Analysis of Algorithms & Big-O

CS16: Introduction to Algorithms & Data Structures
Spring 2018
Outline

- Running time
- Big-$\mathcal{O}$
- Big-$\Omega$ and Big-$\Theta$
- Analyzing Seamcarve
- Dynamic programming
- Fibonacci sequence
Algorithms

› What can we analyze about an algorithm?
  › Make a list
Algorithms

- What can we analyze about an algorithm?
  - Make a list
Algorithms

› What can we analyze about an algorithm?
  › Make a list
What is an “Efficient” Algorithm

- Possible efficiency measures
  - Total amount of time on a stopwatch?
  - Low memory usage?
  - Low power consumption?
  - Network usage?

- The analysis of algorithms helps us quantify this
Q: How can we measure running time?
Measuring Running Time

- Experimentally?
  - Implement algorithm
  - Run algorithm on inputs of different size
  - Measure the running time
  - Plot the results

Great! We’re done, right?
Measuring Running Time

- What if you can’t implement algorithm?
- Which inputs exactly should you choose?
- Which hardware should you run on?
- Which operating system?
- Which compiler?
- Which compiler flags?
- …
Measuring Running Time

- We need a measure that
  - 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?
Knuth’s Observation

- Running time can be determined using
  - Time/cost of each operation
  - Frequency of each operation

Example:

- function that sums 100 integers

\[
\text{time}(\text{sum}) = \text{time}(\text{read}) \cdot 100 + \text{time}(\text{add}) \cdot 99
\]

Key insight!

- cost of operations depend on hardware, OS, compiler,…
- frequency of operations depend on algorithm
Q: What operations exactly?
Elementary Operations

- Algorithmic running “time” is measured in 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
  - in algorithm analysis we assume each operation takes 1 unit of time
Towards an **Algorithmic** Running Time

- **Problem #1**
  - running time varies with hardware, OS etc…
  - solution #1: focus on number of operations

- **Problem #2**
  - number of operations varies with input size
  - solution #2: focus on number of operations for *large* inputs

- **Problem #3**
  - number of operations varies with input
  - solution #3: focus on number of operations on *worst-case* inputs
Towards an **Algorithmic** Running Time

- Why worst-case inputs?
  - Easier to analyze
  - Gives useful information
    - what if a plane autopilot program runs slower than predicted due to an unexpected input?
- Why large inputs?
  - Easier to analyze
  - We usually care what happens on large data
  - Allows us to ignore odd behaviors that happen on small data
(Worst-case) Analysis of Algorithms

- Based only on high-level algorithm descriptions
  - not on implementation
- Takes into all possible inputs
  - by considering the worst-case inputs
- Quantifies running time independently of
  - hardware, OS, compiler etc.
  - Algorithm’s running time vs program’s running time
**Constant Running Time**

- How many operations are executed?
  - 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

1 op
1 op per loop
3 ops per loop
1 op per loop (sometimes)
1 op
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

Activity #1
Linear Running Time

- How many operations are executed?
  - What if array has 10 elements?
  - What if array has 100,000 elements?

**key observation:** running time depends on array size
- $5n+2$ operations where $n=\text{size}(\text{array})$
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

Activity #2

1 min
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

1op
1op per loop
1op per loop per loop
4ops per loop per loop
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?
  - What if array has 10 elements?
  - What if array has 100,000 elements?

**key observation:** running time depends on the *square* of array size
  - $5n^2+n+2$ operations where $n=size(array)$
Running Times

**Constant**
independent of input size

**Linear**
depends on input size

**Quadratic**
depends on square of input size
Plotting Running Times

\( T(n) \)

\[ 5n^2 + n + 2 \]
Plotting Running Times

\[ T(n) \]

\[ 5n^2 + n + 2 \]

\[ 5n + 2 \]
We don’t care what happens here  We only care what happens here
Big-O Notation

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

$$f(n) \leq c \cdot g(n)$$

for all $n \geq n_0$

- Example: $2n+10$ 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, $2 \cdot 10+10 \leq 3 \cdot 10$
Big-O Notation

