

CS61B Lecture #15

Announcements:

- Please use `bug-submit` for code problems.
- Watch the newsgroup and class web site for updates, hints, useful new utilities, etc.

Readings for Today: *Data Structures (Into Java), Chapter 1;*

Readings for next Topics: *Data Structures, Chapter 2-4*

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?
 - Costs of failure: How robust? How safe?
- Is this program fast enough? Depends on:
 - *For what purpose;*
 - *What input data.*
- How much space (memory, disk space)?
 - Again depends on what input data.
- How will it scale, as input gets big?

Enlightening Example

Problem: Scan a text corpus (say 10^7 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 20MB in 1 minute.
 - I pick #2.
- In most cases, anything will do: Keep It Simple.

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.

- Number of times certain 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 or statement counts in terms of input size.
- Advantages: applies to all inputs, makes scaling clear.
- Disadvantage: practical formula must be approximate, may tell very little about actual time.

Asymptotic Cost

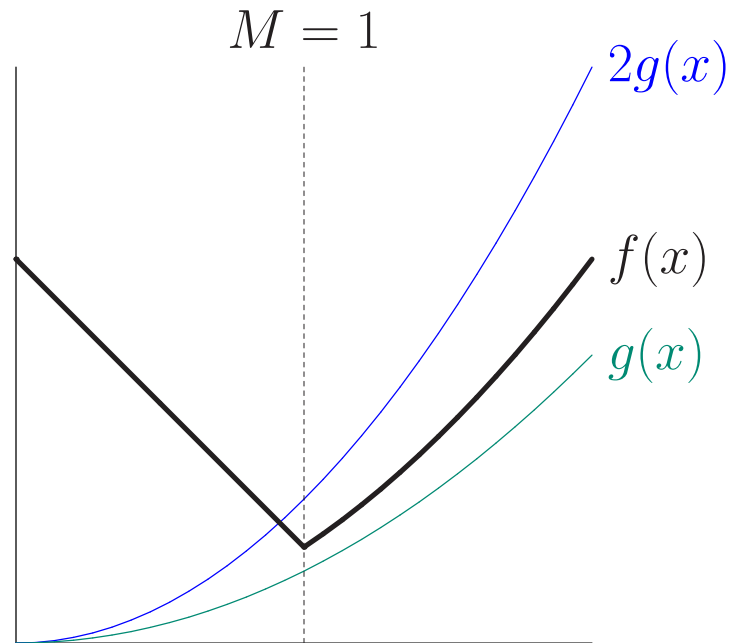
- Symbolic execution time lets us see *shape* of the cost function.
- Since we are approximating anyway, pointless to be precise about certain things:
 - Behavior on small inputs:
 - * Can always pre-calculate some results.
 - * Times for small inputs not usually important.
 - Constant factors (as in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

Handy Tool: Order Notation

- Idea: Don't try to produce *specific* functions that specify size, but rather *families* of *similar* functions.
- Say something like " f is bounded by g if it is in g 's family."
- For any function $g(x)$, the functions $2g(x)$, $1000g(x)$, 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.
- If we want upper limits, throw in all functions that are everywhere \leq some other member of g 's family. Call this family $O(g)$ or $O(g(n))$.
- Or, if we want lower limits, throw in all functions that are everywhere \geq some other member of g 's family. Call this family $\Omega(g)$.
- Finally, define $\Theta(g) = O(g) \cap \Omega(g)$ —the set of functions *bracketed* by members of g 's family.

Big Oh

- Goal: Specify bounding from above.



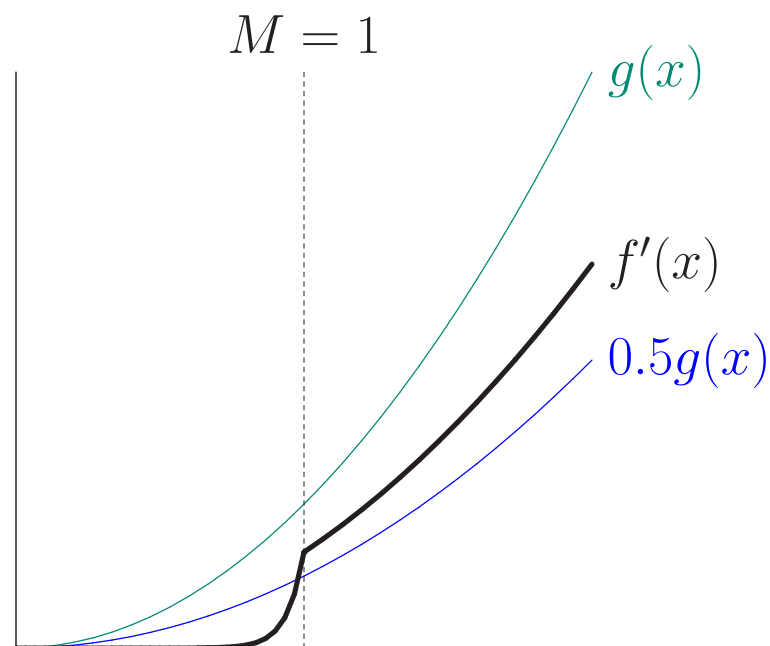
- Here, $f(x) \leq 2g(x)$ as long as $x > 1$,
- So $f(x)$ is in g 's upper-bound family, written

$$f(x) \in O(g(x)),$$

- ...even though $f(x) > g(x)$ everywhere.

