# 202: Computer Science II

Northern Virginia Community College

# Hashing and ”Linear” Sorting

Cody Narber, M.S.
Dec 09, 2017

# Problem: Searching

We have seen searching for items takes

- $O(n)$ time with unsorted data using **sequential search**
- $O(log_2 n)$ time with sorted data using **binary search**

But imagine we want to access something faster, maybe if we had someway of directly accessing the data.
**We could search in $O(1)$ time.**

**Hashing:** is a technique that orders and accesses elements iteratively in constant time, by using a **hash code** as a index/key. This hash code is computed via a **hash function**.

# Hash Function

**Hash function:** is a function/method that generates a numeric key to reference an object. In Java, the `Object` class has a `hashCode()` method built-in that uses the object's memory address. This is what is print out by default `toString()` method. We can override this behavior to produce any hash code we want:

```
1  public class Letter{
2       private char let;
3       public Letter(char l){
4            let = l;
5       }
6       @Override
7       public int hashCode(){
8            return Character.toUpperCase(let)-65;
9       }
10 }
```

# Hash Function

We can verify the functionality of our hash functions by creating objects and outputting them:

```
1  public class Circle{
2       private int rad;
3       public Letter(int r){
4            rad = r;
5       }
6       @Override
7       public int hashCode(){
8            return rad*rad;
9       }
10
11      public static void main(String[] args){
12           Circle c1 = new Circle(3);
13           Circle c2 = new Circle(5);
14           System.out.println(c1);     //Circle@9
15           System.out.println(c2);     //Circle@19 - Why 19 ??????
16      }
17 }
```

# Hash Table

**Hash table:** is essentially an array where the hash code is related to the object. Therefore we can access our array very simply by calculating the hash code and looking up the object.

```
1  public class SimpleHashTable{
2       private Object[] storage;
3       public SimpleHashTable(int size){
4            storage = new Object[size];
5       }
6       public void add(Object o){
7            storage[o.hashCode()] = o;
8       }
9       public void remove(Object o){
10           storage[o.hashCode()] = null;
11      }
12      public boolean contains(Object o){
13           return storage[o.hashCode()].equals(o);
14      }
15 }
```

# Hash Table - Problems?

In certain circumstances our simple example works fantastic! In constant running time. But what if we wanted to add two things with the same hash code or outside the array! ERROR!

```
1  public class TestSimpleHashTable{
2        public static void main(String [] args){
3              SimpleHashTable shm1 = new SimpleHashMap(26);
4              SimpleHashTable shm2 = new SimpleHashMap(10);
5              shm1.add(new Letter('D'));   //works adding into 3
6              shm1.add(new Letter('d'));
7                    //ERROR: replaces above object!
8              shm2.add(new Circle(2));   //works adding into 4
9              shm2.add(new Circle(4));
10                   //ERROR: Array-Index Out of Bounds
11       }
12 }
```

# Hash Table - Bounds

There is a simple fix in order to have our hash code stay within the size of our table: take the modulus of the hash code to map within our array.

```
1  public class BoundHashTable{
2       private Object[] storage;
3       public SimpleHashTable(int size){
4            storage = new Object[size];
5       }
6       public void add(Object o){
7            int ind = o.hashCode() % storage.length;
8            storage[ind] = o;
9       }
10      public void remove(Object o){
11           int ind = o.hashCode() % storage.length;
12           storage[ind] = null;
13      }
14      public boolean contains(Object o){
15           int ind = o.hashCode() % storage.length;
16           return storage[ind].equals(o);
17      }
18 }
```

# Hash Table - Collisions

When two different objects are hashed to the same value, a **collision** occurs. Of course, we would probably want to choose a better hash function, but we may be stuck with one (e.g. stuck with the java default). One way to handle this is starting at the hased location look for the next available spot.
This is known as **linear probing.**

```
1  public class CollHashTable{
2       //...
3       public void add(Object o){
4            int ind = o.hashCode() % storage.length;
5            int sLoc = ind;
6            while(storage[ind]!=null) {
7                 ind = (ind + 1) % storage.length;
8                 if(ind == sLoc) return; //cannot add, full
9            }
10           storage[ind] = o;
11      }
12      //...
13 }
```