- Another example
  - \( n^2 \) is not \( O(n) \)
  - Why? To prove that \( n^2 \) is \( O(n) \) we have to show that there exists constants \( c \) and \( n_0 \) such that
    - \( n^2 \leq c \cdot n \iff n \leq c \text{ for all } n \geq n_0 \)
  - This is not possible!
    - for example set \( c = 10 \)
Big-O & Growth Rate

Activity #3
Big-O & Growth Rate

Activity #3
Big-O & Growth Rate
Big-O & Growth Rate

- Big-O gives upper bound on
  - growth rate of function when input is large

- An algorithm is $\mathcal{O}(g(n))$ if growth its rate is
  - no more than growth rate of $g(n)$

- Examples

  - $n^2$ is not $\mathcal{O}(n)$
  - $n$ is $\mathcal{O}(n^2)$
  - $n^2$ is $\mathcal{O}(n^3)$
Summary of Big-O Rules

- If $f(n)$ is a polynomial of degree $d$ then
  - $f(n)$ is $O(n^d)$

- In other words you can ignore
  - lower-order terms
  - constant factors

- Use the term with the smallest possible degree
  - $2n$ is $O(n^{50})$ but that’s not helpful
  - instead it is better to say it is $O(n)$

- **Discard constant factors & use smallest possible degree**
Example Big-O Notation

- Count number of operations as a function of input size
- For example
  - **first**: $T(n) = 2$
  - **argmax**: $T(n) = 5n + 2$
  - **possible_products**: $T(n) = 5n^2 + n + 3$
- Can replace constants with $c$ b/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 Notation

- **Discard constant factors & use smallest possible degree**

- For example
  - **first**: $T(n) = c$ is $O(1)$
  - **argmax**: $T(n) = c_0n + c_1$ is $O(n)$
  - **possible_products**: $T(n) = c_0n^2 + n + c_1$ is $O(n^2)$

- The convention for $T(n) = c$ is to write $O(1)$
5n^2 + n + 2

O(n^2)

?
Big-Omega

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

$$f(n) \leq c \cdot g(n)$$

for all $n \geq n_0$

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

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

\[
f(n) \geq c \cdot g(n)
\]

for all \( n \geq n_0 \)

- \( f(n) \)'s growth rate is lower bounded by \( g(n) \)'s growth rate
- What about an upper **and** a lower bound?
- We use Big-\( \Omega \) notation
Big-Theta

Definition (Big-\(P\)): \( f(n) \) is \( P(g(n)) \) if it is \( \Theta(g(n)) \) and \( \Omega(g(n)) \).

- \( f(n) \)'s growth rate is the same as \( g(n) \)'s
More Examples

Activity #4

2 min
More Examples
## More Examples

<table>
<thead>
<tr>
<th>$f(n)$</th>
<th>Big-$\mathcal{O}$</th>
<th>Big-$\Omega$</th>
<th>Big-$\mathcal{P}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$an + b$</td>
<td>?</td>
<td>?</td>
<td>$\mathcal{P}(n)$</td>
</tr>
<tr>
<td>$an^2 + bn + c$</td>
<td>?</td>
<td>?</td>
<td>$\mathcal{P}(n^2)$</td>
</tr>
<tr>
<td>$a$</td>
<td>?</td>
<td>?</td>
<td>$\mathcal{P}(1)$</td>
</tr>
<tr>
<td>$3^n + an^{40}$</td>
<td>?</td>
<td>?</td>
<td>$\mathcal{P}(3^n)$</td>
</tr>
<tr>
<td>$an + b \log n$</td>
<td>?</td>
<td>?</td>
<td>$\mathcal{P}(n)$</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
Finding Low Importance Seams

- How many seams in a $c \times r$ image?
  - At each row the seam can go Left, Right or Down
  - It chooses 1 out of 3 dirs at each row and there are $r$ rows
  - So $3^r$ possible seams from some starting pixel
  - Since there are $c$ starting pixels total # of seams is $c \times 3^r$
- For square image with $n$ total pixels
  - there are $\sqrt{n} \times 3^{\sqrt{n}}$ possible seams
Finding Low Importance Seams

