CS211
Asymptotic Complexity

David I. Schwartz

Overview

- code (number guessing example)
- approximate analysis to motivate math
- machine model
- analysis (space, time)
- machine architecture and time assumptions
- code analysis
- more assumptions
- Big-O notation
- examples
- extra material (theory, background, math)

1. Number Guessing Example

1.1 Code

```
import java.util.Scanner;

public class NumberGuess {
  public static void main(String[] args) {

    try {
      int guess;
      int count;
      int LOW=Integer.parseInt(args[0]);
      int HIGH=Integer.parseInt(args[1]);
      if (LOW > HIGH) throw new Exception();
      final int STOP=HIGH-LOW+1;
      int target=(int)(Math.random()*(HIGH-LOW+1))+(int)LOW;
      Scanner in = new Scanner(System.in);
      System.out.print("\nGuess an integer: ");
      guess = in.nextInt();
      count = 1;

      while ( guess != target && guess >= LOW &&
            guess <= HIGH   && count < STOP ) {

        if (guess < target) System.out.println("Too low!");
          else if (guess > target) System.out.println("Too high!");
        else System.exit(0);
        System.out.print("\nGuess an integer: ");
        guess = in.nextInt();
        count++ ;
      }
        if (target == guess)
          System.out.println("\nCongratulations!\n");
        else System.out.println("Out of bounds...quitting.");

    } catch(Exception e) {
      System.out.println("Entered wrong input...quitting.");
      System.exit(0);
    }

  } // main
} // NG
```

1.2 Solution Algorithms

- Random:
  - pick any number to get lucky!
  - worst-case time could be infinite
- Linear:
  - guess one number at a time
  - start from bottom and head to top
  - worst-case time could be size of range
- Binary:
  - start at middle and check guess
  - go up or down based on guess (numbers are “pre-sorted!”)
  - worst-case time?

1.3 Example: Guess between 1 and 100

- linear:
  - 100 possible guesses
- binary?
  - assume 100 is the target
  - assume always round down for integer division
  - pattern: 50→75→87→93→96→98→99→100
  - 8 guesses in worst case

1.4 More General

- Other examples:

Range	Linear	Binary	Selections
1	1	1	1→1
1–10	10	5	5→7→8→9→10
1–100	100	8	50→75→87→93→96→98→99→100
1–1000	1000	11	500→750→875→937→968→984→992→ 996→998→999→1000

- Analysis:
  Time for binary search grows very slowly in comparison to linear search algorithm for large ranges.
- Key ideas:
  - Algorithms can cause radically different efficiencies.
  - Efficiency based on time to analyze input data.
  - To measure, count ***comparisons*** (k), which are guesses in this problem.

1.5 Equations

- Express runtime T in terms of input range n.
- Linear:
  - runtime is proportional to n
  - Since k=n (number of guesses=input data range):
    $T(n) \ = \ n$

- Binary:
  - set up experiment for some cases of n
  - assume target is always last number for worst case:

data size (n)	guessing pattern	guess count (k)
1	1	1
2	1→2	2
4	2→3→4	3
8	4→6→7→8	4
16	8→12→14→15→16	5

  - pattern for n and k: $n = 2^{k-1}$
  - so, $k = \log_2 n + 1$
    rounding down: $T(n) = \log_2 n + 1$
    rounding up $T(n) = \lceil \log_2 n \rceil$
- Understanding these equations:
  - LHS: $T(n)$ is the effort you will expend
  - RHS: is k, the amount of guesses (the effort!)
- Contrast binary and linear approaches:
  - Binary: $\log_2 n$ (2 raised to k guesses yields n; so you need to pick a small k, as powers grow quickly)
  - Linear: n (perform n comparisons)
  - Compared to linear choosing, binary choosing has k growing less quickly as n increases.

2. Algorithm Analysis

2.1 Algorithm

- steps to solve a problem
- supposed to be independent of implementation
- before programming, should decide on best algorithm!
- how to rate algorithms?

2.2 How to analyze an algorithm?

- running time
- memory needed/used
- correctness of output/results
- clarity for a human
- robustness/generality

2.3 Exact analysis

- exact analysis is too detailed
  - hardware (CPU, memory,…)
  - software (OS, lang, compiler,...)
- focus on ***time*** and ***space*** for measurement
  - ***time***: how many steps required?
  - ***space***: how much memory needed?
- we tend to focus on time, since memory is cheap

3. Machine Model

3.1 Stack Machine

- memory for stack, static, registers, heap, program areas:
  - space: how much of the memory is needed
  - time: how quickly can the values be stored and retrieved in/from the memory
- JVM:
  - Java code is compiled to ***byte code*** and stored in program area
  - interpreter acts on each instruction one at a time
  - instructions store and retrieve values from memory

3.2 Space Analysis of Machine Model

- Run CS212's SaM to see the stack grow and shrink
- best and worst case: try not to run out of stack space!

