Analysis of Algorithms & Big-O

CS16: Introduction to Algorithms & Data Structures
Spring 2019
Outline

- Running time
- Big-$\mathcal{O}$
- Big-$\Omega$ and Big-$\Theta$
What is an “Efficient” Algorithm

- Possible efficiency measures
  - Total amount of time on a stopwatch?
  - Low memory usage?
  - Low power consumption?
  - Network usage?
- In CS16 we will focus on running time
Q: How should we measure running time?
A Simple Algorithm

- How do we measure its running time?

```javascript
function sum_array(array)
    // Input: an array of 100 integers
    // Output: the sum of the integers
    if array.length = 0
        return error
    sum = 0
    for i in [0, array.length-1]:
        sum = sum + array[i]
    return sum
```
Measuring Running Time

- Experimentally?
  - Implement algorithm
  - Run algorithm on inputs of different size
  - Measure time it takes to finish
  - Plot the results
Q: Was that useful?
Experimental Running Time

- How large should the array be in the experiment?
- Which array should we use (i.e., which ints)?
- Which hardware should we run on?
- Which operating system?
- Which compiler should we use?
- Which compiler flags?
- ...

...
Measuring Running Time

- We need a measure that is
  - independent of hardware
  - independent of OS
  - independent of compiler
  - ...

- It should depend only on
  - “intrinsic properties of the algorithm”
Q: What is the *intrinsic* running time of an algorithm?
A Simple Algorithm

```plaintext
function sum_array(array)
    // Input: an array of integers
    // Output: the sum of the integers
    if array.length = 0
        return error
    sum = 0
    for i in [0, array.length-1]:
        sum = sum + array[i]
    return sum
```
Knuth’s Observation

› Experimental running time can be determined using
  › Time of each operation & frequency of each operation

› Example:
  › run sum_array on array of size 100

\[
\text{time}(\text{sum_array}) = \text{time}(\text{read}) \cdot 100 + \text{time}(\text{add}) \cdot 99 + \text{time}(\text{comp}) \cdot 1
\]
\[
= 3\text{ms} \cdot 100 + 100\text{ms} \cdot 99 + 10\text{ms} \cdot 1
\]
\[
= 10.21\text{s}
\]

› **Key insight!**
  › the time an operation takes depends on hardware but…
  › the number of times an operation is repeated does not depend on hardware
  › So let’s ignore time and only focus on number of times an operation is repeated
Knuth’s Observation

- How do we ignore time?
  - we’ll assume each operation takes 1 unit of time

- Example:
  - sum_array on array of size 100

\[
\text{time}(\text{sum\_array}) = \text{time(\text{read})} \cdot 100 + \text{time(\text{add})} \cdot 99 + \text{time(\text{comp})} \cdot 1 \\
= 1 \cdot 100 + 1 \cdot 99 + 1 \cdot 1 \\
= 100 \text{ reads} + 99 \text{ adds} + 1 \text{ comp}
\]

- Let's simplify and just report total number of operations
  - \( \text{time(\text{sum\_array})} = 200 \text{ ops} \)
Elementary Operations

- Most algorithms make use of standard “elementary” operations:
  - Math: +, -, *, /, max, min, log, sin, cos, abs, ...
  - Comparisons: ==, >, <, ≤, ≥
  - Variable assignment
  - Variable increment or decrement
  - Array allocation
  - Creating a new object
  - Function calls and value returns
  - Careful: an object's constructor & function calls may have elementary ops too!

- In practice all these operations take different amounts of time but

  - **we will assume each operation takes 1 unit of time**
What is Running Time?

“Running time”

= 

Number of elementary operations

Running time ≠ Experimental time
Towards **Algorithmic** Running Time

- Problem #1
  - experimental running time depends on hardware
  - solution: *focus on number of operations*
A Simple Algorithm

```plaintext
function sum_array(array)
    // Input: an array of integers
    // Output: the sum of the integers
    if array.length = 0
        return error
    sum = 0
    for i in [0, array.length-1]:
        sum = sum + array[i]
    return sum
```

- Do we count "return error"?
  - depends on whether input array is empty
    - if `array` is empty then `sum_array` takes 2 ops
    - if `array` is not empty then `sum_array` takes $3+4\cdot n$ ops
Towards Algorithmic Running Time

- Problem #1
  - experimental running time depends on hardware
  - solution: focus on number of operations
- Problem #2
  - number of operations depends on input
  - solution: focus on number of operations for worst-case input
A Simple Algorithm

```plaintext
function sum_array(array)
    // Input: an array of integers
    // Output: the sum of the integers
    if array.length = 0
        return error
    sum = 0
    for i in [0, array.length-1]:
        sum = sum + array[i]
    return sum
```

- What is the worst-case input for our algorithm?
  - any array that is non-empty
  - so we'll just ignore "return error"
What is Running Time?

