CS206

Finish Hashtables
Analysis of Algorithms

Error from last class

```
private int h(Object k) {
  return k.hashCode() % backingArray.length;
}
public void put(Object key, Object value) {
        backingArray[h(key)] = value;
    }
```

- Java hashcode can return negative values
  - So need to add absolute value

```
private int h(Object k) {
  return Math.abs(k.hashCode()) % backingArray.length;
}
```

Probing Handling Deletions

Suppose:

Tablesize=11

h(t) = t%11

quadratic probing

put(2,A)

put(13,B)

put(24,C)

put(35,D)

get(35)

Loca	Key	Valu
0	35	D
1		
2	1	A
3	13	B
4		
5		
6	24	C
7		
8		
9		
10		

del(13)

get(24)

put(35,E)

get(24)

put(46,F)

Locati	Key	Value
0	35	D
1		
2	1	A
3		
4		
5		
6	24	C
7		
8		
9		
10		

Tombstones!

Probing vs Chaining

- Probing is significantly faster in practice
- locality of references – much faster to access a series of elements in an array than to follow the same number of pointers in a list
- Efficient probing requires soft/lazy deletions – tombstoning
  - de-tombstoning

Rehashing

- Need to make the hashtable bigger when it gets "full"
- Need to remove tombstones when there are too many

```java
public class ProbeHT<K, V> {
private class Pair<L, W> { … }
private Pair<K, V>[] backingArray;
private void rehash(int newSize) {
 // ignore tombstone problem
}
```

Using Hashtables

- No worries about hashing functions, rehashing, …
  - Someone else’s responsibility
- java.util.HashMap

Private vars and inheritance

Problem, how to use private vars of parent class

Inherit.java

Exceptions, return values and toString

Main.java, Shak.java and Line.java

Running Time

- The run time of a program depends on
  - efficiency of the algorithm/implementation
  - size of input
  - what else?
- The running time typically grows with input size
- How do you measure running time?
  - CPU usage?
    - Reliability?

Timing Code

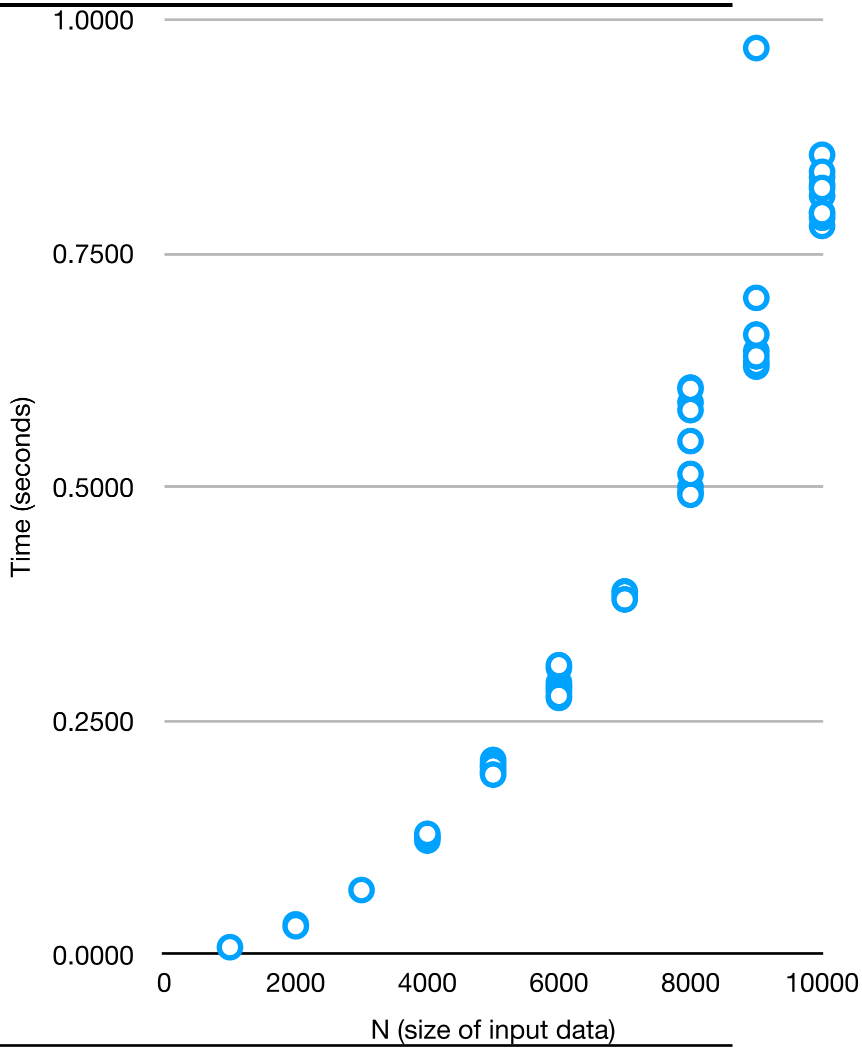
```
public class Timer {
    private static final int REPS = 10; // number of trials
    private static final int NANOS_SEC = 1000000000; // nanosec per sec

    public double doSomething(int[] data) {
        double k = 0;
        for (long i = 0; i < data.length; i++) {
            for (long j = 0; j < data.length; j++) {
                k += Math.sqrt(i * j);
            }
        }
        return k;
    }

    public static void main(String[] args) {
        Timer timer = new Timer();
        long data[] = new long[REPS];
        for (int j = 1000; j < 10001; j = j + 1000) {
            for (int i = 0; i < REPS; i++) {
                long start = System.nanoTime();
                timer.doSomething(new int[j]);
                long finish = System.nanoTime();
                data[i] = (finish - start);
                System.out.println(String.format("%d %.4f", j, (double) (finish - start) /
NANOS_SEC));
              }}}}
```

