Big-O-ology

Jim Royer

EECS

September 26, 2016

How do you tell how fast a program is?

Answer? Run some test cases.

Problem You can only run a few test cases.
There will be many inputs you won't test
. . . and **BAD** things may be happening on that stuff.

Fix? Test all possible inputs! HA!

Example With a 100 element int array as input with 32 bit ints,
there are $100 \cdot 32 = 3200$ bits in the input.
Each bit can be 0 or 1.
So there 2^{3200} possible inputs where . . .

$2^{3200} =$

```
  1 976 906 478 982 563 993 654 226 439 837 963 340 315 390 682 625 773
828 918 265 710 158 340 601 093 951 126 756 295 848 974 613 063 099 294
244 703 164 628 428 967 968 057 547 050 608 904 859 234 600 159 014 229
329 102 195 101 574 081 057 061 661 948 106 884 800 321 129 818 693 914
608 845 281 661 462 333 814 326 544 389 741 164 009 367 602 548 103 882
724 187 831 587 394 954 463 183 137 735 657 307 019 637 359 169 290 834
318 700 453 890 617 892 714 561 362 370 427 388 384 101 316 010 134 426
924 662 084 888 461 376 218 489 653 794 242 999 053 891 151 382 465 888
482 003 300 085 676 110 173 467 997 003 494 159 830 094 271 947 506 024
974 271 953 414 706 038 068 210 170 338 961 663 202 839 203 641 120 865
263 292 248 718 692 924 915 189 291 455 200 665 479 606 951 612 257 868
495 299 167 071 771 306 894 428 954 788 679 149 900 427 954 823 300 393
640 007 649 397 742 106 635 573 828 425 752 730 305 375 232 721 339 803
871 889 299 281 134 208 211 131 341 001 135 605 446 809 477 409 979 279
627 213 188 610 112 867 929 569 789 492 640 465 736 633 925 065 052 540
962 862 027 736 312 499 143 902 692 033 755 536 952 046 162 410 311 395
501 619 568 814 547 777 271 031 259 247 973 250 866 583 116 853 615 908
352 881 305 587 297 178 183 145 388 745 781 297 002 238 181 376
```

So you can't test that many inputs!

How do you tell how fast a program is? *(2nd Try)*

Another answer: Do some run time analysis.

A simple, sample method.

```
public static int findMin(int[] a) {
   // PRECONDITION: a.length > 0
   // RETURNS: the minimal value in the array a
   int n = a.length;
   int minVal = a[0];

   for (int j=1; j < n; j++)
      if (a[j]<minVal) { minVal = a[j]; }

   return minVal;
}
```

Q: How much time does this take?

How much time does findMin take?

Problems with the question:

1. Different size arrays will produce different run times.
   Example
   You would expect findMin to take longer on a length 100,000 array than on a length 3 array.
2. On a particular input, you will see different run times under:
   - different machines
   - different Java compilers
   - different Java run-time systems

Dealing with Problem 1: Different size arrays produce different run times

Obvious fix:

- Make the answer depend on the length of the array:
  $T(n)$ = the run time for a length n array
- Finding $T(n)$ *exactly* is usually **hard**.
- But an *estimate* is almost always good enough.

WARNING: Oversimplified.

Dealing with Problem 2: Different setups produce different run times

Assumption 1: Proportionality

Given two setups, there are constants c_{low} and c_{high} such that

$$c_{low} \leq \frac{\text{the run-time for setup 1 on a size n input}}{\text{the run-time for setup 2 on a size n input}} \leq c_{high}.$$

Example

- Suppose we have two computers, a new WONDERBOX CLOUD 9 and an old WONDERBOX 4.
- We expect programs on the newer machine to be between 2.5 and 3.8 times faster than the old one.

WARNING: Also oversimplified.

Dealing with Problem 2: Different setups produce different run times

Assumption 2: Straight-line code takes constant time

- Straight-line code = code with no loops, recursions
- The constant depends on the setup (of machine, compiler, ...)