# Hash Table - Collisions

When we want to check for an object or remove it with linear probing (like adding) it is no longer perfectly constant. Our methods will now iteratively look for the object/place to add.

```
1  public class CollHashTable{
2       //...
3       public boolean contains(Object o){
4            int ind = o.hashCode() % storage.length;
5            int sLoc = ind;
6            while(!o.equals(storage[ind])){
7                 if(storage[ind]==null) return false;
8                 ind = (ind + 1) % storage.length;
9                 if(ind == sLoc) return false; // checked it all
10           }
11           return true;
12      }
13      //...  for remove, just set location to null if found
             instead of returning true
14 }
```

# Hash Table - Clustering

With less and less collisions, our **amortized** cost gets closer and closer to $O(1)$. So we will want to minimize collisions. But if we, say, chose a horrible hash function where every object we want to add hashes to the same hash code, every single object will collide and have to be linearly probed...and essentially our amortized cost will go down to $O(n)$.

When we have a lot of collisions in one area of the table we say that there is **clustering** around a certain hash key/code/location.

**EXAMPLE:** Our generateBarcode() method from Project #2. This essentially generated a hash code. However, the letters N,R,S,T,A,E,I, and O are the most common letters of use in the english langauge so even if we had a table large enough to accomodate all possible barcodes, we would see clustering around barcodes generated with these letters.

# Hash Table - Rehashing

Rather than using Linear probing to avoid conflict we can **rehash** in other ways:

- **Quadratic Probing** - adjusts the index by plus/minus squares of a shift $s = (-1)^i * (\frac{i+1}{2})^2$,
  which for $i = 0, 1, 2, 3, 4, 5, ...$
  is $s = 0, -1, 1, -4, 4, -9, ...$ respectively.
- **Random Probing** - adjusts the index from a collision by a seeded (so we can access when same object added) random amount.

# Hash Table - Quadratic Probing

```
1  public class QuadHashTable{
2       //...
3       public int rehashVal(int i){
4            int nV = (i+1)/2;
5            nV*=nV;
6            if(i%2 == 1) nV=storage.length-nV;
7            return nV;
8       }
9       public void add(Object o){
10           int ind = o.hashCode() % storage.length;
11           int sLoc = ind;
12           int cInd = 0;
13           while(storage[ind]!=null) {
14                ind = (ind + rehashVal(cInd++)) % storage.length;
15                if(ind == sLoc) return; //cannot add, full
16           }
17           storage[ind] = o;
18      }
19      //...
20 }
```

# Hash Table - Random Probing

```
1  import java.util.Random;
2  public class RandHashTable{
3       //...
4       public int rehashVal(Random rand){
5            return rand.next(storage.length);
6       }
7       public void add(Object o){
8            int ind = o.hashCode() % storage.length;
9            int sLoc = ind;
10           Random rand = new Random(sLoc);
11           while(storage[ind]!=null) {
12                ind = (ind + rehashVal(rand)) % storage.length;
13                if(ind == sLoc) return; //cannot add, full
14           }
15           storage[ind] = o;
16      }
17      //...
18 }
```

# Hash Table - Buckets/Chains

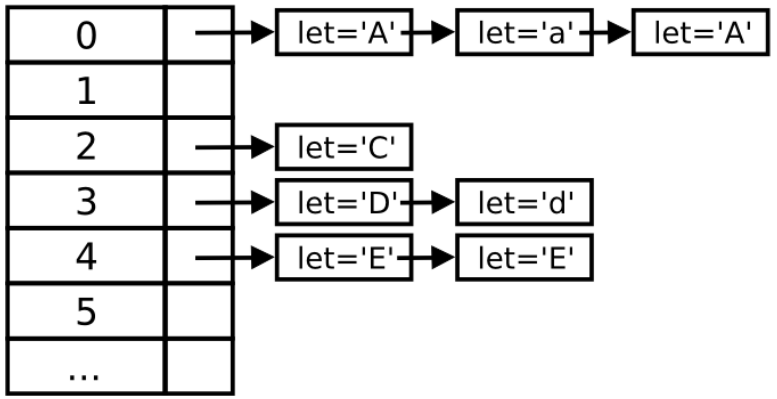
Rather than storing single values in the array and probing along the array (which can impact the spots we place the collided data), we can

