

## CS61B Lecture #16: Complexity

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 1

## What Are the Questions?

- Cost is a principal concern throughout engineering:
  - "An engineer is someone who can do for a dime what any fool can do for a dollar."
- Cost can mean
  - *Operational cost* (for programs, time to run, space requirements).
  - *Development costs*: How much engineering time? When delivered?
  - *Maintenance costs*: Upgrades, bug fixes.
  - *Costs of failure*: How robust? How safe?
- Is this program *fast enough*? Depends on:
  - *For what purpose*;
  - *For what input data*.
- How much *space* (memory, disk space)?
  - Again depends on what input data.
- How will it *scale*, as input gets big?

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 2

## Enlightening Example

**Problem:** Scan a text corpus (say  $10^8$  bytes or so), and find and print the 20 most frequently used words, together with counts of how often they occur.

- Solution 1 (Knuth): Heavy-Duty data structures
  - Hash Trie implementation, randomized placement, pointers galore, several pages long.
- Solution 2 (Doug McIlroy): UNIX shell script:

```
tr -c -s '[:alpha:]' '\n*' < FILE | \
sort | \
uniq -c | \
sort -n -r -k 1,1 | \
sed 20q
```
- Which is better?
  - #1 is much faster,
  - but #2 took 5 minutes to write and processes 100MB in  $\approx 50$  sec.
  - I pick #2.
- In very many cases, almost anything will do: **Keep It Simple.**

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 3

## Cost Measures (Time)

- *Wall-clock or execution time*
  - You can do this at home:

```
time java FindPrimes 1000
```
  - Advantages: easy to measure, meaning is obvious.
  - Appropriate where time is critical (real-time systems, e.g.).
  - Disadvantages: applies only to specific data set, compiler, machine, etc.
- *Dynamic statement counts* of # of times statements are executed:
  - Advantages: more general (not sensitive to speed of machine).
  - Disadvantages: doesn't tell you actual time, still applies only to specific data sets.
- *Symbolic execution times*:
  - That is, *formulas* for execution times as functions of input size.
  - Advantages: applies to all inputs, makes scaling clear.
  - Disadvantage: practical formula must be approximate, may tell very little about actual time.

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 4

## Asymptotic Cost

- Symbolic execution time lets us see *shape* of the cost function.
- Since we are approximating anyway, pointless to be precise about certain things:
  - *Behavior on small inputs*:
    - \* Can always pre-calculate some results.
    - \* Times for small inputs not usually important.
    - \* Often more interested in *asymptotic behavior* as input size becomes very large.
  - *Constant factors* (as in "off by factor of 2"):
    - \* Just changing machines causes constant-factor change.
- How to abstract away from (i.e., ignore) these things?

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 5

## Handy Tool: Order Notation

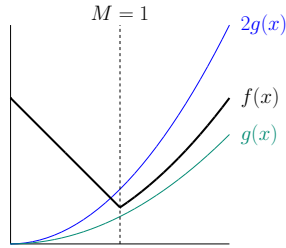
- Idea: Don't try to produce specific functions that specify size, but rather *families of functions with similarly behaved magnitudes*.
- Then say something like "*f* is bounded by *g* if it is in *g*'s family."
- For any function  $g(x)$ , the functions  $2g(x)$ ,  $0.5g(x)$ , or for any  $K > 0$ ,  $K \cdot g(x)$ , all have the same "shape". So put all of them into *g*'s family.
- Any function  $h(x)$  such that  $h(x) = K \cdot g(x)$  for  $x > M$  (for some constant  $M$ ) has *g*'s shape "except for small values." So put all of these in *g*'s family.
- For upper limits, throw in all functions whose absolute value is everywhere  $\leq$  some member of *g*'s family. Call this set  $O(g)$  or  $O(g(n))$ .
- Or, for lower limits, throw in all functions whose absolute value is everywhere  $\geq$  some member of *g*'s family. Call this set  $\Omega(g)$ .
- Finally, define  $\Theta(g) = O(g) \cap \Omega(g)$ —the set of functions *bracketed in magnitude by* two members of *g*'s family.

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 6

## Big Oh

- Goal: Specify bounding from above.



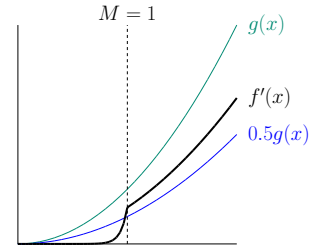
- Here,  $f(x) \leq 2g(x)$  as long as  $x > 1$ ,
- So  $f(x)$  is in  $g$ 's "bounded-above family," written  $f(x) \in O(g(x))$ ,
- ... even though (in this case)  $f(x) > g(x)$  everywhere.

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 7

## Big Omega

- Goal: Specify bounding from below:



- Here,  $f'(x) \geq \frac{1}{2}g(x)$  as long as  $x > 1$ ,
- So  $f'(x)$  is in  $g$ 's "bounded-below family," written  $f'(x) \in \Omega(g(x))$ ,
- ... even though  $f(x) < g(x)$  everywhere.

