ANALYSIS OF ALGORITHMS AND BIG-O

CS16: Introduction to Algorithms & Data Structures
Outline

1) Running time and theoretical analysis
2) Big-O notation
3) Big-Ω and Big-Θ
4) Analyzing Seamcarve runtime
5) Dynamic programming
6) Fibonacci sequence
What is an “Efficient” Algorithm?

- Efficiency measures
  - Small amount of time? on a stopwatch?
  - Low memory usage?
  - Low power consumption?
  - Network usage?
- Analysis of algorithms helps quantify this
Measuring Running Time

- Experimentally
  - Implement algorithm
  - Run program with inputs of varying size
  - Measure running times
  - Plot results

- Why not?
  - What if you can’t implement algorithm?
  - Which inputs do you choose?
  - Depends on hardware, OS, …
Measuring Running Time

• Grows with the input size
  • Focus on large inputs

• Varies with input
  • Consider the worst-case input
Measuring Running Time

• Why worst-case inputs?
  • Easier to analyze
  • Practical: what if autopilot was slower than predicted for some untested input?

• Why large inputs?
  • Easier to analyze
  • Practical: usually care what happens on large data
Theoretical Analysis

• Based on high-level description of algorithm
• Not on implementation
• Takes into account all possible inputs
  • Worst-case or average-case
• Quantifies running time independently of hardware or software environment
Theoretical Analysis

- Associate cost to elementary operations
- Find number of operations as function of input size
Elementary Operations

- Algorithmic “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, object's constructor and function calls may have elementary ops too!)
- In practice, all these operations take different amounts of time
- In algorithm analysis assume each operation takes 1 unit of time
Example: Constant Running Time

function first(array):
    // Input: an array
    // Output: the first element
    return array[0] // index 0 and return, 2 ops

• How many operations are performed in this function if the list has ten elements? If it has 100,000 elements?
  • Always 2 operations performed
  • Does not depend on the input size
Example: Linear Running Time

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

• How many operations if the list has ten elements? 100,000 elements?
  • Varies proportional to the size of the input list: 5n + 2
  • We’ll be in the for loop longer and longer as the input list grows
  • If we were to plot, the runtime would increase linearly
Example: Quadratic Running Time

```cpp
function possible_products(array):
    // Input: an array
    // Output: a list of all possible products
    //    between any two elements in the list
    products = []  // make an empty list, 1 op
    for i in [0, array.length):  // 1 op per loop
        for j in [0, array.length):  // 1 op per loop per loop
            products.append(array[i] * array[j])  // 4 ops per loop per loop
    return products  // 1 op
```

- About $5n^2 + n + 2$ operations (okay to approximate!)
  - A plot of number of operations would grow quadratically!
- Each element must be multiplied with every other element
- Linear algorithm on previous slide had 1 for loop
- This one has 2 nested for loops
  - What if there were 3 nested loops?
Summarizing Function Growth

• For large inputs growth rate is less affected by:
  • constant factors
  • lower-order terms

• Examples
  • $10^5n^2 + 10^8n$ and $n^2$ grow with same slope despite differing constants and lower-order terms
  • $10n + 10^5$ and $n$ both grow with same slope as well

In this graph (log scale on both axes), the slope of a line corresponds to the growth rate of its respective function.
Big-O Notation

- Given any two functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there exist positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

- Example: $2n + 10$ is $O(n)$
  - Pick $c = 3$ and $n_0 = 10$
    - $2n + 10 \leq 3n$
    - $2(10) + 10 \leq 3(10)$
    - $30 \leq 30$
Big-O Notation (continued)

• Example: $n^2$ is not $O(n)$
  • $n^2 \leq cn$
  • $n \leq c$
  • The above inequality cannot be satisfied because $c$ must be a constant, therefore for any $n > c$ the inequality is false
Big-O and Growth Rate

• Big-O gives upper bound on growth of function
• An algorithm is $O(g(n))$ if growth rate is no more than growth rate of $g(n)$
• $n^2$ is not $O(n)$
  • But $n$ is $O(n^2)$
  • And $n^2$ is $O(n^3)$
• Why? Because Big-O is an upper bound!
Summary of Big-O Rules

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

- In other words
  - forget about lower-order terms
  - forget about constant factors

- Use the smallest possible degree
  - True that $2n$ is $O(n^{50})$, but that’s not helpful
  - Instead, say it’s $O(n)$
    - discard constant factor & use smallest possible degree
Constants in Algorithm Analysis

• Find number of elementary operations executed as a function of input size
  • first: \( T(n) = 2 \)
  • argmax: \( T(n) = 5n + 2 \)
  • possible_products: \( T(n) = 5n^2 + n + 3 \)
• In the future skip counting operations
• Replace constants with \( c \) since irrelevant as \( n \) grows
  • first: \( T(n) = c \)
  • argmax: \( T(n) = c_0n + c_1 \)
  • possible_products: \( T(n) = c_0n^2 + n + c_1 \)
Big-O in Algorithm Analysis

• Easy to express $T(n)$ in big-O
  • drop constants and lower-order terms
• In big-O notation
  • first is $O(1)$
  • argmax is $O(n)$
  • possible_products is $O(n^2)$
• Convention for $T(n) = c$ is $O(1)$
Big-Omega (Ω)

• Recall $f(n)$ is $O(g(n))$  
  • if $f(n) \leq cg(n)$ for some constant as $n$ grows

• Big-O means $f(n)$ grows no faster than $g(n)$  
  • $g(n)$ acts as upper bound to $f(n)$’s growth rate

• What if we want to express a lower bound?

