Algorithm Analysis & Intro to Data Structures

John Rodriguez

Algorithm Analysis

Model

- assume infinite memory; i.e., runtime isn't affected by limited memory
- ignore disk latency, file & reads, etc.
- Simple instructions = 1 time unit
- for loops = running time of statements * # of iterations
- *unitless* time - tough to be accurate since different hardware will result in different measurements

A Simple Example

// computes $0^2 + 1^2 + 2^2 + ... + n^2$

```
public static int sum(int n) {
  int sum = 0;                  1

                  1        N+1      N        = 2N + 2
  for(int i = 0; i <= n; i++) {
    sum += i * i * i;
  }                                 4 N

  return sum;                  1
}
```

Total running time = 2 + 4N + 2N + 2

$= 6N + 4$

Running Time

- Measuring running time itself is great, but…
- …the *rate* or *order* of growth is more important
- We consider an algorithm to be more efficient than another if its worst-case running time has a lower rate of growth

Functions

constant - f(1)	whose license plate is...
linear - f(N)	counting all the students in a class
logarithmic - f(log N)	20 questions game
exponential - $f(a^N)$	cells splitting biologically, doubling each time
quadratic - $f(N^2)$	finding all pairs of students
cubic - $f(N^3)$	finding all triples OR matrix multiplication
factorial - f(N!)	all permutations of N elements
loglinear - f(N log N)	most sorting algorithms

Problem

$f(1)$	$f(\log N)$	$f(N^2)$	$f(N!)$
$f(N)$	$f(a^N)$	$f(N^3)$	$f(N \log N)$

Using the previous descriptions, take a guess and rank the functions above from smallest to largest order of growth

Visualizing

f(1)	f(log N)	f(N^2)	f(N!)
f(N)	f(a^N)	f(N^3)	f(N log N)

Now using https://www.desmos.com/calculator, rank the functions from smallest to largest order of growth

f(1) , f(log N) , f(N) , f(N log N) , f(N^2) , f(N^3) , f(a^N) , f(N!)

Time Comparisons

Running time for algorithm

f(n)	n=256	n=1024	n=1,048,576
1	1µsec	1µsec	1µsec
$\log_2 n$	8µsec	10µsec	20µsec
n	256µsec	1.02ms	1.05sec
$n \log_2 n$	2.05ms	10.2ms	21sec
n^2	65.5ms	1.05sec	1.8wks
n^3	16.8sec	17.9min	36,559yrs
2^n	3.7×10^{63}yrs	5.7×10^{294}yrs	2.1×10^{315639}yrs

Input Comparisons

Largest problem that can be solved if Time <= T at
1μsec per step

f(n)	T=1min	T=1hr	T=1wk	T=1yr
n	6×10^{7}	3.6×10^{9}	6×10^{11}	3.2×10^{13}
nlogn	2.8×10^{6}	1.3×10^{8}	1.8×10^{10}	8×10^{11}
n^2	7.8×10^{3}	6×10^{4}	7.8×10^{5}	5.6×10^{6}
n^3	3.9×10^{2}	1.5×10^{3}	8.5×10^{3}	3.2×10^{4}
2^n	25	31	39	44

Refining Our Model

- The extra precision is not worth the effort
- Multiplicative constants can be ignored

  $5N \longrightarrow N$
- Lower-order terms are dominated by higher-order terms; ignore

  $N^2 + N \longrightarrow N^2$

Take 2

```
// computes 0^2 + 1^2 + 2^2 + ... + n^2

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

  return sum;          1
}
```

Total running time = 2N + 2

$= O(\text{N})$

Asymptotic Efficiency

- Big-O notation describes an “upper bound”
- “*By how much does the running time of this algorithm increase as the size of the input increases without bound?*”
- Big-Ω notation describes an “lower bound”
- Big-Θ notation describes a “tight bound”

Take 3

```
// computes 0² + 1² + 2² + ... + n²

public static int sum(int n) {
  int sum = 0;


  for(int i = 0; i <= n; i++) {
    sum += i * i * i;
  }

  return sum;
}
```

Total running time = $O(N) = O(N^2) = O(N^3)$

$= \Theta(N)$

Another Example

```
int k = 0;

for(int i = 0; i < n; i++) {

  for(int j = 0; j < n; j++) {
    k++;
  }

}
```