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 8

## Big Theta

- In the two previous slides, we not only have  $f(x) \in O(g(x))$  and  $f'(x) \in \Omega(g(x))$ ,...
- ... but also  $f(x) \in \Omega(g(x))$  and  $f'(x) \in O(g(x))$ .
- We can summarize this all by saying  $f(x) \in \Theta(g(x))$  and  $f'(x) \in \Theta(g(x))$ .

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 9

## Aside: Various Mathematical Pedantry

- Technically, if I am going to talk about  $O(\cdot)$ ,  $\Omega(\cdot)$  and  $\Theta(\cdot)$  as sets of functions, I really should write, for example,  $f \in O(g)$  instead of  $f(x) \in O(g(x))$
- In effect,  $f(x) \in O(g(x))$  is short for  $\lambda x. f(x) \in O(\lambda x. g(x))$ .
- The standard notation outside this course, in fact, is  $f(x) = O(g(x))$ , but personally, I think that's a serious abuse of notation.

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 10

## How We Use Order Notation

- Elsewhere in mathematics, you'll see  $O(\dots)$ , etc., used generally to specify bounds on functions.
- For example,

$$\pi(N) = \Theta\left(\frac{N}{\ln N}\right)$$

which I would prefer to write

$$\pi(N) \in \Theta\left(\frac{N}{\ln N}\right)$$

(Here,  $\pi(N)$  is the number of primes less than or equal to  $N$ .)

- Also, you'll see things like

$$f(x) = x^3 + x^2 + O(x) \quad (\text{or } f(x) \in x^3 + x^2 + O(x)),$$

meaning that  $f(x) = x^3 + x^2 + g(x)$  where  $g(x) \in O(x)$ .

- For our purposes, the functions we will be bounding will be **cost functions**: functions that measure the amount of execution time or the amount of space required by a program or algorithm.

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 11

## Why It Matters

- Computer scientists often talk as if constant factors didn't matter at all, only the difference of  $\Theta(N)$  vs.  $\Theta(N^2)$ .
- In reality they do matter, but at some point, constants always get swamped.

$n$	$16 \lg n$	$\sqrt{n}$	$n$	$n \lg n$	$n^2$	$n^3$	$2^n$
2	16	1.4	2	2	4	8	4
4	32	2	4	8	16	64	16
8	48	2.8	8	24	64	512	256
16	64	4	16	64	256	4,096	65,536
32	80	5.7	32	160	1024	32,768	$4.2 \times 10^9$
64	96	8	64	384	4,096	262,144	$1.8 \times 10^{19}$
128	112	11	128	896	16,384	$2.1 \times 10^9$	$3.4 \times 10^{38}$
:	:	:	:	:	:	:	:
1,024	160	32	1,024	10,240	$1.0 \times 10^6$	$1.1 \times 10^9$	$1.8 \times 10^{308}$
:	:	:	:	:	:	:	:
$2^{20}$	320	1024	$1.0 \times 10^6$	$2.1 \times 10^7$	$1.1 \times 10^{12}$	$1.2 \times 10^{18}$	$6.7 \times 10^{315.652}$

- For example: replace column  $n^2$  with  $10^6 \cdot n^2$  and it still becomes dominated by  $2^n$ .

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 12

## Some Intuition on Meaning of Growth

- How big a problem can you solve in a given time?
- In the following table, left column shows time in microseconds to solve a given problem as a function of problem size  $N$ .
- Entries show the *size of problem* that can be solved in a second, hour, month (31 days), and century, for various relationships between time required and problem size.
- $N$  = problem size.

Time ( $\mu$ sec) for problem size $N$	Max $N$ Possible in			
	1 second	1 hour	1 month	1 century
$\lg N$	$10^{300000}$	$10^{1000000000}$	$10^{8 \cdot 10^{11}}$	$10^{10^{14}}$
$N$	$10^6$	$3.6 \cdot 10^9$	$2.7 \cdot 10^{12}$	$3.2 \cdot 10^{15}$
$N \lg N$	63000	$1.3 \cdot 10^8$	$7.4 \cdot 10^{10}$	$6.9 \cdot 10^{13}$
$N^2$	1000	60000	$1.6 \cdot 10^6$	$5.6 \cdot 10^7$
$N^3$	100	1500	14000	150000
$2^N$	20	32	41	51

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 13

## Using the Notation

- Can use this order notation for any kind of real-valued function.
- We will use them to describe cost functions. Example:
 

```
/** Find position of X in list L, or -1 if not found. */
int find(List L, Object X) {
    int c;
    for (c = 0; L != null; L = L.next, c += 1)
        if (X.equals(L.head)) return c;
    return -1;
}
```
- Choose representative operation: number of `.equals` tests.
- If  $N$  is length of  $L$ , then loop does *at most*  $N$  tests: *worst-case time* is  $N$  tests.
- In fact, total # of instructions executed is roughly proportional to  $N$  in the worst case, so can also say worst-case time is  $O(N)$ , regardless of units used to measure.
- Use  $N > M$  provision (in defn. of  $O(\cdot)$ ) to ignore empty list.

