CS61B

Lecture 36: The End of Sorting

- An Intuitive, Analytical, and Empirical look at Radix vs. Comparison Sorting
- The Just-In-Time Compiler
- Radix Sorting Integers
- Summary

Intuitive: Radix Sort vs. Comparison Sorting

Merge Sort Runtime yellkey.com/wall

Merge Sort requires Θ(N log N) compares.

What is Merge Sort's runtime on strings of length W?

Merge Sort Runtime

Merge Sort requires Θ(N log N) compares.

What is Merge Sort's runtime on strings of length W?

- It depends!
  - Θ(N log N) if each comparison takes constant time.
    - Example: Strings are all different in top character.
  - Θ(WN log N) if each comparison takes Θ(W) time.
    - Example: Strings are all equal.

LSD vs. Merge Sort

The facts.

- Treating alphabet size as constant, LSD Sort has runtime Θ(WN).
- Merge Sort has runtime between Θ(N log N) and Θ(WN log N).

Which is better? It depends.

- When might LSD sort be faster?
- When might Merge Sort be faster?

LSD vs. Merge Sort (Your Answer)

The facts:

- Treating alphabet size as constant, LSD Sort has runtime $\Theta(WN)$.
- Merge Sort has runtime between $\Theta(N \log N)$ and $\Theta(WN \log N)$.

Which is better? It depends.

- When might LSD sort be faster?
  - Intuitively if $W < \log N$. **If N is really really big.**
  - If the Strings are relatively samish.
- When might Merge Sort be faster?
  - If the Strings are very different.

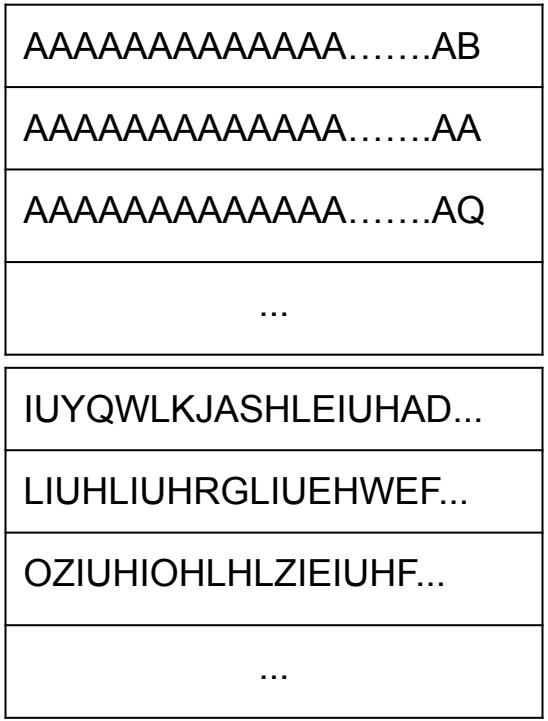
LSD vs. Merge Sort (My Answer)

The facts:

- Treating alphabet size as constant, LSD Sort has runtime Θ(WN).
- Merge Sort is between Θ(N log N) and Θ(WN log N).

Which is better? It depends.

- When might LSD sort be faster?
  - Sufficiently large N.
  - If strings are very similar to each other.
    - Each Merge Sort comparison costs Θ(W) time.
- When might Merge Sort be faster?
  - If strings are highly dissimilar from each other.
    - Each Merge Sort comparison is very fast.



Cost Model: Radix Sort vs. Comparison Sorting

Alternate Approach: Picking a Cost Model

An alternate approach is to pick a cost model.

- We'll use number of characters examined.
- By "examined", we mean:
  - Radix Sort: Calling charAt in order to count occurrences of each character.
  - Merge Sort: Calling charAt in order to compare two Strings.

MSD vs. Mergesort

Suppose we have 100 strings of 1000 characters each.

- Estimate the total number of characters examined by MSD Radix Sort **if all strings are equal**.

MSD vs. Mergesort

Suppose we have 100 strings of 1000 characters each.

- Estimate the total number of characters examined by MSD Radix Sort **if all strings are equal**.

For MSD Radix Sort, in the worst case (all strings equal), every character is examined exactly once. Thus, we have exactly 100,000 total character examinations.