Example (Code: straight-line and otherwise)

✓
```
if (a[j] < minVal) { minVal = a[j]; }
```
✓
```
n = a.length; minVal = a[0];
```
✗
```
for (j=1; j < n; j++)
    if (a[j] < minVal) { minVal = a[j]; }
```

Back to: How much time does findMin take?

```
public static int findMin(int[] a) {
   int n = a.length, minVal = a[0];
   for (int j=1; j < n; j++)
      if (a[j] < minVal) { minVal = a[j]; }                    (⋆)
   return minVal;
}
```

- $(\star)$ is the straight line code that is buried deepest in for-loops.
- For a given value of n, $(\star)$ is executed $n-1$ times.
- So, for a given setup, there are constants c_{low} and c_{high} $\ni$

$$c_{\text{low}} \cdot (n-1) \quad \leq \quad \text{the run time of findMin} \quad \leq \quad c_{\text{high}} \cdot (n-1).$$

- Use testing to get estimates for c_{low} and c_{high}

Q: How do we account for the time taken by:

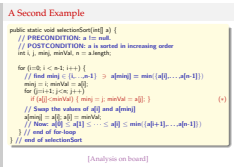
```
int n = a.length, minVal = a[0];
```


A Second Example

```
public static void selectionSort(int[] a) {
   // PRECONDITION: a != null.
   // POSTCONDITION: a is sorted in increasing order
   int i, j, minj, minVal, n = a.length;

   for (i=0; i < n-1; i++) {
      // find minj ∈ {i,...,n-1}  ∋  a[minj] = min({a[i],...,a[n-1]})
      minj = i; minVal = a[i];
      for (j=i+1; j<n; j++)
         if (a[j]<minVal) { minj = j; minVal = a[j]; }          (∗)
      // Swap the values of a[i] and a[minj]
      a[minj] = a[i]; a[i] = minVal;
      // Now: a[0] ≤ a[1] ≤ ··· ≤ a[i] ≤ min({a[i+1],...,a[n-1]})
   } // end of for-loop
} // end of selectionSort
```

[Analysis on board]



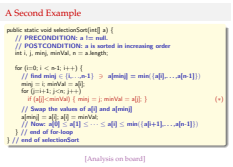
Step 1: Identify the pieces of straightline code that are buried deepest.
In `selectionSort` this is:

```
if (a[j]<minVal) { minJ = j; minVal = a[j];}
```

since it occurs within both loops. Whereas both

- `minJ = i; minVal = a[i];`
- `a[minJ] = a[i]; a[i] = minVal;`

occur within only the outermost loop.



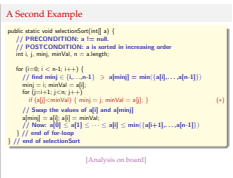
Step 2: Count how many times these innermost pieces of the program are executed for a given value of n.
We handle the two loops in `selectionSort` one at a time, starting with the innermost.

THE INNERMOST LOOP: This is

```
for (j=i+1; j<n; j++)
  if (a[j]<minVal) {minJ = j; minVal = a[j];}
```

So:

- the first iteration has $j = i+1$, the last iteration has $j = n-1$, and j increases by 1 with each iteration.
- there are $(n-1)-(i+1)+1 = n-i-1$ many iterations.
- for particular values of `i` and `n` the innermost code is executed $n-i-1$ times every time the innermost for loop is executed.



THE OUTERMOST LOOP: This is

```
for (i=0; i<n-1; i++) {
   minJ = i; minVal = a[i];
    — the innermost loop —
   a[minJ] = a[i]; a[i] = minVal;
}
```

The first iteration thus has $i = 0$, the last iteration has $i = n-2$, and i increases by 1 with each iteration. So, the number of times the innermost code is executed is:

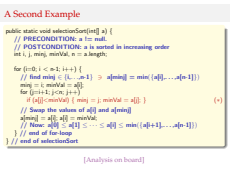
iteration #	value of i	# of executions $= (n-i-1)$
1	0	n-1
2	1	n-2
3	2	n-3
⋮	⋮	⋮
n-3	n-4	3
n-2	n-3	2
n-1	n-2	1

Therefore, the total # of executions is

$$1+2+3+\cdots+(n-1).$$

By a standard bit of math this last sum =

$$\frac{(n-1)\cdot n}{2} \quad = \quad \frac{1}{2}\cdot n^2 - \frac{1}{2}\cdot n.$$