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 14

## Be Careful

- It's also true that the worst-case time is  $O(N^2)$ , since  $N \in O(N^2)$  also: Big-Oh bounds are loose.
- The worst-case time is  $\Omega(N)$ , since  $N \in \Omega(N)$ , but that does *not* mean that the loop *always* takes time  $N$ , or even  $K \cdot N$  for some  $K$ .
- Instead, we are just saying something about the *function* that maps  $N$  into the *largest possible* time required to process any array of length  $N$ .
- To say as much as possible about our worst-case time, we should try to give a  $\Theta$  bound: in this case, we can:  $\Theta(N)$ .
- But again, that still tells us nothing about *best-case* time, which happens when we find  $X$  at the beginning of the loop. Best-case time is  $\Theta(1)$ .

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 15

## Effect of Nested Loops

- Nested loops often lead to polynomial bounds:
 

```
for (int i = 0; i < A.length; i += 1)
    for (int j = 0; j < A.length; j += 1)
        if (i != j && A[i] == A[j])
            return true;
return false;
```
- Clearly, time is  $O(N^2)$ , where  $N = A.length$ . *Worst-case time* is  $\Theta(N^2)$ .
- Loop is inefficient though:
 

```
for (int i = 0; i < A.length; i += 1)
    for (int j = i+1; j < A.length; j += 1)
        if (A[i] == A[j]) return true;
return false;
```
- Now worst-case time is proportional to
 
$$N - 1 + N - 2 + \dots + 1 = N(N - 1)/2 \in \Theta(N^2)$$
 (so asymptotic time unchanged by the constant factor).

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 16

## Recursion and Recurrences: Fast Growth

- Silly example of recursion. In the worst case, both recursive calls happen:

```
/** True iff X is a substring of S */
boolean occurs(String S, String X) {
    if (S.equals(X)) return true;
    if (S.length() <= X.length()) return false;
    return
        occurs(S.substring(1), X) ||
        occurs(S.substring(0, S.length()-1), X);
}
```

- Define  $C(N)$  to be the worst-case cost of `occurs(S,X)` for  $S$  of length  $N$ ,  $X$  of fixed size  $N_0$ , measured in # of calls to `occurs`. Then

$$C(N) = \begin{cases} 1, & \text{if } N \leq N_0, \\ 2C(N-1) + 1 & \text{if } N > N_0 \end{cases}$$

- So  $C(N)$  grows exponentially:

$$C(N) = 2C(N-1) + 1 = 2(2C(N-2) + 1) + 1 = \dots = \underbrace{2(\dots 2 \cdot 1 + 1)}_{N-N_0} + \dots + 1 = 2^{N-N_0} + 2^{N-N_0-1} + 2^{N-N_0-2} + \dots + 1 = 2^{N-N_0+1} - 1 \in \Theta(2^N)$$

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 17

## Binary Search: Slow Growth

```
/** True X iff is an element of S[L..U]. Assumes
 * S in ascending order, 0 <= L <= U-1 < S.length. */
boolean isIn(String X, String[] S, int L, int U) {
    if (L > U) return false;
    int M = (L+U)/2;
    int direct = X.compareTo(S[M]);
    if (direct < 0) return isIn(X, S, L, M-1);
    else if (direct > 0) return isIn(X, S, M+1, U);
    else return true;
}
```

- Here, worst-case time,  $C(D)$ , (as measured by # of calls to `.compareTo`), depends on size  $D = U - L + 1$ .
- We eliminate  $S[M]$  from consideration each time and look at half the rest. Assume  $D = 2^k - 1$  for simplicity, so:

$$C(D) = \begin{cases} 0, & \text{if } D \leq 0, \\ 1 + C((D-1)/2), & \text{if } D > 0. \end{cases} = \underbrace{1+1+\dots+1}_k + 0 = k = \lg(D+1) \in \Theta(\lg D)$$

Last modified: Thu Feb 27 23:10:03 2020

CS61B: Lecture #16 18

## Another Typical Pattern: Merge Sort

```
List sort(List L) {  
  if (L.length() < 2) return L;  
  Split L into L0 and L1 of about equal size;  
  L0 = sort(L0); L1 = sort(L1);  
  return Merge of L0 and L1  
}
```

Merge ("combine into a single ordered list") takes time proportional to size of its result.

- Assuming that size of L is  $N = 2^k$ , worst-case cost function,  $C(N)$ , counting just merge time (which is proportional to # items merged):

$$\begin{aligned} C(N) &= \begin{cases} 0, & \text{if } N < 2; \\ 2C(N/2) + N, & \text{if } N \geq 2. \end{cases} \\ &= 2(2C(N/4) + N/2) + N \\ &= 4C(N/4) + N + N \\ &= 8C(N/8) + N + N + N \\ &= N \cdot 0 + \underbrace{N + N + \dots + N}_{k=\lg N} \\ &= N \lg N \end{aligned}$$

- In general, can say it's  $\Theta(N \lg N)$  for arbitrary  $N$  (not just  $2^k$ ).