MSD vs. Mergesort

Suppose we have 100 strings of 1000 characters each.

- Estimate the total number of characters examined by Merge Sort **if all strings are equal.**

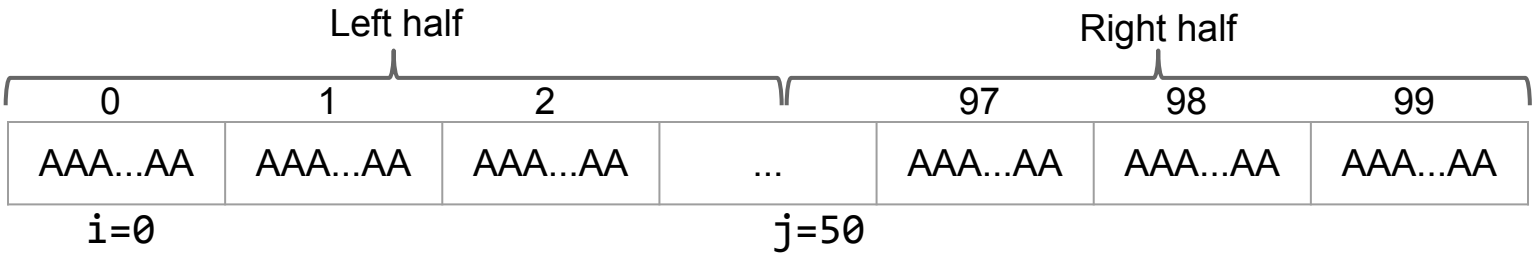
MSD vs. Mergesort

Suppose we have 100 strings of 1000 characters each.

- Estimate the total number of characters examined by Merge Sort **if all strings are equal.**

Merging 100 items, assuming equal items results in always picking left:

- Comparing A[0] to A[50]: 2000 character examinations.



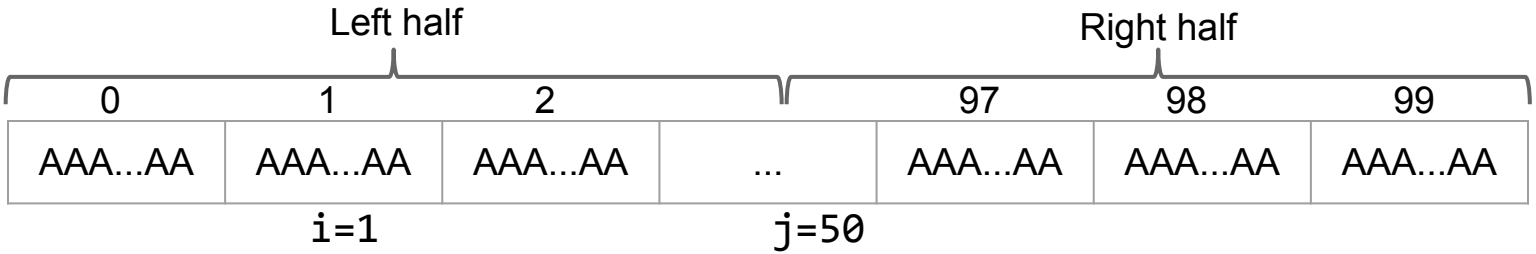
MSD vs. Mergesort

Suppose we have 100 strings of 1000 characters each.

- Estimate the total number of characters examined by Merge Sort **if all strings are equal.**

Merging 100 items, assuming equal items results in always picking left:

- Comparing A[0] to A[50]: 2000 character examinations.
- Comparing A[1] to A[50]: 2000 character examinations.