- represent the table as a 2D array with multiple locations (columns) for each hash code (row/**bucket**). If a single row becomes full we can still rehash using some probing scheme.
- have linked lists inside those array locations that contain each object with the specified hash code. (**chains**)

**Buckets**

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | let='A' | let='a' | let='A' |
| 1 | | | |
| 2 | let='C' | | |
| 3 | let='D' | let='d' | |
| 4 | let='E' | let='E' | |
| 5 | | | |
| ... | ... | ... | ... |

**Chaining**



# Hash Table - Buckets

```
1  import java.util.Random;
2  public class BucketHashTable{
3       private Object[][] table;
4       //...
5       public void add(Object o){
6            int ind = o.hashCode() % table.length;
7            int sLoc = ind;
8            int bucketSlot = 0;
9            while(storage[ind][bucketSlot]!=null) {
10                while(bucketSlot!=table[ind].length-1 &&
11                     table[ind][bucketSlot]!=null)
12                          bucketSlot++;
13                if(table[ind][bucketSlot]!=null){
14                     bucketSlot = 0;
15                     ind = (ind + rehashVal(ind)) % table.length;
16                     if(ind == sLoc) return; //cannot add, full
17                }
18           }
19           table[ind][bucketSlot] = o;
20      }
21      //...
22 }
```

# Hash Table - Chaining

```
1  import java.util.Random;
2  public class ChainHashTable{
3       private ArrayList[] chains;
4       //...
5       public void add(Object o){
6            int ind = o.hashCode() % chains.length;
7            chains[ind].add(o);
8       }
9       //...
10 }
```

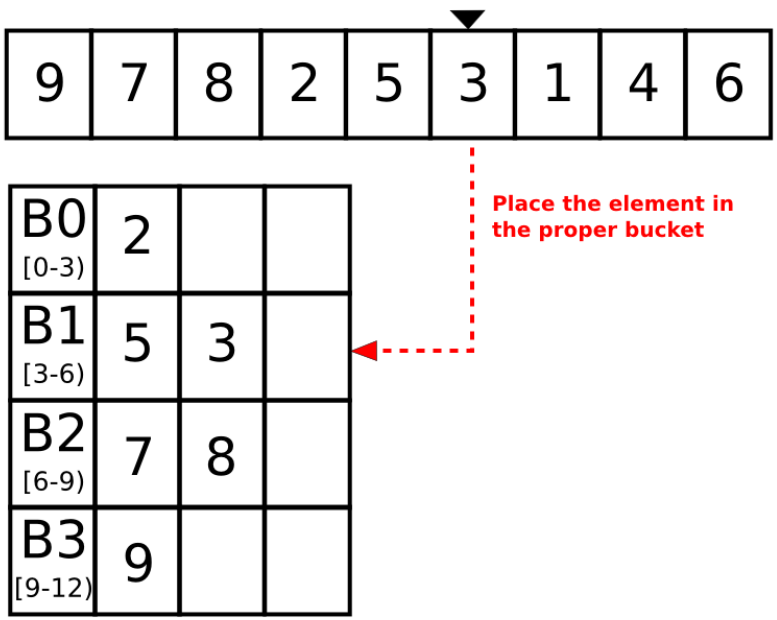
Provides a really simple solution to our collision problem, but in analysis we can clearly see that performance can be impacted, in the worst-case all elements hash to same chain $O(n)$ or hash to unique chains $O(1)$.

*Another way to improve performance is to have a secondary hash table at each location. i.e. hash the value using a secondary function for a second hash table.*