1

N

1

N

Total running time = $O(N^2+1)$

$= O(N^2)$

Recursive Example

```
public static long factorial(int n) {

  if(n <= 1)
    return 1;          ← 1

  else
    return n * factorial(n - 1);

}
```

1 ?

$$T(N) = T(N\text{-}1) + 2$$

Total running time = $O(N)$

Fibonacci

```java
public static long fib(int n) {
  if(n <= 1)
    return 1;

  else
    return fib(n - 1) + fib(n - 2);
}
```

$T(N) = T(N\text{-}1) + T(N\text{-}2) + 2$

Total running time = $O(1.5^N)$

Search

```
// return index of key in a if present; -1 otherwise
public static int search(int[] a, int key) {
  for (int i = 0; i < a.length; i++) {
    if (a[i] == key) {
      return i;
    }
  }
  return -1;
}
```

Total running time = $O(N)$

Binary Search

```
// return index of key in a if present; -1 otherwise
// a must be sorted
public static int binarySearch(int[] a, int key) {
  int lo = 0;
  int hi = a.length - 1;

  while (lo <= hi) {
    int mid = lo + (hi - lo) / 2;

    if (key < a[mid])
      hi = mid - 1;
    else if (key > a[mid])
      lo = mid + 1;
    else
      return mid;
  }
  return -1;
}
```

Total running time = $O(\log N)$

Evaluating a^b

```
public static long pow(long a, long b) {
  long pow = 1;

  for(int i = 1; i <= b; i++) {
    pow *= a;
  }

  return pow;
}
```

Total running time = $O(N)$

Evaluating a^b

```
public static long pow2(long a, long b) {
  if(b == 0) return 1;

  if(b == 1) return a;

  if(b % 2 == 0) {
    return pow(a * a, b / 2);
  }
  else {
    return pow(a * a, b / 2) * a;
  }
}
```

Total running time = $O(\log N)$

Homework

- Show that X^{62} can be computed with only 8 multiplications

- A majority element in an array, A, of size N is an element that appears more than N / 2 times. For example,

  3,3,4,2,4,4,2,4,4

  has a majority element (4), whereas

  3,3,4,2,4,4,2,4

  does not. Write a program to solve and determine its runtime.

Data Structures

Arrays

- fixed length
- elements stored contiguously in memory (fragmentation)
- easy to iterate - sequential or random access

Lists

- variable length
- elements are not stored contiguously in memory
- can be slower to access, depending on implementation

Sets

- Like a List, but contains no duplicate elements
- Based on the mathematic definition of a set
- Review Set Theory from the Discrete Math slides

Problem

- Construct a class ListSet that extends the ArrayList class and implements the Set interface.
- Override ONLY
  - add(E e)
  - removeAll(Collection)
  - retainAll(Collection)
- Test using Strings
- What’s the runtime of the 3 methods?

Stacks

- visualize as a stack of papers or pancakes
- what you put on top (push) is the first item you take off (pop)
- LIFO = last in, first out
- Examples:
  - “undo” feature, back button

Stack Operations

- push(o) inserts o at top of stack - O(1)
- pop() remove top of stack - O(1)
- size()
- isEmpty()
- top() return top, without removing - O(1)

Problem

- Construct a class ListStack that extends the ArrayList class. Add methods push(), pop() and top().

Applications of Stacks

- Method Calls
- Postfix Expressions

Queues

- visualize as a waiting line
- you add to the back of the line (enqueue), the front of the line is the first item you take off (dequeue)
- FIFO = first in, first out
- Examples:
    - messaging queues, routers, online ticketing

Queue Operations

- enqueue(o) inserts o at rear of queue - O(1)
- dequeue() remove from front of queue - O(1)
- size()
- isEmpty()
- front() return front, without removing - O(1)

Problem

- Construct a class ListQueue that extends the ArrayList class. Add methods enqueue(), dequeue() and front().

Applications of Queues

- Server HTTP Request Handling
- CPU Scheduling
- Printing Queues

References

- http://bigocheatsheet.com/
- https://www.desmos.com/calculator
- https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/Stacks%20and%20Queues.html

Homework

- Given a string made from the characters {}()[], write a program that returns true when balanced and returns false when unbalanced, e.g.,

([{ }]) → true

{ [}] →
false

(([ ]))) → false

() [{ }] → true

Exit Ticket

- See Slack channel