MSD vs. Mergesort

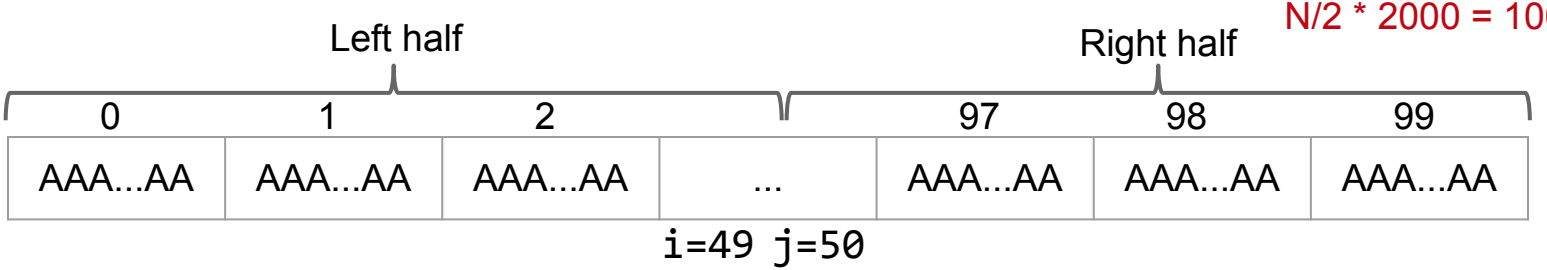
Suppose we have 100 strings of 1000 characters each.

- Estimate the total number of characters examined by Merge Sort **if all strings are equal.**

Merging 100 items, assuming equal items results in always picking left:

- Comparing A[0] to A[50]: 2000 character examinations.
- Comparing A[1] to A[50]: 2000 character examinations.
- … Comparing A[49] to A[50]: 2000 character examinations.
- Total characters examined: 50 * 2000 = 100000.

N/2 * 2000 = 1000N



MSD vs. Mergesort

Suppose we have 100 strings of 1000 characters each.

- Estimate the total number of characters examined by Merge Sort **if all strings are equal.**
- From previous slide: Merging N strings of 1000 characters requires N/2 * 2000 = 1000N examinations.

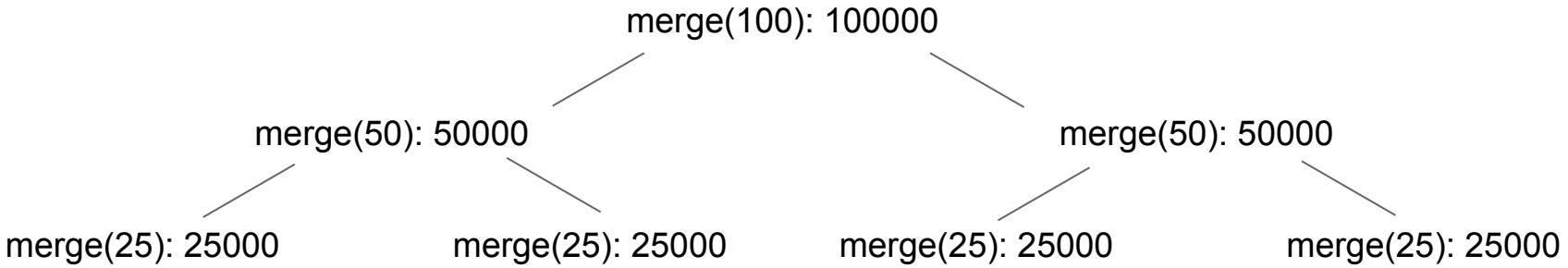
MSD vs. Mergesort

Suppose we have 100 strings of 1000 characters each.

- Estimate the total number of characters examined by Merge Sort **if all strings are equal.**
- From previous slide: Merging N strings of 1000 characters requires N/2 * 2000 = 1000N examinations.

In total, we must examine approximately $1000N \log_2 N$ total characters.

- 100000 + 50000*2 + 25000 * 4 + … = ~660,000 characters.



MSD vs. Mergesort Character Examinations

For N equal strings of length 1000, we found that:

- MSD radix sort will examine ~1000N characters (For N= 100: 100,000).
- Merge sort will examine ~1000N$\log_2$(N) characters (For N=100: 660,000).

If character examination are an appropriate cost model, we'd expect Merge Sort to be slower by a factor of $\log_2$N.

To see if we're right, we'll need to do a computational experiment.

Empirical Study: Radix Sort vs. Comparison Sorting

Computational Experiment Results

Computational experiment for W = 100.

- MSD and merge sort implementations are highly optimized versions taken from our optional algorithms textbook.
- Does our data match our runtime hypothesis?