4. Time Analysis of Machine Model

4.1 Actions that we will count as constant

- time required to fetch an operand
- time required to store a result
- time required to perform an arithmetic/logic operation
- time required to call a method
- time requited to return from a method
- time required to pass arg to method
- time required to calculate the address for array index
- time time to create an object (does not include fields)

4.2 Examples

- `y = y + 1;` $2fetch + op + store$
- `y = a[i];` $3fetch + array + store$

4.3 Average running time

- kind of complicated (need to look up)
- probability of getting certain inputs

4.4 Best case

- assume best possible ordering of data or input
- not too useful

4.5 Worst case

- assume worse possible ordering of data or input
- this is what we focus on!
- example) number guessing for [1, 100]
  - best case #: 1 guess!
  - average case #: maybe 4?
  - worst case #:
    8 for binary
    100 for linear
  - so, worst-case analysis helps to determine choice of binary search more useful!

5. Machine Architecture

5.1 Processor clock

- coordinates memory ops by generating time reference
- clock generates signal called ***clock cycle*** or ***tick***

5.2 Clock speed

- how fast instructions execute
- measured as ***clock frequency***: how many ticks per second (MHz or GHz)
- how many instructions executed per tick?
  - depends on CPU, instruction set, instruction type
  - one instruction takes one tick, maybe more
  - architectures: CISC (PCs), RISC (Macs, Suns)

5.3 Clock period

- T = 1/frequency (unit of time/tick)
- measured nano (or smaller) seconds
- e.g.) 2.4 GHz Gateway? E-6000 has
  - clock speed: 2.4×10^{-9} tick/s
  - clock period: 4.17×10^{-10} (s/tick) (or just 0.417 ns)

5.4 Algorithm time and clock period

- Time for instruction to happen is proportional to T
  - T for clock period
  - so, action = kT, where $k > 0$, k is integer
- Simplifications:
  - express actions in terms of T
  - let T=1 since we express all actions in terms of T
  - let each $k = 1$
  - so, we assume each operation takes the same amount of time (1 cycle or operation)
- Examples:
  - *ALU*: 1 op
  - *Assign*: 1 op
    store value on LHS; ops on RHS counted separately
  - *Loop*:
    ((loop iterations)*(# of ops/iteration)) ops
  - *Selection*:
    (worst-case of condition or any sub-statement) ops
  - *Method*:
    (number of ops inside a method) ops

5.5 Algorithm Examples

- Summation: $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$
- Three approaches to compute the sum:

```
public class test {

  public static int n = 5;

  public static void main(String[] args) {

    System.out.println("Test 1: "+test1());
    System.out.println("Test 2: "+test2());
    System.out.println("Test 3: "+test3());
  }

  public static int test1() {
    return n*(n+1)/2;
  }

  public static int test2() {
    int sum = 0;
    for (int i=1; i<= n; i++)
      sum += i;
    return sum;
  }

  public static int test3() {
    int sum = 0;
    for (int i=1; i<= n; i++)
      for (int j=1; j<=i; j++)
        sum += 1;
    return sum;
  }

}
```

- All output is identical.

5.6 Time Analysis of Examples

code (1)	time analysis
`sum = n*(n+1)/2;`	8
	total: 7

code (2)	time analysis
`sum = 0;`	2
`for(int i=1;`	2
`    i <= n;`	$3n$
`    i++)`	$4n$
`    sum = sum + i;`	$4n$
	total: $11n + 4$

code (3)	time analysis
`sum = 0;`	2
`for(int i=1;`	2
`    i <= n;`	$3n$
`    i++)`	$4n$
`    for(int j=1;`	$2n$
`        j <= i;`	3 * sum(i , i = 1..n)
`        j++)`	3 * sum(i , i= 1..n)
`        sum = sum + 1;`	4 * sum(i , i = 1..n)
	total: $5n^2 + 13n + 4$

5.7 Analysis Of The Analysis

- approaches:
  - $T_1(n) = 8$
  - $T_2(n) = 11n + 4$
  - $T_3(n) = 5n^2 + 13n + 8$
- as n increases, approach #1 runs a lot faster!