Worst-case running time

= Number of elementary operations on worst-case input
A Simple Algorithm

```javascript
function sum_array(array)
    // Input: an array of integers
    // Output: the sum of the integers
    if array.length = 0
        return error
    sum = 0
    for i in [0, array.length-1]:
        sum = sum + array[i]
    return sum
```

- How many times does loop execute?
  - depends on size of input array
Towards an **Algorithmic** Running Time

- **Problem #1**
  - experimental running time depends on hardware
  - solution: *focus on number of operations* (Knuth’s observation)

- **Problem #2**
  - number of operations depends on input
  - solution: *focus on number of operations on worst-case* input! Why?

- **Problem #3**
  - number of operations depends on input size
  - solution: *focus on number of operations as a function of input size* $n$. 
A Simple Algorithm

function sum_array(array)
    // Input: an array of integers
    // Output: the sum of the integers
    if array.length = 0
        return error
    sum = 0
    for i in [0, array.length-1]:
        sum = sum + array[i]
    return sum

- How many times does loop execute?
  - depends on size of input array
  - sum_array takes $3 + 4 \cdot n$ ops
What is Running Time?

Worst-case running time

$= T(n): \text{Number of elementary operations on worst-case input as a function of input size } n$
Constant Running Time

function first(array):
// Input: an array
// Output: the first element
return array[0]  

- How many operations are executed?
  - \( T(n) = 2 \) ops
  - What if array has 100 elements?
  - What if array has 100,000 elements?

- **key observation:**
  - running time does not depend on array size!
function argmax(array)
    // Input: an array
    // Output: the index of the maximum value
    index = 0
    for i in [1, array.length):
        if array[i] > array[index]:
            index = i
    return index
function argmax(array)
    // Input: an array
    // Output: the index of the maximum value
    index = 0
    for i in [1, array.length):
        if array[i] > array[index]:
            index = i
    return index

1op
1op per loop
3ops per loop
1op per loop (sometimes)
1op
function argmax(array)
    // Input: an array
    // Output: the index of the maximum value
    index = 0
    for i in [1, array.length):
        if array[i] > array[index]:
            index = i
    return index

Activity #1
### Linear Running Time

How many operations are executed?

- $T(n) = 5n + 2$ ops where $n = \text{size}(\text{array})$

**Key observation:**

- running time depends (mostly) on array size
function possible_products(array):
    // Input: an array
    // Output: a list of all possible products
    //         between any two elements in the list
    products = []
    for i in [0, array.length):
        for j in [0, array.length):
            products.append(array[i] * array[j])
    return products

1 op
1 op per loop
1 op per loop
1 op per loop
4 ops per loop
1 op

Activity #2
function possible_products(array):
  // Input: an array
  // Output: a list of all possible products
  //         between any two elements in the list
  products = []
  for i in [0, array.length):
    for j in [0, array.length):
      products.append(array[i] * array[j])
  return products
function possible_products(array):
    // Input: an array
    // Output: a list of all possible products
    //         between any two elements in the list
    products = []
    for i in [0, array.length):
        for j in [0, array.length):
            products.append(array[i] * array[j])
    return products
Quadratic Running Time

**function possible_products(array):**

// Input: an array
// Output: a list of all possible products
// between any two elements in the list

products = []
for i in [0, array.length):
    for j in [0, array.length):
        products.append(array[i] * array[j])
return products

- How many operations are executed?
  - \( T(n) = 5n^2 + n + 2 \) operations where \( n = \text{size(array)} \)

**key observation:**
- running time depends (mostly) on the square of array size
Running Times

**Constant**
- independent of input size

**Linear**
- depends on input size

**Quadratic**
- depends on square of input size
Q: how do we compare running times?
Which Algorithm is Better?

- Algorithm A takes $T_A(n) = 30n + 10$ ops
- Algorithm B takes $T_B(n) = 5n$ ops
Which Algorithm is Better?