Merge

N	Runtime	# chars
10,000		
100,000		
1,000,000		
10,000,000		

MSD

N	Runtime	# chars
10,000		
100,000		
1,000,000		
10,000,000		

Computational Experiment Results

Computational experiment for W = 100.

- MSD and merge sort implementations are highly optimized versions taken from our optional algorithms textbook.
- Does our data match our runtime hypothesis?

Merge

N	Runtime	# chars
10,000	0.05	13,801,600
100,000	0.26	170,780,800
1,000,000	0.87	2,013,286,400
10,000,000	13.8	23,757,632,000

MSD

N	Runtime	# chars
10,000	0.05	1,000,000
100,000	1.98	10,000,000
1,000,000	30.21	100,000,000
10,000,000	370.26	1,000,000,000

As we expected, Merge sort considers $\log_2 N$ times as many characters, e.g. $\log_2(10{,}000{,}000) = 23.25$

Computational Experiment Results

Computational experiment for W = 100.

- MSD and merge sort implementations are highly optimized versions taken from our optional algorithms textbook.
- Does our data match our runtime hypothesis? No!
  - Any guesses as to why not?

Merge

N	Runtime	# chars
10,000	0.05	13,801,600
100,000	0.26	170,780,800
1,000,000	0.87	2,013,286,400
10,000,000	13.8	23,757,632,000

MSD

N	Runtime	# chars
10,000	0.05	1,000,000
100,000	1.98	10,000,000
1,000,000	30.21	100,000,000
10,000,000	370.26	1,000,000,000

Computational Experiment Results (Your Answers)

Computational experiment for W = 100.

- MSD and merge sort implementations are highly optimized versions taken from our optional algorithms textbook.
- Does our data match our runtime hypothesis? No! Why not?
  - MSD needs more memory allocation. (I think not).
  - MSD across all the strings. Merge, going two adjacent strings at once. Could be caching related.

Merge

N	Runtime	# chars
10,000	0.05	13,801,600
100,000	0.26	170,780,800
1,000,000	0.87	2,013,286,400
10,000,000	13.8	23,757,632,000

MSD

N	Runtime	# chars
10,000	0.05	1,000,000
100,000	1.98	10,000,000
1,000,000	30.21	100,000,000
10,000,000	370.26	1,000,000,000

Computational Experiment Results (Your Answers)

Computational experiment for W = 100.

- MSD and merge sort implementations are highly optimized versions taken from our optional algorithms textbook.
- Does our data match our runtime hypothesis? No! Why not?
  - Mergesort: Uses the built in compareTo method -- this could be an issue, could be optimized in some crazy way.
  - There are potentially extra copy operations in MSD.

Merge

N	Runtime	# chars
10,000	0.05	13,801,600
100,000	0.26	170,780,800
1,000,000	0.87	2,013,286,400
10,000,000	13.8	23,757,632,000

MSD

N	Runtime	# chars
10,000	0.05	1,000,000
100,000	1.98	10,000,000
1,000,000	30.21	100,000,000
10,000,000	370.26	1,000,000,000

Computational Experiment Results (Your Answers)

Computational experiment for W = 100.

- MSD and merge sort implementations are highly optimized versions taken from our optional algorithms textbook.
- Does our data match our runtime hypothesis? No! Why not?
  - There are potentially extra copy operations in MSD. Our cost model may not capture everything that is important.

Merge

N	Runtime	# chars
10,000	0.05	13,801,600
100,000	0.26	170,780,800
1,000,000	0.87	2,013,286,400
10,000,000	13.8	23,757,632,000

MSD

N	Runtime	# chars
10,000	0.05	1,000,000
100,000	1.98	10,000,000
1,000,000	30.21	100,000,000
10,000,000	370.26	1,000,000,000

Computational Experiment Results (My Answers)

Computational experiment for W = 100.

- MSD and merge sort implementations are highly optimized versions taken from our optional algorithms textbook.
- Does our data match our runtime hypothesis? No! Why not?
  - Our cost model isn't representative of everything that is happening.
  - One particularly thorny issue: The "Just In Time" Compiler.