- Brute force algorithm:
  - **Try every possible** seam & find least important one

- What is running time of brute force algorithm?
  - Suppose image is of length $n = c \times r$
  - $O(\sqrt{n} \times 3^{\sqrt{n}})$
Seamcarve

- What is the runtime of Seamcarve (from last class)?
- The algorithm
  - Iterate over all pixels from bottom to top
  - Populate `costs` and `dirs` arrays
  - Create seam by choosing minimum value in top row and tracing downward
- How many operations per pixel?
  - A constant number of operations per pixel (4)
- Constant number of operations per pixel means algorithm is linear
  - $O(n)$ where $n$ is number of pixels
- Also could have counted # of nested loops in pseudocode…
Seamcarve

- How can we possibly go from
  - Exponential running time with brute force
  - Linear running time with Seamcarve?
  - What’s the secret to this magic trick?

Dynamic Programming!
Dynamic Programming

- Idea
  - re-use computation you’ve already done or in other words
  - avoid re-computing what you’ve already computed

- Seamcarve observation
  - many seams cross paths
  - so we don’t need to re-compute entire sums of importance if we’ve already computed terms before
  - that’s the purpose of the costs matrix!

- Dynamic programming
  - “store information you’ve computed to avoid re-computing it”
Fibonacci (Recursive)

- Defined by the recursive relation
  - \( F_0 = 0, \ F_1 = 1 \)
  - \( F_n = F_{n-1} + F_{n-2} \)

- We can implement this with a recursive function

```python
function fib(n):
    if n = 0:
        return 0
    if n = 1:
        return 1
    return fib(n-1) + fib(n-2)
```
Visualization of Fibonacci (Recursive)

- Each node of tree is a recursive call of Fib( )
- Each level of the tree is a level of the recursion

```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```
Fibonacci (Recursive)

Big-O runtime of recursive \texttt{fib} function?
Fibonacci (Recursive)

Big-O runtime of recursive \texttt{fib} function?
Fibonacci (Recursive)

Big-O runtime of recursive $\text{fib}$ function?

Activity #5
Fibonacci (Recursive)

- How many times does \texttt{fib} get called for \texttt{fib(4)}?
  - 8 times
- At each level it makes twice as many recursive calls as last
  - For \texttt{fib(n)} it makes approximately $2^n$ recursive calls
  - Algorithm is $O(2^n)$

```python
function fib(n):
    if n = 0:
        return 0
    if n = 1:
        return 1
    return fib(n-1) + fib(n-2)
```
Fibonacci: Dynamic Programming

- How many times does \texttt{fib(1)} get computed?
- Instead of recomputing Fibonacci numbers over and over again
- Compute them \textbf{once} and store them for later

```python
function dynamicFib(n):
    fibs = [] // make an array of size n
    fibs[0] = 0
    fibs[1] = 1

    for i from 2 to n:
        fibs[i] = fibs[i-1] + fibs[i-2]

    return fibs[n]
```
Fibonacci: Dynamic Programming

- What’s the runtime of `dynamicFib()`?
  - Calculates Fibonacci numbers from 0 to n
  - Performs $O(1)$ ops for each one
  - Runtime is clearly $O(n)$

- We again reduced runtime of algorithm
  - From exponential to linear
  - with dynamic programming!
Readings

- Dasgupta et al. section 0.2 (pp. 12-15)
  - Fibonacci example (without mentioning dynamic programming)
- Dasgupta et al. section 0.3 (pp. 15-17)
  - Describes Big-O notation
- Dasgupta et al. chapter 6 (pp. 169-199)
  - Goes into Dynamic Programming
  - This chapter builds significantly on earlier ones so might be challenging now (but we’ll see much of it this semester)
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 Salomon DECl (here!)
  - Otherwise, go to the Sunlab.
- 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