5.8 Intuitive Approach

- pick the dominant or most important operation
- count the dominant operation, which is usually a comparison inside a loop
- don't worry about lower order terms, since we worry about case of large n
- don't worry about constants in front of leading term

5.9 Recurrence Relations

- another way to set up algorithms in terms of T
- If searching 1 item, then time is $T(1)$.
- Else, if searching n sorted items, then time is $T(1)$ (check the middle item) + $T(n/2)$ to search half the data.
- So,
  $T(1) = 1$
  $T(n) = 1 + T(n/2), n > 1$
- Function defined in terms of itself: ***Recurrence Relation***

5.10Solving The Recurrence Relation

- Base case: $T(1) = 1$ is true (also works using $T(n)$)
- Guess: $T(n) = 1 + T(n/2), n > 1$
- Assume $T(k) = 1 + T(k/2), k < n$ (use strong induction)
- Note that $T(k) = 1 + T(k/2)$ (time to search is time for middle element + rest of data)
- Substitute hypothesis for $T(k/2)$: $T(k) = 1 + (1 + \log_2(k/2))$
- Some math: $T(k) = \log_2 4 + \log_2(k/2)$
- Things work out: $T(k) = 1 + \log_2 k$
- So, original assumption was correct.

6. Asymptotic Complexity

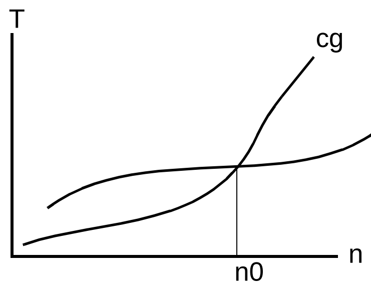
- Also growth notation, asymptotic notation, Big O, …
- The gist: describe algorithm efficiency in terms of the general, large case.

6.1 How to compare algorithms?

- find each $T(n)$ function
- problem:
  - they're not really that exact because of limitations in assumptions (e.g., are all processors created equally?)
  - we need worst-case scenarios!
- solution:
  - need a measure that is accurate in the extreme
  - one measure: ***asymptotic upper bound***

6.2 Big-O Notation

- Definition:
  $O(g(n)) = \{f(n) | (c, n_0) \text{ such that } 0 \le f(n) \le cg(n), n \ge n_0\}$
- $O(g(n))$ provides an asymptotic upper bound
- not the tightest upper bound, though
- $f(n) \in O(g(n))$ means that *f* is function that belongs to a set of bounding functions $O(g(n))$
- books usually say $f(n) = O(g(n))$



6.3 Witness Pair

- the idea is that for a large data set, an algorithm becomes dominated by the input
- *g* bounds *f* given a certain value of *c* for all *n* past a certain *n* called n_0
- need to find a value of *c* and n_0 so that $f(n) \le cg(n)$ is satisfied
- both *c* and n_0 must be greater or equal to zero

6.4 Why *asymptotic*?

- We focus on highest-order term, which is the asymptote that the runtime approaches
  - For example, 2*n*+10 takes more time than *n*+1, but they have the same asymptotic complexity!
  - For small *n*, difference is important
  - For big *n* (worst case), difference is negligible
- For worst-case, drop the constants and lower terms:

$$T_1(n) = 8 \to T_1(n) \in O(1)$$

$$T_2(n) = 11n + 4 \to T_2(n) \in O(n)$$

$$T_3(n) = 5n^2 + 13n + 8 \to T_3(n) \in O(n^2)$$

6.5 Why complexity?

- Because we talk about the amount of work that must be done (how many steps must be performed.) Usually this is in terms of number of comparisons or number of swaps, for a searching or sorting algorithm.
- Amount of work required = complexity of the problem.

6.6 Why complexity classes?

- ***Class***: classification of sets of bounding functions
- Why is this important?
  1. It allows us to compare two algorithms that solve the same problem and decide which one is more efficient.
  2. It allows us to estimate approximately how long it might take to run a program on a specific input.

  Example: If I have an $O(n^3)$ algorithm, and I give it an input with 300 items, it will take about 27,000,000 steps! If I know how many such steps my computer can do per second, I can figure out how long I'll have to wait for the program to finish.
- Note that it's important to know what is being measured. Number of comparisons? Swaps? Additions? Divides? (ouch! Divide is expensive!)

7. Complexity Classes

7.1 Limits

- Given two non-decreasing functions of positive integers f and g, denote

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

- $L(f, g)$ is a constant that's either 0, positive, or infinite.
- We are interested in knowing whether or not *f* grows faster or slower that *g*.
- The limit determines the eventual relationship as *n* increases.
- If the limit reaches a constant or zero, *g* is growing faster than *f* and thus gives an upper bound to *f*.