Merge

N	Runtime	# chars
10,000	0.05	13,801,600
100,000	0.26	170,780,800
1,000,000	0.87	2,013,286,400
10,000,000	13.8	23,757,632,000

MSD

N	Runtime	# chars
10,000	0.05	1,000,000
100,000	1.98	10,000,000
1,000,000	30.21	100,000,000
10,000,000	370.26	1,000,000,000

An Unexpected Factor: The Just-In-Time Compiler

Java's Just-In-Time Compiler secretly optimizes your code when it runs.

- The code you write is not necessarily the code that executes!
- As your code runs, the "interpreter" is watching everything that happens.
  - If some segment of code is called many times, the interpreter actually studies and re-implements your code based on what it learned by watching WHILE ITS RUNNING (!!).
    - Example: Performing calculations whose results are unused.
    - See this video if you're curious.

JIT Example

The code below creates Linked Lists, 1000 at a time.

- Repeating this 500 times yields an interesting result.

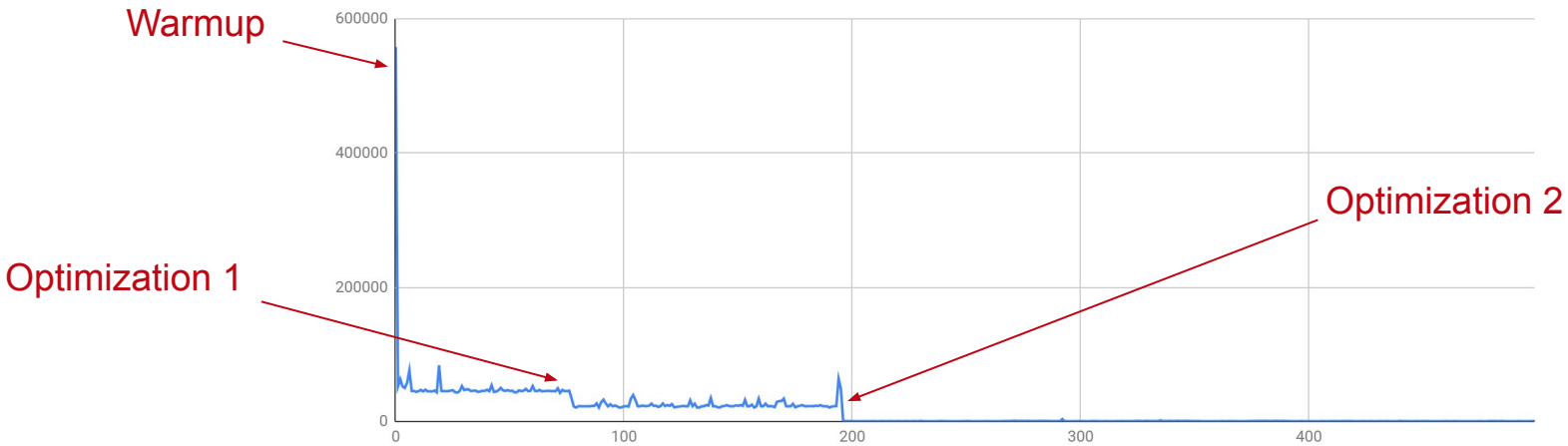
```
public class JITDemo1 {
  static final int NUM_LISTS = 1000;

  public static void main(String[] args) {
    for (int i = 0; i < 500; i += 1) {
      long startTime = System.nanoTime();
      for (int j = 0; j < NUM_LISTS; j += 1) {
        LinkedList<Integer> L = new LinkedList<>();
      }
      long endTime = System.nanoTime();
      System.out.println(i + ": " + endTime - startTime);
    }
  }
}
```

Create 1000 linked lists and print total time it takes.

JIT Example

The code below creates Linked Lists, 1000 at a time.

- Repeating this 500 times yields an interesting result.
- First optimization: Not sure what it does.
- Second optimization: Stops creating linked lists since we're not actually using them.



Rerunning Our Empirical Study With No JIT

Computational Experiments Results

Results with JIT disabled (using the -Xint option).

N	Runtime	# chars
1,000		
10,000		
100,000		

Merge

N	Runtime	# chars
1,000		
10,000		
100,000		

MSD

Computational Experiments Results

Results with JIT disabled (using the -Xint option).

