COMP20003
Algorithms and Data Structures
Complexity Analysis

Nir Lipovetzky
Department of Computing and Information Systems
University of Melbourne
Semester 2

So far…

- We have looked at one calculation (fib):
  - Obvious algorithm slow.
  - Memoization faster – but takes space.
  - Storing last values in variables – more time *and* space efficient.
- We have estimated computation time by counting operations.

Outline of the first few lectures

- Algorithms: general
- This subject: details
- Algorithm efficiency
- → Computational complexity
- Data structures
  - Basic data structures
  - Algorithms on basic data structures
  - Complexity analysis of algorithms on basic ds's

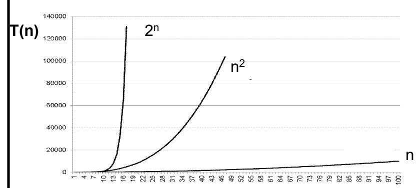
This lecture

- Formalize approach:
  - Characterize run time of any algorithm
    - Identify the most expensive operation.
    - Count that operation.
    - Express in terms of input size *n*.

Textboook

- Skiena: Chapter 2, Algorithm Analysis

Why is complexity analysis important?



T(n) is a function of *n*

- Can grow very large as *n* grows

We want to know this *before* we code

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Data given assume every operation takes 1 nanosec.
Data from Skiena Lecture Notes
http://www.cs.suny.edu.au/~skiena

Big-O definition

- For two functions *f(n)* and *g(n)*, we say that *f(n)* is in *O(g(n))* if:
  - There are constants c_0 and N_0, such that $f(N) < c_0 * g(N)$ for all $N > N_0$.

Big-O definition

- For two functions *f(n)* and *g(n)*, we say that *f(n)* is in *O(g(n))* if:
  - There are constants c_0 and N_0, such that $f(N) < c_0 * g(N)$ for all $N > N_0$.
- Notice:
  - We are only interested in large N, $N > N_0$.

Big-O definition

- For two functions *f(n)* and *g(n)*, we say that *f(n)* is in *O(g(n))* if:
  - There are constants c_0 and N_0, such that $f(N) < c_0 * g(N)$ for all $N > N_0$.
- Examples:
  - $x^2 + 33$ is in $O(x^2)$
  - $x^2 + 33x + 17$ is in $O(x^2)$
  - $15x^2 + 33x + 17$ is in $O(x^2)$

Exercizes

- $x^2 + 33$ is in $O(x^2)$
  - For c_0 = ?, N_0 = ? : $X^2 + 33 < c_0 x^2$ *for all* $N > N_0$
- $x^2 + 33x + 17$ is in $O(x^2)$
  - For c_0 = ?, N_0 = ? $X^2 + 33x + 17 < c_0 x^2$
- $15x^2 + 33x + 17$ is in $O(x^2)$
  - For c_0 = ?, N_0 = ? $15X^2 + 33x + 17 < c_0 x^2$

Big-O heuristics

- Examples:
  - $x^2 + 33$ is in $O(x^2)$
  - $x^2 + 33x + 17$ is in $O(x^2)$
  - $15x^2 + 33x + 17$ is in $O(x^2)$
- Easy way to classify functions into big-O
  - Drop the lower order terms.
  - Forget about constants.

Why?

- Why can we drop constants and lower order terms?

Terminology

- Examples:
  - $x^2 + 33$ is in $O(x^2)$
  - $x^2 + 33x + 17$ is in $O(x^2)$
  - $15x^2 + 33x + 17$ is in $O(x^2)$
- Actually all these are also in $O(x^3)$…
- … and in $O(2^n)$…..
- But we are usually most interested in the closest bound.

Big-O

- Easy way to classify functions into big-O
  - Drop the lower order terms.
  - Forget about constants.
- What does this give us?
  - A *theoretical* way to compare growth rate.
  - Machine-independent
  - Ignoring constants – not completely *practical*.

Big-O arithmetic

- If a program is in stages:
  - Stage 1 operates on m inputs, is linear O(m)
  - Then Stage 2 operates on n inputs, is linear O(n)
  - Whole program is
    - O(m) + O(n) = O(max(m,n)) ← Big-O Addition
    - If m << n, then O(n)
- If the program operates on each of n inputs m times, program is
    - O(m) * O(n) = O(m*n) ← Big-O Multiplication

Big-O hierarchy

- Dominance Relation
  - n! >> 2^n >> n^3 >> n^2 >> n log n
  - n log n >> n >> log n >> 1
- The base of *log n* doesn't matter, because:
  - Changing base of $\log_a n \rightarrow \log_c n$?
    - $\text{Log}_c n = \log_a n * \log_c a$
    - $\text{Log}_c a$ is a constant and is lost in Big-O notation
  - Doesn't make a big difference:
    - $\text{Log}_2(10^6) = 19.9$ $\text{Log}_3(10^6) = 12.5$ $\text{Log}_{100}(10^6) = 3$

Workshops

- Workshops start this week.
- If you haven't been able to enrol, just attend a convenient workshop.
  - To register, send e-mail to madalain@unimelb.edu.au
- Workshops are a great place to clarify concepts and ask questions!

Unix from the student labs

- ssh dimefox.eng.unimelb.edu.au (or nutmeg.eng.unimelb.edu.au)
- mkdir <dir_name>
- cd <dir_name>
- ls
- touch <filename>
- less <filename>
- MobaXterm (or other) editor --- write your program, remember to save!
- gcc <filename>
- a.out
- gcc –o <program_filename> <filename>
- ./<program_filename>
  - More on GCC: https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html

So far….