Experimental Studies

- Write a program implementing the algorithm
- Run it with different input sizes and compositions
- Record times and plot results



Limitation of Experiments

- You have to implement the algorithm
- You have to generate inputs that represent all cases
- Comparing two algorithms requires exact same hardware and software environments
  - Even then timing is hard
    - multiprocessing
    - file i/o

Theoretical Analysis

- Use a high-level description of algorithm
  - pseudo-code
- Running time as a function input size, n
- Ignore other details of the input
- Independent of the hardware/software environment

Primitive Operations

- Basic computations
  - * / + -
- Comparisons
  - ==, >, <
- Setting
  - x=y
- Assumed to take constant time
  - exact constant is not important
  - Because constant is not important, can do more than just this list

Example
Time required to compute an average

```
public double calcA(long[] data)
{
     double res = 0;
     for (int i=0; i<data.length; i++)
     {
          res = res+data[i];
     }
     return res/data.length;
}

public static calcB(long[] data) {
     double res = 0;
     long pd = 0;
     for (int i=0; i<data.length; i++) {
          long datum=data[i];
          if (pd<datum) {
           res = res+datum;
          }
          pd=datum;
     }
     return res/data.length;
}
```

How many operations? (In terms of the length of data)

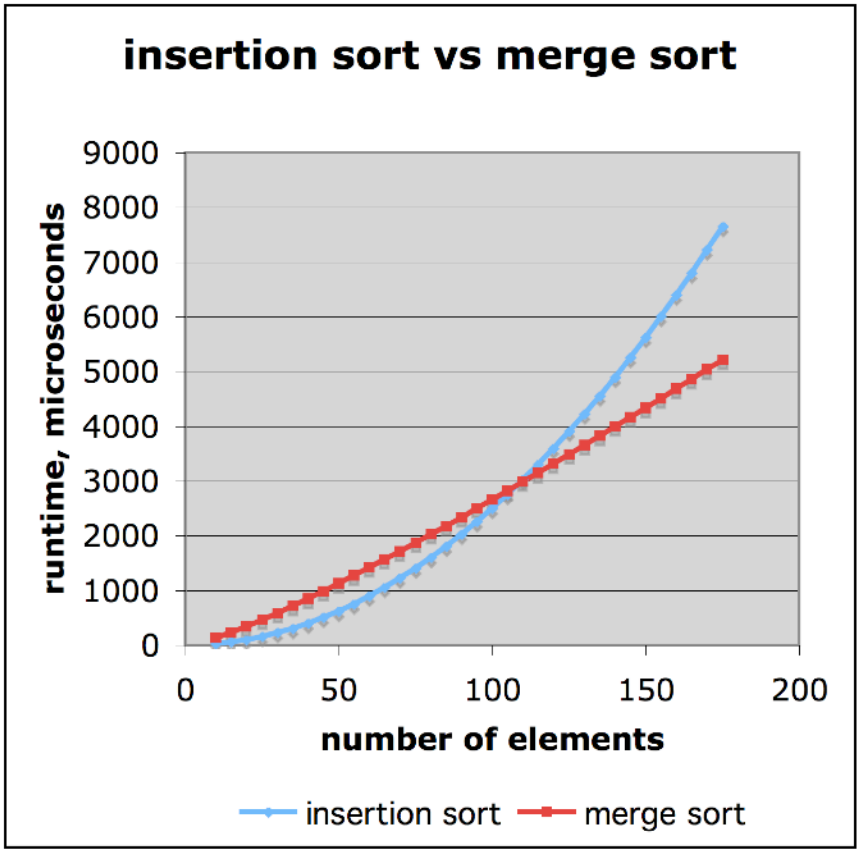
Estimate Running Time

- `calcB` executes a total of 7N+1 primitive operations in the worst case, 5N+1 in the best case.
- Let a be the fastest primitive operation time, b be the slowest primitive operation time
- Let $\mathrm{T}(n)$ denote the worst-case time of `calcB`. Then $\mathrm{a}\left(5n+1\right) \le T(n) \le b(7n+1)$