• We say $f(n)$ is $\Omega(g(n))$ if $f(n) \geq cg(n)$  
  • $f(n)$ grows no slower than $g(n)$
Big-Theta (Θ)

• What about an upper and lower bound?

• We say \( f(n) \) is \( \Theta(g(n)) \) if

\[
f(n) \text{ is } O(g(n)) \text{ and } \Omega(g(n))
\]

• \( f(n) \) grows the same as \( g(n) \) (tight-bound)
How fast is the seamcarve algorithm?

• How many seams in $c \times r$ image?
  • At each row, seam can go: Left, Right, 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 seams is
    • $c \times 3^r$

• For square image with $n$ total pixels
  • there are $\sqrt{n} \times 3^{\sqrt{n}}$ possible seams
Seamcarve

- Algorithms that try every possible solution are known *exhaustive algorithms* or *brute force algorithms*
- Exhaustive approach is to consider all $\sqrt{n} \times 3^{\sqrt{n}}$ seams and choose the least important
- What would be the big-O running time?
  - $O(\sqrt{n} \times 3^{\sqrt{n}})$: exponential and not good
Seamcarve

• What is runtime of the algorithm from last class?
  • Remember: constants don’t affect big-O runtime

• 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 times do we evaluate each pixel?
  • A constant number of times
  • So algorithm is linear: $O(n)$, $n$ is number of pixels

• Hint: we also could have looked at pseudocode and counted number of nested loops!
Seamcarve: Dynamic Programming

• From exponential algorithm to a linear algorithm?!?

• Avoided recomputing information we already calculated!
  • Many seams cross paths
  • we don’t need to recompute sums of importances if we’ve already calculated it before
  • That’s the purpose of the additional costs array

• Storing computed information to avoid recomputing later, is called dynamic programming
Fibonacci: Recursive

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

• The Fibonacci sequence is defined by the recurrence relation:

\[ F_0 = 0, \quad F_1 = 1 \]
\[ F_n = F_{n-1} + F_{n-2} \]

• This lends itself very well to a recursive function for finding the \( n^{th} \) Fibonacci number

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

The recursive function for calculating the Fibonacci Sequence can be illustrated using a tree, where each row is a level of recursion.

This diagram illustrates the calls made for fib(4).

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

• In order to calculate \texttt{fib}(4), how many times does \texttt{fib}() get called?

  \begin{itemize}
    \item \texttt{fib}(4)
      \begin{itemize}
        \item \texttt{fib}(3)
          \begin{itemize}
            \item \texttt{fib}(2)
              \begin{itemize}
                \item \texttt{fib}(1)
                  \begin{itemize}
                    \item \texttt{fib}(0)
                    \item \texttt{fib}(1)
                  \end{itemize}
                \end{itemize}
              \end{itemize}
          \end{itemize}
        \end{itemize}
  \end{itemize}

  \texttt{fib}(1) alone gets recomputed 3 times!

• At each level of recursion
  • Algorithm makes twice as many recursive calls as last
  • For \texttt{fib}(n) number of recursive calls is approx $2^n$
  • Algorithm is $O(2^n)$
Fibonacci: Dynamic Programming

- Instead of recomputing same Fibonacci numbers over and over
- We compute each one once & store it for later
- Need a table to keep track of intermediary values

```python
def 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 (2)

• 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)

• Again, we reduced runtime of algorithm 
  • From exponential to linear 
  • With dynamic programming!
Readings

• Dasgupta Section 0.2, pp 12-15
  • Goes through this Fibonacci example (although without mentioning dynamic programming)
  • This section is easily readable now

• Dasgupta Section 0.3, pp 15-17
  • Describes big-O notation

• Dasgupta Chapter 6, pp 169-199
  • Goes into Dynamic Programming
  • This chapter builds significantly on earlier ones and will be challenging to read now, but we’ll see much of it this semester.
Announcements 1/31/17

• Homework 1 due this Friday at 3pm!

• **Thursday is in-class Python lab!**
  • If you are able to work on your own laptop, go to Salomon DECI (here!). Otherwise, go to the Sunlab.
  • Make sure you can log into your CS account before attending the lab
  • See a SunLab consultant if you have any account issues!

• Sections started yesterday – if you are not signed up, you could be in trouble!