- Alg A takes $T_A(n) = 5n + 1000$ ops
- Alg B takes $T_B(n) = 10n + 2$ ops
- It depends on $n$

$rtime(A) < rtime(B) \iff 5n + 1000 < 10n + 2 \iff 5n > 998 \iff n > 199.6$
Which Algorithm is Better?

- Alg A takes $T_A(n) = 1000n^2$ ops
- Alg B takes $T_B(n) = n^8$ ops
- It depends on $n$

$rtime(A) < rtime(B) \iff 1000n^2 < n^8$

\[ \iff 1000n^2 - n^8 < 0 \]
\[ \iff n^2(1000 - n^6) < 0 \]
\[ \iff 1000 - n^6 < 0 \]
\[ \iff n > 1000^{1/6} \]
\[ \iff n > 3.16... \]
What is Running Time?

Asymptotic worst-case running time

\[ \text{Number of elementary operations on worst-case input as a function of input size } n \]

when \( n \) tends to infinity

In CS “running time” usually means asymptotic worst-case running time…but not always!
we will learn about other kinds of running times
Comparing Running Times

Comparing asymptotic running times

\[ T_A(n) \text{ is better than } T_B(n) \text{ if}\]

for large enough \( n \)

\[ T_A(n) \text{ grows slower than } T_B(n) \]
Q: can we formalize all this mathematically?
**Big-O**

**Definition (Big-O):** $T_A(n)$ is $O(T_B(n))$ if there exists positive constants $c$ and $n_0$ such that:

$$T_A(n) \leq c \cdot T_B(n)$$

for all $n \geq n_0$

- $T_A(n)$'s order of growth is at most $T_B(n)$'s order of growth
- **Examples**
  - $2n+10$ is $O(n)$
  - $n^{10}+2019$ is $O(n^{10})$ and also $O(n^{50})$
Big-O

‣ How do we find “the Big-O of something”?
  ▸ Usually you “eyeball” it
  ▸ Then you try to prove it
    ▸ (though most of the time in CS16 it will be simple enough that you don't need to prove it)
Big-O Examples

Definition (Big-O): \( T_A(n) \) is \( O(T_B(n)) \) if there exists positive constants \( c \) and \( n_0 \) such that:

\[
T_A(n) \leq c \cdot T_B(n)
\]

for all \( n \geq n_0 \)

- \( 2n+10 \) is \( O(n) \)
  - for example, choose \( c=3 \) and \( n_0=10 \)
  - Why? because
  - \( 2n+10 \leq 3 \cdot n \) when \( n \geq 10 \)
  - for example, \( 2 \cdot 10+10 \leq 3 \cdot 10 \)
We don’t care what happens here

We only care what happens here
Experimental measurement

Big-O
More Big-O Examples

› $n^2$ is not $O(n)$ Why?

› To prove that $n^2$ is $O(n)$ we have to find a positive constant $c$ and a positive constant $n_0$ such that

› $n^2 \leq c \cdot n$ for all $n > n_0$

› This is not possible!

› equivalent to asking that

› $n \leq c$ for all $n > n_0$
Big-O & Growth Rate
Big-O & Growth Rate

Activity #3

1 min
Big-O & Growth Rate
Eyeballing Big-O

- If $T(n)$ is a polynomial of degree $d$ then $T(n)$ is $O(n^d)$
- In other words you can ignore
  - lower-order terms
  - constant factors
- Examples
  - $1000n^2 + 400n + 739$ is $O(n^2)$
  - $n^{80} + 43n^{72} + 5n + 1$ is $O(n^{80})$
- For the Big-O, use the smallest upper bound
  - $2n$ is $O(n^{50})$ but that’s not really a useful bound
  - instead it is better to say that $2n$ is $O(n)$
Example Big-O Analysis

- Given algorithm, find number of ops as a function of input size
  - first: $T(n)=2$
  - argmax: $T(n)=5n+2$
  - possible_products: $T(n)=5n^2+n+3$
- Replace constants with “$c$” (they are irrelevant as $n$ grows)
  - first: $T(n)=c$
  - argmax: $T(n)=c_0n+c_1$
  - possible_products: $T(n)=c_0n^2+n+c_1$
Example Big-O Analysis

- Discard constants & use smallest possible degree
  - first: \( T(n) = c \) is \( O(1) \)
  - argmax: \( T(n) = c_0 n + c_1 \) is \( O(n) \)
  - possible products: \( T(n) = c_0 n^2 + n + c_1 \) is \( O(n^2) \)
- The convention for \( T(n) = c \) is to write \( O(1) \)
Big-O