- Both sorts are MUCH MUCH slower than before.
- Merge sort is slower than MSD (though not by as much as we predicted).
- What this tells us: The JIT was somehow able to massively optimize the compareTo calls.
  - Makes some intuitive sense: Comparing “AAA...A” to “AAA...A” over and over is redundant. I have no idea what it did specifically.

N	Runtime	# chars
1,000	0.05	1,000,800
10,000	1.41	13,801,600
100,000	16.58	170,780,800

Merge

N	Runtime	# chars
1,000	0.04	100,000
10,000	0.4	1,000,000
100,000	5.90	10,000,000

MSD

… So Which is Better? MSD or MergeSort?

We showed that if the JIT is enabled, merge sort is much faster for the case of equal strings, and slower if JIT is disabled.

- Since JIT is usually on, I'd say merge sort is better for this case.

Many other possible cases to consider:

- Almost equal strings (maybe the trick used by the JIT won't work?).
- Randomized strings.
- Real world data from some dataset of interest.

See code in lectureCode repo if you want to try running experiments yourself.

- In real world applications, you'd profile different implementations on real data and pick the best one.

Bottom Line: Algorithms Can Be Hard to Compare

Comparing algorithms that have the same order of growth is challenging.

- Have to perform computational experiments.
- In modern programming environments, experiments can be tricky due to optimizations like the JIT in Java.

Note: There's always the chance that some small optimization to an algorithm can make it significantly faster.

- Example: Change to Quicksort suggested by Vladimir Yaroslavskiy that we mentioned briefly in the quicksort lecture.

JIT Compilers Are Always Evolving

The JIT is a fantastically complex and important piece of code.

- Active area of research and development in the field of compilers.
- The old JIT compiler called C2 is so complicated that its code base is AFAIK being abandoned: “However, C2 has been delivering diminishing returns in recent years and no major improvements have been implemented in the compiler in the last several years. Not only that, but the code in C2 has become very hard to maintain and extend, and it is very hard for any new engineer to get up to speed with the codebase, which is written in a specific dialect of C++.” (from this site)
- If you like this stuff, try taking CS164 and maybe even try to get involved in compiler research.

Radix Sorting Integers (61C Preview)

Sorting Integers

Wow videos looked like trash in 2007.

Linear Time Sorting

As we've seen, estimating radix sort vs. comparison sort performance is very hard.

- But in the very large N limit, it's easy. Radix sort is simply faster!
  - Treating alphabet size as constant, LSD Sort has runtime $\Theta(WN)$.
  - Comparison sorts have runtime $\Theta(N \log N)$ in the worst case.

Linear Time Sorting

As we've seen, estimating radix sort vs. comparison sort performance is very hard.

- But in the very large N limit, it's easy. Radix sort is simply faster!
  - Treating alphabet size as constant, LSD Sort has runtime $\Theta(WN)$.
  - Comparison sorts have runtime $\Theta(N \log N)$ in the worst case.

Issue: We don't have a charAt method for integers.

- How would you LSD radix sort an array of integers?

Linear Time Sorting (Your Answers)

As we've seen, estimating radix sort vs. comparison sort performance is very hard.

- But in the very large N limit, it's easy. Radix sort is simply faster!
  - Treating alphabet size as constant, LSD Sort has runtime $\Theta(WN)$.
  - Comparison sorts have runtime $\Theta(N \log N)$ in the worst case.

Issue: We don't have a charAt method for integers.

- How would you LSD radix sort an array of integers?
  - Mods and floorMods and division operations and whatever → write yourself a getDigit method for integer.
    - This approach will be faster and more general.
  - Make them into a String. We get an alphabet size 10 String.
    - REductions style approach.

Linear Time Sorting (My Answer)

As we've seen, estimating radix sort vs. comparison sort performance is very hard.

- But in the very large N limit, it's easy. Radix sort is simply faster!
  - Treating alphabet size as constant, LSD Sort has runtime $\Theta(WN)$.
  - Comparison sorts have runtime $\Theta(N \log N)$ in the worst case.

Issue: We don't have a charAt method for integers.