- $\mathrm{T}(n)$ is bounded by two functions
  - both are linear in terms of n

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm `calcB` (and `calcA`)

Comparison of Two Algorithms



- insertion sort: $n^2/4$
- merge sort: $2n\lg n$
- suppose n=10^8
  - insertion sort: $10^8 * 10^8/4 = 2.5*10^{15}$
  - merge sort: $10^8 * 26 * 2 = 5.2 * 10^9$
  - or merge sort can be expected to be about 10^6 times faster
  - so if merge sort takes 10 seconds then insertion sort takes about 100 days

Asymptotic Notation

- Provides a way to simplify analysis
- Allows us to ignore less important elements
  - constant factors
- Focus on the largest growth of n
  - Focus on the dominant term

How do these functions grow?

- $f_1(x) = 43n^2 \log^4 n + 12n^3 \log n + 52n \log n$
- $f_2(x) = 15n^2 + 7n \log^3 n$
- $f_3(x) = 3n + 4 \log_5 n + 91n^2$
- $f_4(x) = 13 \cdot 3^{2n+9} + 4n^9$

Big O

- Constant factors are ignored
- Upper bound on time
- Goal is to have an easily understood summary of algorithm speed
  - not implementation speed

Sublinear Algorithms

- O(1)
  - runtime does not depend on input
- $O(\lg_2 n)$
  - algorithm constantly halves input

Linear Time Algorithms: $O(n)$

- The algorithm's running time is at most a constant factor times the input size
- Process the input in a single pass spending constant time on each item
  - max, min, sum, average, linear search
- Any single loop

$O(n \log n)$ time

Frequent running time in cases when algorithms involve:

- Sorting
  - only the “good” algorithms
    - e.g. quicksort, merge sort, …

Quadratic Time: $O(n^2)$

- Nested loops, double loops
  - The `doSomething` algorithm
- Processing all pairs of elements
- The less-good sorting algorithms
  - e.g., insertion sort