Definition (Big-O): \( T_A(n) \) is \( \mathcal{O}(T_B(n)) \) if there exists positive constants \( c \) and \( n_0 \) such that:
\[
T_A(n) \leq c \cdot T_B(n)
\]
for all \( n \geq n_0 \)

- \( T_A(n) \)'s growth rate is upper bounded by \( T_B(n) \)'s growth rate
- But what if we need to express a lower bound?
  - we use Big-\( \Omega \) notation!
Big-Omega

**Definition (Big-Ω):** $T_A(n)$ is $\Omega(T_B(n))$ if there exists positive constants $c$ and $n_0$ such that:

$$T_A(n) \geq c \cdot T_B(n)$$

for all $n \geq n_0$

- $T_A(n)$’s growth rate is lower bounded by $T_B(n)$’s growth rate
- What about an upper and a lower bound?
  - We use Big-$\mathbf{P}$ notation
Big-Theta

Definition (Big-$\Omega$): $T_A(n)$ is $\Omega(T_B(n))$ if it is $O(T_B(n))$ and $\Omega(T_B(n))$.

- $T_A(n)$’s growth rate is the same as $T_B(n)$’s
More Examples

Activity #4

2 min
More Examples

1 min

Activity #4
More Examples

Activity #4

0 min
## More Examples

<table>
<thead>
<tr>
<th>$T(n)$</th>
<th>Big-$O$</th>
<th>Big-$\Omega$</th>
<th>Big-$P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$an + b$</td>
<td>?</td>
<td>?</td>
<td>$P(n)$</td>
</tr>
<tr>
<td>$an^2 + bn + c$</td>
<td>?</td>
<td>?</td>
<td>$P(n^2)$</td>
</tr>
<tr>
<td>$a$</td>
<td>?</td>
<td>?</td>
<td>$P(1)$</td>
</tr>
<tr>
<td>$3^n + an^{40}$</td>
<td>?</td>
<td>?</td>
<td>$P(3^n)$</td>
</tr>
<tr>
<td>$an + b \log n$</td>
<td>?</td>
<td>?</td>
<td>$P(n)$</td>
</tr>
</tbody>
</table>
Running Times

- $O(1)$: independent of input size
- $O(n)$: depends on input size
- $O(n^2)$: depends on square of input size
- $O(n^3)$: depends on cube of input size
- $O(n^{70})$: depends on 70th power of input size
- $O(2^n)$: exponential in input size
<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log n$</th>
<th>$n$</th>
<th>$n \log n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>4,096</td>
<td>65,536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>32</td>
<td>160</td>
<td>1,024</td>
<td>32,768</td>
<td>4,294,967,296</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>64</td>
<td>384</td>
<td>4,096</td>
<td>262,144</td>
<td>$1.84 \times 10^{19}$</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>128</td>
<td>896</td>
<td>16,384</td>
<td>2,097,152</td>
<td>$3.40 \times 10^{38}$</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>256</td>
<td>2,048</td>
<td>65,536</td>
<td>16,777,216</td>
<td>$1.15 \times 10^{77}$</td>
</tr>
<tr>
<td>512</td>
<td>9</td>
<td>512</td>
<td>4,608</td>
<td>262,144</td>
<td>134,217,728</td>
<td>$1.34 \times 10^{154}$</td>
</tr>
</tbody>
</table>
Readings

- Asymptotic runtime and Big-O
  - Dasgupta et al. section 0.3 (pp. 15-17)
  - Roughgarden Part 1, Chap 2
Announcements

- Homework 1 due this Friday at 5pm!
- Thursday is in-class Python lab!
- If you are able to work on your own laptop
  - Go to McMillan 117 (here!)
- Make sure you can log into your CS account before attending lab
- See SunLab consultant if you have any account issues!
- Sections started yesterday
  - if you are not signed up, you could be in trouble!
References

- Slide #10
  - the portrait on the left is a drawing; really!
- Slide #25
  - Usain Bolt (constant): 8-time Olympic gold medalist and greatest sprinter of all time
  - Sally Pearson (linear): 2012 Olympic world champion in 100m hurdles, 2011 and 2017 World Champion
  - Wilson Kipsang (quadratic): former marathon world-record holder, Olympic marathon bronze medalist
  - Eliud Kipchoge (quadratic): 2016 Olympic marathon gold medalist, greatest marathoner of the modern era