7.2 Varieties of complexity classes

- Note: See math review near end. For example, $\{x|y\}$ can be read as "the set of all x such that y holds."
- Given M = set of all positive monotonically increasing function of positive integers. A function $f(x)$ is monotonic increasing if $a < b$ implies $f(a) < f(b)$.
  - $o(g(n)) = \{f(n) \in M | L(f(n), g(n)) = 0\}$ (little o)
  - $O(g(n)) = \{f(n) \in M | 0 \le L(f(n), g(n)) < \infty\}$ (big O)
  - $\Theta(g(n)) = \{f(n) \in M | 0 < L(f(n), g(n)) < \infty\}$ (big Theta)
  - $\Omega(g(n)) = \{f(n) \in M | 0 < L(f(n), g(n)) \le \infty\}$ (big Omega)
  - $\omega(g(n)) = \{f(n) \in M | L(f(n), g(n)) = \infty\}$ (little omega)
- We focus on $O(g(n))$ (the set of functions $g(n)$ that bound $f(n)$ asymptotically.
- Think of $O(g(n))$ as set of all functions $f(n)$ that belong to M that grow faster. (The limit of $f(n)/g(n)$ becomes 0 or constant).)

8. Examples

8.1 Example 1

- Prove that $f_1(n) = \frac{11}{2}n^2 + \frac{47}{2}n + 22$ is $O(n^3)$.
- Let $f_1(n) = cn^3$ for $c > 0$ and $n \ge n_0 \ge 0$.
- Find witness pair (c, n_0).
- Let $c = 1$. So, $\frac{11}{2}n^2 + \frac{47}{2}n + 22 \le n^3$.
- Thus, $n_0 = 8.6$.
- Given a bounding function $cn^3 = n^3$, before $n = 8.6$, $f_1(n)$ exceeds n^3.
- But after n_0, n^3 is higher and thus an upper bound.
- Note: you could actually make tighter bounds is you said $f_1(n) = O(n^2)$.
- In general, for polynomials, with highest term nm, $f(n) = O(mn)$.

8.2 Example 2

- Problem: Given $f_1(n) = O(g(n))$, $f_2(n) = O(g(n))$, and $h(n) = f_1(n) + f_2(n)$, is $h(n) = O(g(n))$? Justify formally using witness pairs (c, n).
- Answer: Yes
- Proof: Using givens and witness pairs (c_1, n_1) and (c_2, n_2):
  - $f_1(n) \le c_1 g(n), n > n_1$
  - $f_2(n) \le c_2 g(n), n > n_2$
- Use $c_s = c_1 + c_2$ and $n_s = \max(n_1, n_2)$ for witness pair (c_s, n_s).
- Add $f_1(n)$ and $f_2(n)$:
  - $f_1(n) + f_2(n) \le (c_1 + c_2)(g(n))$
  - $f_1(n) + f_2(n) \le (c_s)(g(n))$
- The above relation is true for all $n > n_s$.
- So, there is a value of n_0 and c such $h(n) = O(g(n))$ holds.

8.3 Example 3

- Problem: Why do not specify logarithm bases for big O analysis? Show that $\log_b n = O(\log_2 n)$.
- Let $L = \log_b n$ and $B = \log_2 b$.
- So, $n = b^L$ and $b = 2^B$.
- So, $n = b^L = (2^B)^L = 2^{BL}$.
- So, $\log_2 n = BL = B\log_b n$.
- So, $\log_b n = \left(\frac{1}{B}\right)\log_2 n$ for $n \ge 1$.
- Choose witness pair $c = 1/B$ and $n_0 = 1$.
- So, $\log_b n \le c\log_2 n$.
- Thus, $\log_b n = O(\log_2 n)$.

9. Math Review

9.1 Powers

- $x^a x^b = x^{(a+b)}$
- $x^a / x^b = x^{(a-b)}$
- $(x^a)^b = x^{ab}$
- $(xy)^a = x^a y^a$

9.2 Logarithms

- Definition:
  - Let $b^p = v$ and $b \neq 1$ and $b \neq 0$.
  - Then, $p = \log_b v$, where p is the logarithm of v to base b.
- Terms:
  - common log: base 10 (social scientists) (log[10])
  - natural log: base e (engineers) (ln)
  - binary log: base 2 (computer scientists) (log or lg)

- Laws:
  - $\log_b(b^y) = y$
  - $b^{\log_b x} = x$
  - $\log_b(uv) = \log_b(u) + \log_b(v)$
  - $\log_b(u/v) = \log_b(u) - \log_b(v)$
  - $\log_b(u^v) = v\log_b(u)$
  - $\log_b(x) = \dfrac{\log_c(x)}{\log_c(b)} = (\log_b(x))(\log_c(x))$
- Integral Binary Logarithm
  - floor($\log_2 n$) for integer n
  - number of times n can be divided by 2 before reaching 1

9.3 Sets

- collection of unique items
  - example) $S = \{a, c, b, 1\}$
- membership:
  - example) $1 \in S$ (1 is in S)
- special notation: $\{x|y\}$
  - a set of items x such that property y holds
  - example) $S = \{x | 1 \leq x \leq 4, x \in integers\}$
    (S is the set of integers between 1 and 4, inclusive)