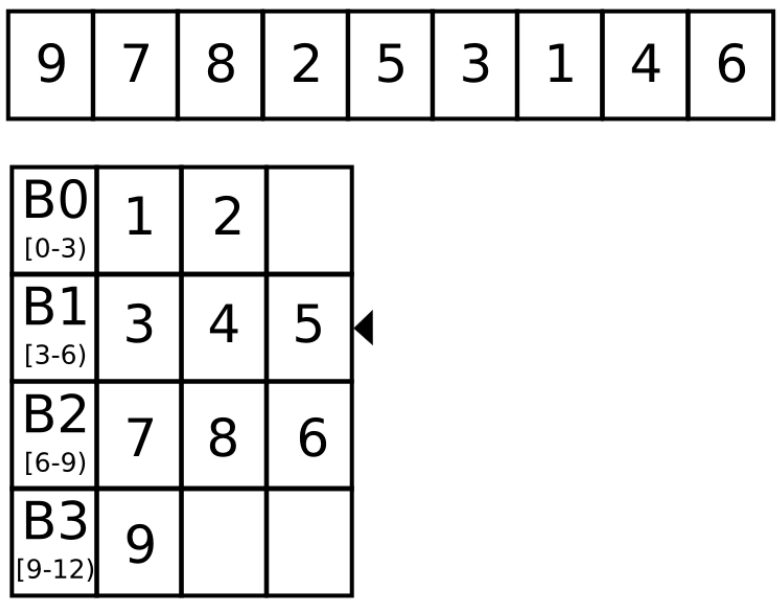
# Bucket Sort

Bucket sort can be thought of like a hash function. Given the set of data we create a uniform set of buckets for our data (for example with text we could create buckets of starting letters...but essentially can be thought of in the same way as creating a hash table, and then sorting each bucket).

**Iterate all elements placing data into buckets**

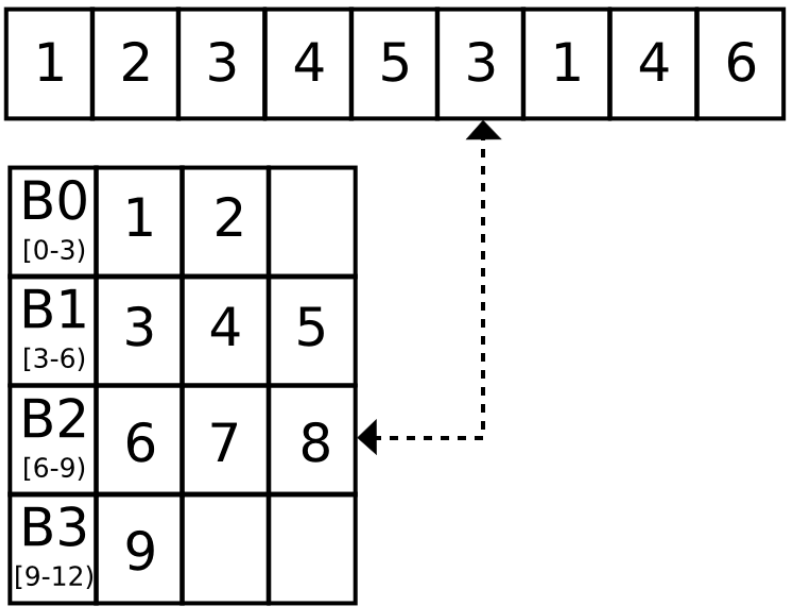



**Sort each bucket with favorite sorting technique!**



# Bucket Sort - 2

Once each bucket is sorted we can iterate over each bucket copying sorted data back into array.




Like hashing, bucket sort's performance really hinges on excellent choices for buckets that minimize the number of elements per bucket. In the worst case, all elements placed in the same bucket we can see that performance is only as good as the sorting technique chosen for each bucket.

But if we chose a buckets a number of buckets $b$ such that each bucket had $\alpha$ elements we can see that performance is given by $n + b * O(\alpha^2)$ if just using a naive sorting algorithm at each bucket. But its is possible to see that if $b < n$ and $\alpha$ is minimized we approach $O(n)$ sorting time.

# Radix Sort

Radix sort is similar to bucket sort, in that we will be placing elements into buckets (specifically buckets 0-9), as radix sort is only applicable to integer values. We will be looking at each digit of the number (can be most significant digit or least significant digit[LSD]) and sorting according to each digit of the numbers

