i=j}^{k}(f(i) + g(i)) = \left(\sum_{i=j}^{k} f(i)\right) + \left(\sum_{i=j}^{k} g(i)\right)$

4. $\sum_{i=j}^{k}(f(i)) = \left(\sum_{i=\ell}^{k}(f(i))\right) - \left(\sum_{i=\ell}^{j-1}(f(i))\right)$ (for any $\ell < j$)

5. $\sum_{i=j}^{k} f(i) = f(j) + f(j+1) + \ldots + f(k-1) + f(k)$

6. $\sum_{i=j}^{k} f(i) = f(j) + \ldots + f(\ell - 1) + \left(\sum_{i=\ell}^{k} f(i)\right)$ (for any $j < \ell \leq k$)

7. $\sum_{i=j}^{k} f(i) = \left(\sum_{i=j}^{\ell} f(i)\right) + f(\ell + 1) + \ldots + f(k)$ (for any $j \leq \ell < k$)

8. $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$

9. $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

10. $n! \leq c_s n^n$ is a tight upper bound (Sterling: Some constant c_s exists)

Bubble Sort

```
bubblesort(seq: Seq[Int]):
  n ← seq length
  for i ← n-2 to 0, by -1:
    for j ← i to n-1:
      if seq(j+1) < seq(j):
        swap seq(j) and seq(j+1)
```

Note: We can ignore the *exact* number of steps required by a portion of the algorithm, as long as we know its complexity…

Bubble Sort

```
   bubblesort(seq: Seq[Int]):
1.   n ← seq length
2.   for i ← n-2 to 0, by -1:
3.     for j ← i to n-1:
4.       if seq(j+1) < seq(j):
5.         swap seq(j) and seq(j+1)
```

Lines 4-5 are executed exactly n-1 times, but we can treat this as O(n) steps for the inner loop…or can we…?

Note: We can ignore the *exact* number of steps required by a portion of the algorithm, as long as we know its complexity…

Bubble Sort

```
   bubblesort(seq: Seq[Int]):
1.   n ← seq length
2.   for i ← n-2 to 0, by -1:
3.     for j ← i to n-1:
4.       if seq(j+1) < seq(j):
5.         swap seq(j) and seq(j+1)
```

Lines 4-5 are executed exactly n-1 times, but we can treat this as O(n) steps for the inner loop…or can we…?

Note: We can ignore the *exact* number of steps required by a portion of the algorithm, as long as we know its complexity…

Can we safely say this algorithm is $\Theta(n^2)$?

Bubble Sort

```
bubblesort(seq: Seq[Int]):
  n ← seq length
  for i ← n-2 to 0, by -1:
    for j ← i to n-1:
      if seq(j+1) < seq(j):
        swap seq(j) and seq(j+1)
```

What is the complexity of this step?

Note: We can ignore the *exact* number of steps required by a portion of the algorithm, as long as we know its complexity…

Can we safely say this algorithm is $\Theta(n^2)$?

Bubble Sort

```
bubblesort(seq: Seq[Int]):
  n ← seq length
  for i ← n-2 to 0, by -1:
    for j ← i to n-1:
      if seq(j+1) < seq(j):
        swap seq(j) and seq(j+1)
```

What is the complexity of this step?
Do not assume function calls take O(1) time!

Note: We can ignore the *exact* number of steps required by a portion of the algorithm, as long as we know its complexity…

Can we safely say this algorithm is $\Theta(n^2)$?

Bubble Sort on Mutable Data

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

Bubble Sort on Immutable Data

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

Searching Sequences

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

What is the complexity?

Searching Sequences

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

What is the complexity? O(n)

Searching Sequences

```
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)
    }
    return count
}
```

What is the complexity?

Searching Sequences

```
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)
    }
    return count
}
```

What is the complexity? O(n)?

Searching Sequences

```
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)
    }
    return count
}
```

What is the complexity? O(n)? What about this line?

Searching Sequences

```
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)
    }
    return count
}
```

What is the complexity? O(n)? What about this line? How many while iterations?

Searching Sequences

```
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)
    }
    return count
}
```

What is the complexity? Each element is only checked once, so O(n).

Searching Sorted Sequences

- Assuming O(1) access to elements ('random access')
  - Divide the set of elements in half by taking the "middle" element, *m*
    - If *m* is greater than what we are looking for, search the lower half
    - If *m* is less than what we are looking for, search the right half
    - Repeat until you've found the element or you can't divide in half

Searching Sorted Sequences

- Assuming O(1) access to elements ('random access')
  - Divide the set of elements in half by taking the "middle" element, *m*
    - If *m* is greater than what we are looking for, search the lower half
    - If *m* is less than what we are looking for, search the right half
    - Repeat until you've found the element or you can't divide in half

If you have n elements, how many times can you divide n in half?

Searching Sorted Sequences

- Assuming O(1) access to elements ('random access')
  - Divide the set of elements in half by taking the "middle" element, *m*
    - If *m* is greater than what we are looking for, search the lower half
    - If *m* is less than what we are looking for, search the right half
    - Repeat until you've found the element or you can't divide in half

If you have n elements, how many times can you divide n in half?

log(n)