Data Structures in Java

Notes for 2014-07-08

Monica Quaintance

mjq2102@columbia.edu

Orientation

Office Hours for TAs:

Mon 7pm-9pm (Matthew)

Wed 6pm-8pm (Susanna)

Fri 6pm-8pm (James)

Class info is at http://www.cs.columbia.edu/~pblaer/cs3134/

Regarding Java:

Dev environment — Eclipse?

For homework, test environment specific for class

Virtual box of ubuntu based linux

Java Version 1.6

Data Structures

Organizing data in a computer for efficient later use

Some goals: analyzing algorithms and their costs, easy access, etc

Basic Structures:

- Arrays
- Primitives (int, char, double)
- Classes and their methods
- Packages
- Stacks
- Linked List
- Hash

Read the math from chapter 1 and info on generics

Arrays -> stored all next to each other in contiguous memory addresses

therefore, jump to next element in constant time

(1 millionth & 5th are same)

offset from first memory address

adding something in the middle — n time, and you may not even have enough space for it anyway

Proofs

Proof by Induction

eg fibonacci numbers: F(k+1) = F(k) + F(k-1)

F(11) = 144

show that [ith fibonacci number] F(i) < (5/3)^i power, for i>=1

1) prove base case

   works for i = 1:

   1 < 5/3

2) Assume inductive hypothesis is true

   Assume for all elements up to k, F(k) < (5/3)^k

3) Use assumption in 2 to prove true for k+1

   Prove F(k+1) < (5/3)^(k+1)

   F(k+1) = F(k) + F(k-1)

   F(k+1) < (5/3)^k + (5/3)^(k-1)

   F(k+1) < (3/5)(5/3)^(k+1) + (3/5)^2*(5/3)^(k+1)

   F(k+1) < (15/25)(5/3)^(k+1) + (9/5)*(5/3)^(k+1)

   F(k+1) < (24/25) * (5/3)^(k+1) < (5/3)^(k+1)

Proof by Counterexample

F(k) <= k^2

F(11) = 144

11^2 = 121

Therefore, not true

Proof by Contradiction

P(k) is the largest prime

Product of all primes, +1, must be bigger than P(k), so contradictory

Recursion

Function: 2^n

static method inside some class, only on integers, and only n >=0

```
static int twoN(int n){
    if (n ==0){
        return 1;
    }


    return 2*twoN(n-1);
}
```

Tail recursion — single recursive call where you iterate a number,

trivial to rewrite this as a loop

Function: fibonacci recursion

```
static int fib(int, n) {
    if(n==0){
        return 1;
    }
    else if(n==1){
        return 1;
    }
    return fib(n-1) + fib(n-2)
}
```

Instead, write them with a loop with two values that slide along

Analysis of Algorithms

Differences in:

- Platforms
- Best / worse case scenario
- Data set

Can affect how an algorithm runs

How can we judge an algorithm without these factors?

Count the "atomic unit" of operations, and assume all operations take the same time

–How many operations are being performed, as compared to the size of the inputs (n inputs)

7n vs 6n are basically the same as n approaches infinity, so constant factors are excluded

e.g.: Two nested loops + a loop: n^2 + n as n approaches infinity, algorithm is n^2 (highest order term)

T(n) - actual time factor of the algorithm

Big O: T(n) = O(f(n))

if there exists positive constant C & n(o) such that T(n) <= C*f(n) for n >= n(o)

f(n) -> heuristic in terms of n

Big Omega:

T(n) = OMEGA(g(n)); T(n) >= c*g(n)

Big Theta:

T(n) = THETA(h(n)); iff T(n) <= c*f(n)* & *T(n)* >= *c*g(n)

Little o

T(o) strictly less than c*function(n)

Growth rates:

- C
- log(N)
- log^2(N)
- N
- NlogN

- N^2
- N^3
- 2^n
- n!
- n^n

T1(n) = O(f(n))

T2(n) = O(g(n))

T1(n) + T2(n) = O(f(n)+g(n))

= O(max(f(n),g(n)))

T1(n)*T2(n) = O(f(n)*g(n))

–eg nested loops

If T(n) is a polynomial of order k then T(n) = THETA(n^k)

log^k(n) = O(n), regardless of the size of k (log is always better than order N)

If/elses – pick the bigger case, if or else, for counting O

Searching Algorithms

Looking for the index of a value in an array.

If value not in an array, return -1

Linear / Sequential Search:

```
static int ls(int[] a, int k){
    for(int i=1; i < a.length, i++){
        if (a[i]==k)
            return i;
    }
    return -1;
}
```

Worst Case: last element in array, or not in array

O(n)

Best Case: first element in array

O(1)

Average Case: O(n)

also, think about likelihood of thing not even being in the array

Binary Search: Assumes array is sorted

Best Case: O(1)

Look at midpoint

If not midpoint, then pick the half of the array the value is in, then start over

If list even, to find midpoint, round down (because dividing ints truncates the number)

EG: Find 10

[1,3,5,7,9]

-> 5

[5,7,9]

-> 7

[9]

10 not there, so return -1

Assume: Array of length n is a power of 2

Therefore, n = 2^x

After first comp, 2^(x-1), Then $2^{(x-2)\cdots 2}0$

Worst case cost is O(x), so cost is O(log(n))

Divide and conquer algorithms are log

Know algorithm is done when start - stop returns a negative number

```
static int bs(int[] a, int k){
    int start = 0;
    int stop = a.length -1;

    while (start <= stop){
        int mid = (start + stop)/2;

        if(k == a[mid])
            return mid;
        else if(k < a[mid]){
            stop = mid - 1;
        }
        else
            start = mid + 1;

    }
    return -1
}
```

rewrite this recursively, and keep track of starts and stops

with 4 inputs: a, k, start, stop

public fuction that calls a private recursive function