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 lower-bound family, written
$$f'(x) \in \Omega(g(x)),$$
- ...even though $f(x) < g(x)$ everywhere.
- In fact, we also have $f'(x) \in O(g(x))$ and $f(x) \in \Omega(g(x))$ and so we can also write

$$f(x), f'(x) \in \Theta(g(x)).$$

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, but we still have a point: at some point, constants 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	65,636
32	80	5.7	32	160	1024	32,768	4.2×10^9
64	96	8	64	384	4,096	262,144	1.8×10^{19}
128	112	11	128	896	16,384	2.1×10^9	3.4×10^{38}
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
1,024	160	32	1,024	10,240	1.0×10^6	1.1×10^9	1.8×10^{308}
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
2^{20}	320	1024	1.0×10^6	2.1×10^7	1.1×10^{12}	1.2×10^{18}	$6.7 \times 10^{315,652}$

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 (μ sec) for problem size N	1 second	Max N Possible in 1 hour	1 month	1 century
$\lg N$	10^{300000}	$10^{10000000000}$	$10^{8 \cdot 10^{11}}$	$10^{9 \cdot 10^{14}}$
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

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. Return -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 handle empty list.

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 an array of length N .
- To say as much as possible about our worst-case time, we should try to give a Θ 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 $\Theta(1)$.

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).

Recursion and Recurrences: Fast Growth

- Silly example of recursion:

```
/** 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);
}
```

- In the worst case, both recursive calls happen.
- 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:

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

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 string comparisons), 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:

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

Another Typical Pattern: Merge Sort

<pre>List sort (List L) { if (L.length () < 2) return L; <i>Split L into L0 and L1 of about equal size;</i> L0 = sort (L0); L1 = sort (L1); return <i>Merge of L0 and L1</i> }</pre>	$\left\{ \begin{array}{l} \text{Merge ("combine into a single, ordered list")} \text{ takes} \\ \text{time proportional to size of} \\ \text{its result.} \end{array} \right.$
--	--

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

$$\begin{aligned} C(N) &= \begin{cases} 1, & \text{if } N < 2; \\ 2C(N/2) + N, & \text{if } N \geq 2. \end{cases} \\ &= 2(2C(N/4) + N/2) + N \\ &= 4C(N/4) + N + N \\ &= 8C(N/8) + N + N + N \\ &= N \cdot 1 + \underbrace{N + N + \dots + N}_{k=\lg N} \\ &= N + N \lg N \in \Theta(N \lg N) \end{aligned}$$

- In general, $\Theta(N \lg N)$ for arbitrary N (not just 2^k).

Amortization: Expanding Vectors

- When using array for expanding sequence, best to double size of array to grow it. Here's why.
- If array is size s , doubling its size and moving s elements to the new array takes time $\propto 2s$.
- Cost of inserting N items into array, doubling size as needed, starting with array size 1:

To Insert Item #	Resizing Cost	Cumulative Cost	Resizing Cost per Item	Array Size After Insertions
1	0	0	0	1
2	2	2	1	2
3 to 4	4	6	1.5	4
5 to 8	8	14	1.75	8
\vdots	\vdots	\vdots	\vdots	\vdots
$2^m + 1$ to 2^{m+1}	2^{m+1}	$2^{m+2} - 2$	≈ 2	2^{m+1}

- If we spread out (amortize) the cost of resizing, we average about 2 time units on each item: "amortized insertion time is 2 units."
- So even though worst-case time for adding *one* element to array of N elements is $2N$, time to add N elements is $\Theta(N)$, *not* $\Theta(N^2)$.

Demonstrating Amortized Time: Potential Method

- To formalize the argument, associate a *potential*, $\Phi_i \geq 0$, to the i^{th} operation that keeps track of “saved up” time from cheap operations that we can “spend” on later expensive ones. Start with $\Phi_0 = 0$.
- Now define the *amortized cost* of the i^{th} operation as

$$a_i = c_i + \Phi_{i+1} - \Phi_i,$$

where c_i is the real cost of the operation.

- On cheap operations, we artificially set $a_i > c_i$ and increase Φ ($\Phi_{i+1} > \Phi_i$).
- On expensive ones, we typically have $a_i \ll c_i$ and greatly decrease Φ (but don't let it go negative—may not be “overdrawn”).
- We try to do all this so that a_i remains as we desired (e.g., $O(1)$ for expanding array), without allowing $\Phi_i < 0$.
- Requires that we choose a_i so that Φ_i always stays ahead of c_i .

Application to Expanding Arrays

- When adding to our array, the cost, c_i , of adding element $\#i$ when the array already has space for it is 1 unit.
- The array does not initially have space when adding items 1, 2, 4, 8, 16, ... —in other words at item 2^n for all $n \geq 0$. So,
 - $c_i = 1$ if $i \geq 0$ and is not a power of 2; and
 - $c_i = 3i + 1$ (allocate $2i$ items, copy i items, and then add item $\#i$) when i is a power of 2.
- So on each operation $\#2^n$ we're going to need to have saved up at least $3 \cdot 2^n$ units of potential to cover the expense, and we have the preceding 2^{n-1} operations to do it (the ones since the preceding doubling operation).
- To do so, just choose $a_i = 7$ (or could let $a_0 = 1, a_1 = 4$)
- Here's what happens:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
c_i	1	4	7	1	13	1	1	1	25	1	1	1	1	1	1	1	49
a_i	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
Φ_i	0	6	9	9	15	9	15	21	27	9	15	21	27	33	39	45	51