Step 3: Take the count from step 2 and conclude the "order" of the runtime on any particular machine.
So, there are constants $c_\ell, c_u > 0$ such that, for any array of length n,

$$c_\ell \cdot (\frac{1}{2}n^2 - \frac{1}{2}n) \text{ picosecs}$$
$$\leq \text{ the runtime of } \texttt{selectionSort} \leq c_u \cdot (\frac{1}{2}n^2 - \frac{1}{2}n) \text{ picosecs.}$$

For a slightly different choice of positive constants c'_ℓ and c'_u, we have

$$c'_\ell \cdot n^2 \text{ picosecs}$$
$$\leq \text{ the runtime of } \texttt{selectionSort} \leq c'_u \cdot n^2 \text{ picosecs}$$

for sufficiently large n.

Big-Θ

Convention

- $\mathbb{N} = \{0, 1, 2, \dots\}$.
- $\mathbb{N}^+ = \{1, 2, 3, \dots\}$.
- $f, g: \mathbb{N} \to \mathbb{N}^+$.
- The collection of functions that have *linear growth rate* is the set:

$$\Theta(n) \quad = \quad \left\{ f \quad : \quad \begin{array}{c} \text{for some } c_\ell,\, c_u > 0, \\ \text{for all sufficiently large } n \\ c_\ell \leq f(n)/n \leq c_u \end{array} \right\}.$$

- The collection of functions that have *quadratic growth rate* is the set:

$$\Theta(n^2) \quad = \quad \left\{ f \quad : \quad \begin{array}{c} \text{for some } c_\ell,\, c_u > 0, \\ \text{for all sufficiently large } n \\ c_\ell \leq f(n)/n^2 \leq c_u \end{array} \right\}.$$

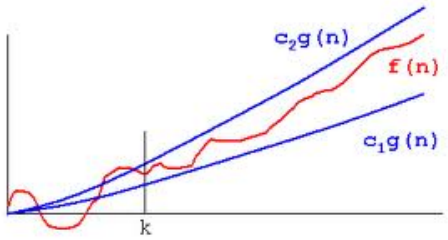
Big-Θ, Continued

The general case

Suppose g:$\mathbb{N} \to \mathbb{N}^+$ is given.

- The collection of functions that have *growth rate proportional* g is:

$$\Theta(g(n)) \quad = \quad \left\{ f \quad : \quad \begin{array}{c} \text{for some } c_\ell,\, c_u > 0, \\ \text{for all sufficiently large } n \\ c_\ell \leq \dfrac{f(n)}{g(n)} \leq c_u \end{array} \right\}.$$



$c_2 = c_u$ and $c_1 = c_\ell$.

Image from: http://xlinux.nist.gov/dads/HTML/theta.html

Some Examples

Example

Suppose f is given by:

$$f(n) = 100n^2 + 8000n + 97.$$

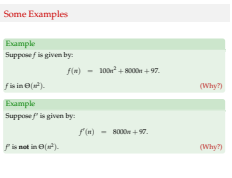
f is in $\Theta(n^2)$. (Why?)

Example

Suppose f' is given by:

$$f'(n) = 8000n + 97.$$

f' is **not** in $\Theta(n^2)$. (Why?)



1. $100 \le f(n)/n^2 \le 200$ for all $n > 81$.
2. $\dfrac{8000n + 97}{n^2} \to 0$

The Limit Rule for Big-Θ

The Limit Rule (Version 1)

Suppose that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

where $0 \le c \le +\infty$.
(a) If $0 < c < +\infty$, then f is in $\Theta(g(n))$.
(b) If $c = 0$ or $c = \infty$, then f is **not** in $\Theta(g(n))$.

Example

Suppose f and f' are given by:

$$f(n) = 100n^2 + 8000n + 97. \qquad f'(n) = 8000n + 97.$$

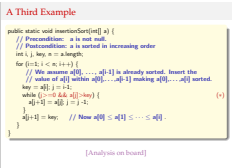
f is in $\Theta(n^2)$ and f' is **not** in $\Theta(n^2)$. (Why?)