- How would you LSD radix sort an array of integers?
  - Could convert into a String and treat as a base 10 number. Since maximum Java int is 2,000,000,000, W is also 10.
  - Could modify LSD radix sort to work natively on integers.
    - Instead of using charAt, maybe write a helper method like getDthDigit(int N, int d). Example: getDthDigit(15009, 2) = 5.

LSD Radix Sort on Integers

Note: There's no reason to stick with base 10!

- Could instead treat as a base 16, base 256, base 65536 number.

Example: 512,312 in base 16 is a 5 digit number:

- $512312_{10} = (7 \times 16^{4}) + (13 \times 16^{3}) + (1 \times 16^{2}) + (3 \times 16^{1}) + (8 \times 16^{0})$

Note this digit is greater than 9! That's OK, because we're in base 16.

Example: 512,312 in base 256 is a 3 digit number:

- $512312_{10} = (7 \times 256^{2}) + (209 \times 256^{1}) + (56 \times 256^{0})$

Note these digit are greater than 9! That's OK, because we're in base 256.

Relationship Between Base and Max # Digits

For Java integers:

- R=10, treat as a base 10 number. Up to 10 digits.
- R=16, treat as a base 16 number. Up to 8 digits.
- R=256, treat as a base 256 number. Up to 4 digits.
- R=65336, treat as a base 65536 number. Up to 2 digits.
- R=2147483647, treat as a base 2147483647 number (this is equivalent to counting sort). Has exactly 1 digit.

Interesting fact: Runtime depends on the alphabet size.

- As we saw with city sorting last time, R = 2147483647 will result in a very slow radix sort (since it's just counting sort).

Another Computational Experiment

Results of a computational experiment:

- Treating as a base 256 number (4 digits), LSD radix sorting integers easily defeats Quicksort.

Sort	Base	# of Digits	Runtime
Java QuickSort	N/A	N/A	10.9 seconds
LSD Radix Sort	2^4 = 16	8	3.6 seconds
LSD Radix Sort	2^8 = 256	4	2.28 seconds
LSD Radix Sort	2^16 = 65536	2	3.66 seconds
LSD Radix Sort	2^30 = 1073741824	2	20 seconds

Sorting 100,000,000 integers

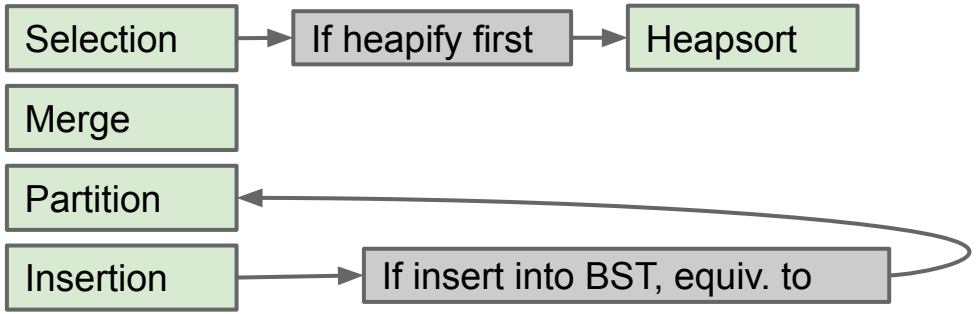
Sorting Summary

Sorting Landscape

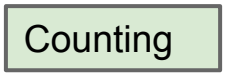
Below, we see the landscape of the sorting algorithms we've studied.

- Three basic flavors: Comparison, Alphabet, and Radix based.
- Each can be useful in different circumstances, but the important part was the analysis and the deep thought!
  - Hoping to teach you how to approach problems in general.