- Computational complexity so far
  - Intuitive: Fibonacci
  - Big-O as upper bound
    - Formal Definition
    - Calculation – unrolling the loop
    - Discarding constants
    - Discarding lower order terms
- Now
  - More complicated big-O arithmetic
  - Θ- and Ω- notation

This lecture

- Big-O examples and fine points
- Other bounds: O() *vs.* Ω() *vs.* Θ()
- Average case *vs.* worst case
- Concrete analysis of algorithms on basic data structures

Big-O addition

- Loop:

```
for(i=0;i<m; i++)
{
        printf("%d\n",i);
}
for(i=0;i<n;i++)
{
        printf("%d\n",i);
}
```


Big-O multiplication

- Loop:

```
for(i=0;i<m; i++)
{
        for(j=0;j<n;j++)
        {
                printf("%d-%d\n",i,j);
        }
}
```


Big-O arithmetic

- Successive operations add:
  - O(m) + O(n) = O(m+n)
- Single loops multiply:
  - O(m)*O(n) = O(mn)
- Smaller variables can drop out:
  - For n>>m, O(m+n) = O(n)

Nested loops

```
for(i=0;i<m; i++)
{
        for(j=0;j<n;j++)
        {
                for(k=0;k<p;k++
                {
                        printf("%d-%d-%d\n",i,j,k);
                } /* for k */
        } /* for j */
}  /* for i */
```

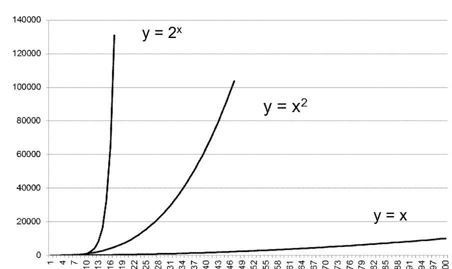

Lower order terms

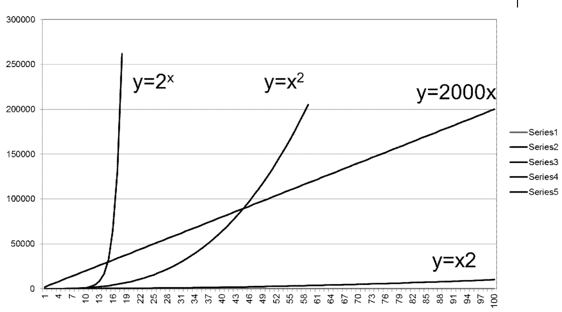
- Previously we showed $x^2 + 3x$ is in $O(x^2)$
- We can drop the 3x lower order term
- Useful for big-O analysis:
  - $n! >> 2^n >> n^3 >> n^2 >> n \log n >> n >> \log n >> 1$

For Small n

- Do we care?

Growth rate of functions





Big-O is an upper bound

- f(n) is O(g(n)) means

  $f(N) < c_0 * g(N)$ for all $N > N_0$
- g(N) is an upper bound:
  - y=x is in O(f(x))
    - Note: it is also in $O(f(x^2))$, BUT
- Usually we use O() to mean the lowest upper bound, but by definition it is really *any* upper bound.

Exercizes

- What is the difference between:
  - $O(\log_2 N)$ and $O(\log_{10} N)$?
  - $O(\log_2 N)$ and $O(\log_2 N^2)$?
- What is the complexity of a 2-stage algorithm where stage 1 is in $O(n^2)$ and stage2 is in O(m)?
- Is 2^{n+1} in $O(2^n)$?
- Is $(n+1)^5$ in $O(n^5)$?

More exercizes

- Show that big-O relationships are transitive, *i.e.* that
  - If f(n) = O(g(n)), and
  - g(n) = O(h(n)), then
  - f(n) = O(h(n))

" = " is an accepted abuse of notation

Big-Omega is a *lower* bound

- Upper bound: O(g(n))
  - f(n) is O(g(n)) : $f(N) < c_0 * g(N)$ for all $N > N_0$
  - 17x is O(x), 17x is also O(x^2)
- Lower bound: $\Omega(g(n))$
  - f(n) is Ω(g(n)) if g(n) is O(f(n))
  - x is $\Omega(x)$, x^2 is $\Omega(x)$

Big-Theta is the growth rate

- Tight bound: $\Theta(g(n))$
  - f(n) is $\Theta(g(n))$ when
  - f(n) is O(g(n)) *and* f(n) is Ω(g(n))
- Example:
  - $f(x) = x^2$ is:
    - $O(x^2)$, $O(x^3)$, $O(2^x)$….
    - $\Omega(x)$, $\Omega(x^2)$, $\Omega(1)$
    - $\Theta(x^2)$

Examples

- *Given the following functions f(n) and g(n), is f in O(g(n)) or is f in Ω(g(n)), or both?*

f(n)	g(n)
n + 100	n + 200
$\log_2 n$	$\log_{10} n$
2^n	2^{n+1}

Average, worst, and best case analysis

- Given an unsorted list or array of items, searching for one item require looking at:
  - n items in the worst case
  - n/2 items on average
  - 1 item if you are lucky
- Average case and worst case analysis are useful.

Average, worst, and best case analysis

- Average case analysis is often difficult!
  - Have to average over all possible inputs
- Worst case analysis and big-O are the most useful and the most widely used!

"Every science has a big lie. The big lie of complexity is worst case analysis." [C. Papadimitriou]

Skiena: The Algorithm Design Manual

- Chapter 2: Sections 2.1 through 2.4

- Next section:
  - Simple data structures and algorithms.
  - Complexity analysis with concrete examples.