**Iterate all elements placing data into buckets, by LSD**

| 225 | 335 | 230 | 220 | 235 |
|---|---|---|---|---|

| | | | |
|---|---|---|---|
| B0 | 230 | 220 | |
| B2 | | | |
| B3 | | | |
| B5 | 225 | 335 | 235 |

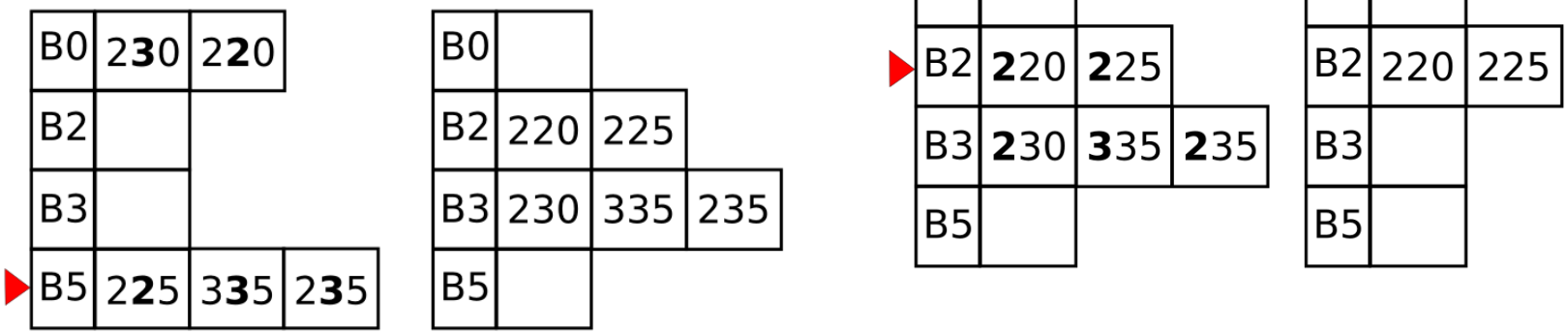
**Next iterate each bucket, placing in a new bucket set using next significant digit**

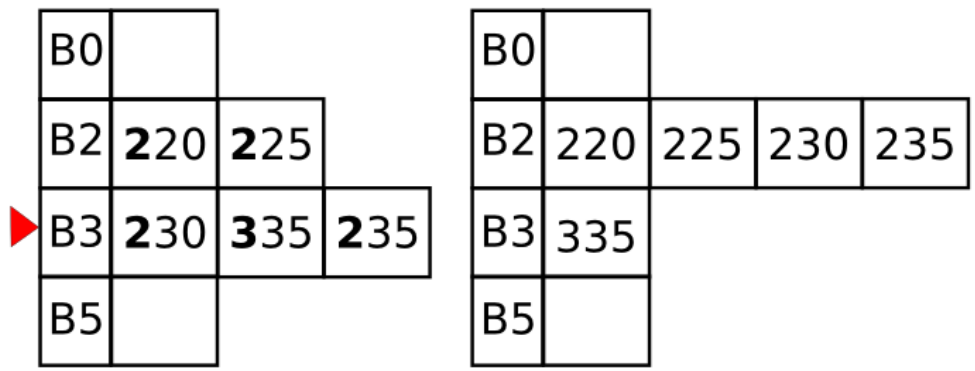
| | | | |
|---|---|---|---|
| B0 | 230 | 220 | |
| B2 | | | |
| B3 | | | |
| B5 | 225 | 335 | 235 |

| | |
|---|---|
| B0 | |
| B2 | 220 |
| B3 | 230 |
| B5 | |

# Radix Sort - 2

Continue iterating through each bucket, and when that is complete iterate each of the new buckets with the next significant digit and repeat.