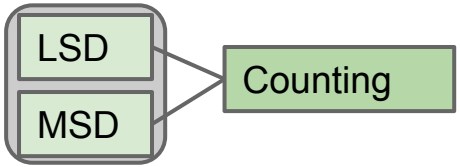
Comparison Based Sorting Algorithms:



Small-Alphabet **(e.g. Integer) Sorting Algorithms**:



Radix Sorting Algorithms:
(require a sorting subroutine)



Sorting vs. Searching

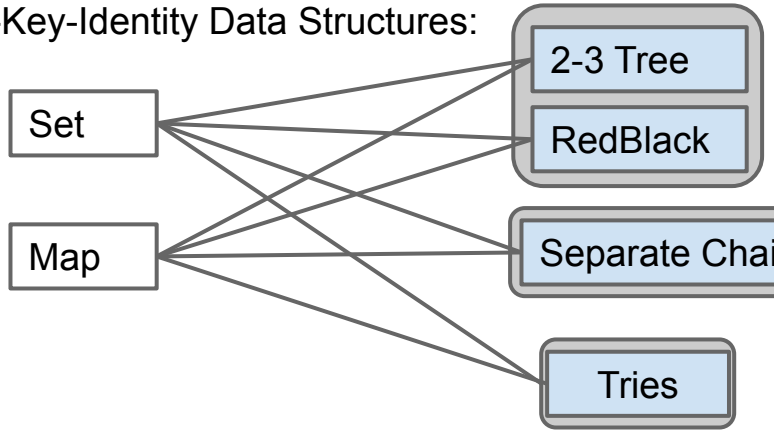
We've now concluded our study of the "sort problem."

- During the data structures part of the class, we studied what we called the "search problem": Retrieve data of interest.
- There are some interesting connections between the two.

Name	Storage Operation(s)	Primary Retrieval Operation	Retrieve By:
List	`add(key)` `insert(key, index)`	`get(index)`	index
Map	`put(key, value)`	`get(key)`	key identity
Set	`add(key)`	`containsKey(key)`	key identity
PQ	`add(key)`	`getSmallest()`	key order (a.k.a. key size)
Disjoint Sets	`connect(int1, int2)`	`isConnected(int1, int2)`	two int values

Partial list of search problem data structures.

Search-By-Key-Identity Data Structures:

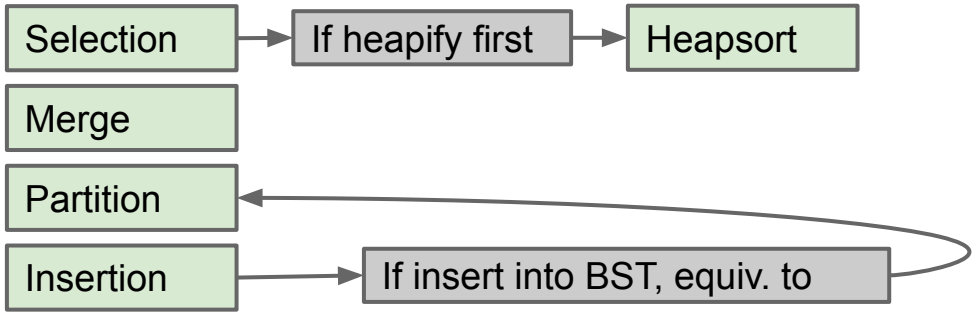


Searches using compareTo()
Analogous to **Comparison-Based**

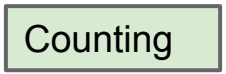
Searches using hashCode() and equals()
Roughly Analogous to **Integer Sorting**

Searches digit-by-digit
Roughly Analogous to **Radix Sorting**

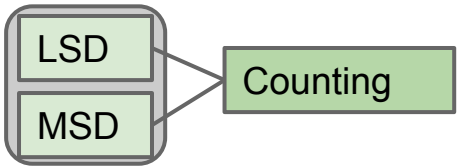
Comparison Based Sorting Algorithms:



Small-Alphabet **(e.g. Integer) Sorting Algorithms**:



Radix Sorting Algorithms:
(require a sorting subroutine)



Going Even Further

There's plenty more to explore!

Many of these ideas can be mixed and matched with others. Examples:

- What if we use quicksort as a subroutine for MSD radix sort instead of counting sort?
- Implementing the `comparable` interface means an object can be stored in our compareTo-based data structures (e.g. TreeSet), or sorted with our comparison based sorts. Is there a single equivalent interface that would allow storage in a trie AND radix sorting? What would that interface look like?
- If an object has both digits AND is comparable, could we somehow use an LLRB to improve radix sort in some way?