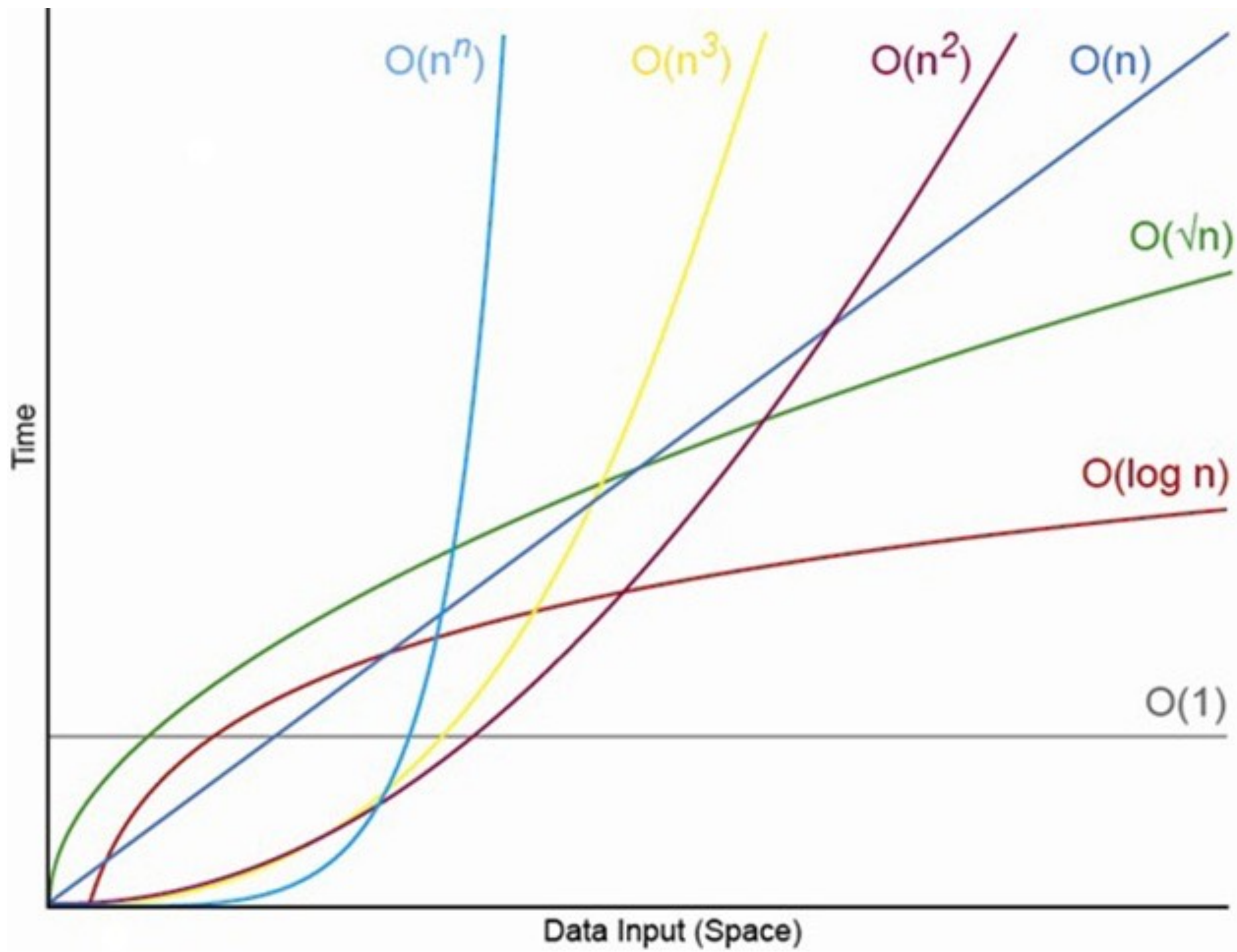
Slow!!!! Times

- polynomial time: $O(n^k)$
  - All subsets of n elements of size k

- exponential time: $O\left(2^n\right)$
  - All subsets of n elements (power set)

- factorial time: $O(n!)$
  - All permutations of n elements

Timing



Writing code that runs in O(x) time

```
public interface SpeedyAlgorithms {
    void orderOne(int[] data);
    void orderLogN(int[] data);
    void orderN(int[] data);
    void orderNSquared(int[] data);
    void orderNCubed(int[] data);
    void orderNFactorial(int[] data);
}
```

chalkboard