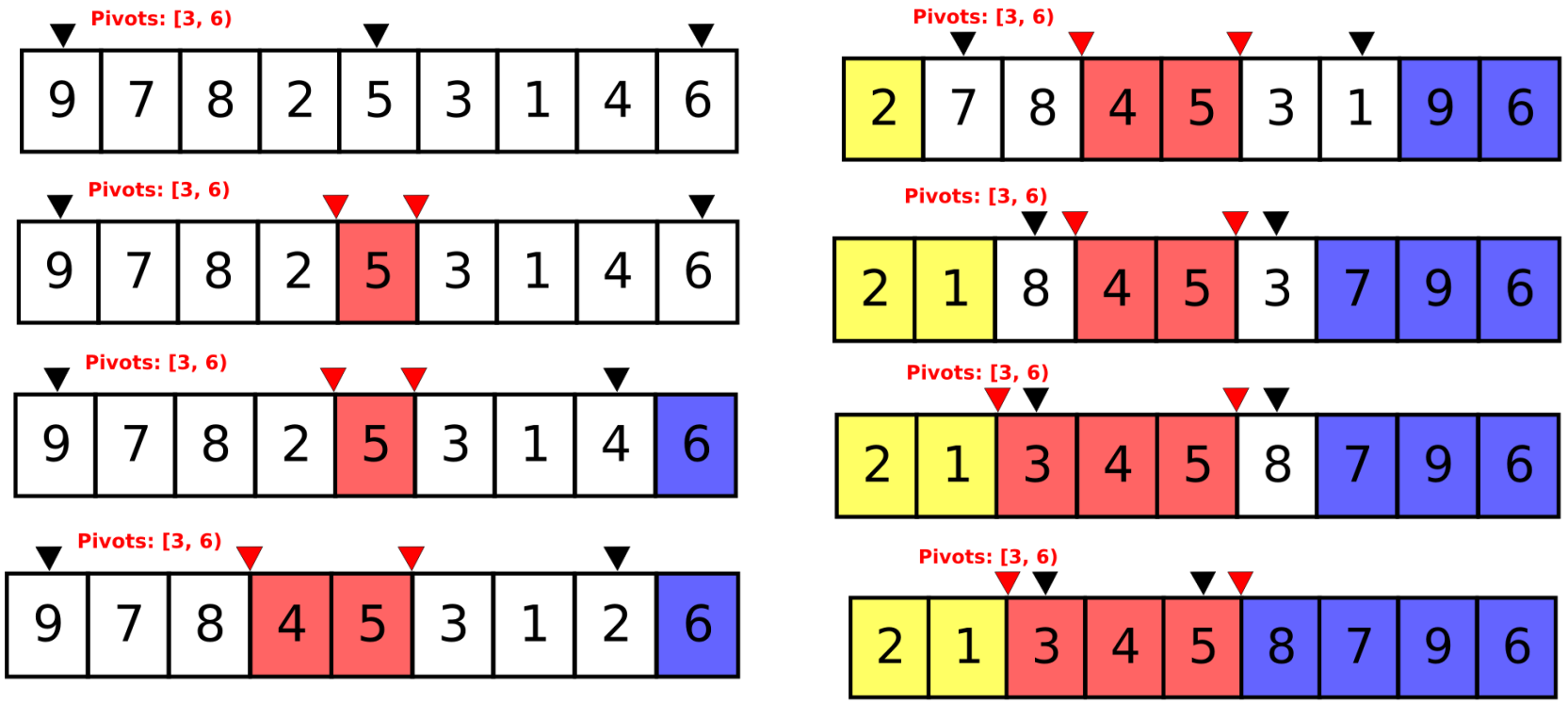

Iterate and copy sorted values back

# Radix Sort - 3

We can see that the performance at each step is n as all we are doing is iterating each element, and is a direct placement into a new bucket when found. The number of steps needed though depends on how many digits are in each number so the cost is given by $O(dn)$ where $d$ is the number of digits, and if minimized we could claim sorts in $O(n)$ time as well.

# M-Way Quick Sort

M-Way Quick Sort is a variant of the quick sort technique where $M - 1$ pivots are chosen in order to segment the data into M different segments, and like before a divide-and-conquer technique, where each segment has M-Way Quick sort performed on it. The data is iterated over and placed in the right location in the array: (example shows 3-Way):



# M-Way Quick Sort Runtime Analysis

Since we are looking at dividing a conquering we want to think about each aspect of the algorithm:

- How long to find the split?

  **SPLIT:** We are comparing each element to the pivot so $O(n)$

- How do we divide, when we find the split?

  **RECURSION:** Now that we are splitting into M parts (assuming each part is equal, which would result in in $O(log_M n)$ and typically does assuming a good pivot choices. Like normal/2-way Quick-Sort, it is entirely possible to find a pivots that splits each time into a 1-and-rest split, which would result in $O(n)$

**TOGETHER:** Like our previous divide-and-conquer techniques since we are find the splits at each recursion, it would be the product of the above values. Average case: $O(nlog_M n)$, Worst-case: $O(n^2)$. Provided we use a large enough M the $log_M n$ term shrinks towards 1 and approaches $O(n)$.