A Third Example

```
public static void insertionSort(int[] a) {
   // Precondition:  a is not null.
   // Postcondition: a is sorted in increasing order
   int i, j, key, n = a.length;

   for (i=1; i < n; i++) {
      // We assume a[0], ..., a[i-1] is already sorted. Insert the
      // value of a[i] within a[0],...,a[i-1] making a[0],...,a[i] sorted.
     key = a[i]; j = i-1;
     while (j>=0 && a[j]>key) {                                    (*)
       a[j+1] = a[j]; j = j -1;
     }
     a[j+1] = key;      // Now a[0] ≤ a[1] ≤ ··· ≤ a[i] .
   }
}
```

[Analysis on board]



InsertionSort is much faster on some length-n inputs than others.
E.g.,

1. If initially $a[0] < a[1] < \cdots < a[n-1]$,
   InsertionSort runs in $\Theta(n)$ time.
2. If initially: $a[0] > a[1] > \cdots > a[n-1]$,
   InsertionSort runs in $\Theta(n^2)$ time.

The *best case run time* on a size-n input is the smallest run time of the algorithm on such inputs.
The *worst case run time* on a size-n input is the biggest run time of the algorithm on such inputs. For example,

- `findMin` has $\Theta(n)$ best and worse case run times.
- `selectionSort` has $\Theta(n^2)$ best and worse case run times.
- `insertionSort` has $\Theta(n)$ best case and $\Theta(n^2)$ worse case.
  Using Θ here is not quite right!

Big-O and Big-Ω

Recall:

$$\Theta(g(n)) = \left\{ f \;:\; \begin{array}{c} \text{for some } c_\ell, c_u > 0, \\ \text{for all sufficiently large } n \\ c_\ell \le f(n)/g(n) \le c_u \end{array} \right\}.$$

To talk about *upper bounds* and *lower bounds* growth rates, we break Θ in two parts.

$$O(g(n)) = \left\{ f \;:\; \begin{array}{c} \text{for some } c_u > 0, \\ \text{for all sufficiently large } n \\ f(n)/g(n) \le c_u \end{array} \right\}.$$

$$\Omega(g(n)) = \left\{ f \;:\; \begin{array}{c} \text{for some } c_\ell > 0, \\ \text{for all sufficiently large } n \\ c_\ell \le f(n)/g(n) \end{array} \right\}.$$

Big-O and Big-Ω, continued

Intuitively:

- $\Theta(g(n))$ is the collection of functions that have growth rates proportional to $g(n)$.
- $O(g(n))$ is the collection of functions $f(n)$ such that $f(n)/g(n)$ is no worse than some constant for large n.
- $\Omega(g(n))$ is the collection of functions $f(n)$ such that $f(n)/g(n)$ is no smaller than some positive constant for large n.

Example (Insertion sort)

- Its run time is $\Omega(n)$ and $O(n^2)$.
- Its worst-case run-time: $O(n^2)$
- Its best-case run-time: $\Theta(n)$.

The Limit Rule, Again

The Limit Rule (Version 2)

Suppose that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

where c is such that $0 \le c \le +\infty$.

(a) If $0 < c < +\infty$, then f is in $\Theta(g(n))$, $O(g(n))$, and $\Omega(g(n))$.

(b) If $c = 0$, then f is in $O(g(n))$, but **not** in either $\Theta(g(n))$ or $\Omega(g(n))$.

(c) If $c = \infty$, then f is in $\Omega(g(n))$, but **not** in $\Theta(g(n))$ or $O(g(n))$.

Important!!

You cannot use the limit rule when $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ fails to exist.

Applying the Limit Rule

Example

(a) $n \mapsto (8000n + 97)$ is in $O(n^2)$.

(b) $n \mapsto (100n^2 + 8000n + 97)$ is in $\Omega(n)$.

Limitations of the Limit Rule

(a) $n \mapsto n^{2+\sin n}$ is in $\Omega(n)$ and $O(n^3)$ but is not in any $\Theta(n^k)$ class.

(b) $n \mapsto (2 + \sin n)n^2$ is in $\Theta(n^2)$, but
$\lim_{n\to\infty} \frac{(2+\sin n)n^2}{n^2} = \lim_{n\to\infty}(2 + \sin n)$, which fails to exist.