More sorting

CS 5006-7: Algorithms, C and Systems

Adrienne Slaughter, Joe Buck

Northeastern University

February 19, 2019

Section 1

Review of Sorts we've seen

Sorts we've seen

- Insertion Sort
- Mergesort
- Bubble Sort
- Heapsort
- Quicksort

Takeaways

- Comparison sorts can never do better than $n \lg n$
- Other sorts can do n
- Sorting is important
- Sorting usually helps make other problems easier

Comparison Sorts

All of these sorts require comparing two elements.

In fact, with these sorts, you can't sort the input without comparing *every* element with at least one other element.

Today, we're going to look at sorts that aren't comparing two elements directly, but doing something a little different.

Comparison Sorts

- Can't do better than $O(n \lg n)$.
- Why?

Slight aside: Parameterizing a Comparison Sort

When we start to sort, we ask (and answer!) a few questions:

Slight aside: Parameterizing a Comparison Sort

When we start to sort, we ask (and answer!) a few questions:

- What order do we want our input sorted– increasing, or decreasing?

Slight aside: Parameterizing a Comparison Sort

When we start to sort, we ask (and answer!) a few questions:

- What order do we want our input sorted– increasing, or decreasing?
- Are we sorting just a key, or an entire record?

Slight aside: Parameterizing a Comparison Sort

When we start to sort, we ask (and answer!) a few questions:

- What order do we want our input sorted– increasing, or decreasing?
- Are we sorting just a key, or an entire record?
- What should we do with equal keys?

Slight aside: Parameterizing a Comparison Sort

When we start to sort, we ask (and answer!) a few questions:

- What order do we want our input sorted– increasing, or decreasing?
- Are we sorting just a key, or an entire record?
- What should we do with equal keys?
- What about non-numerical data?

Slight aside: Parameterizing a Comparison Sort

When we start to sort, we ask (and answer!) a few questions:

- What order do we want our input sorted– increasing, or decreasing?
- Are we sorting just a key, or an entire record?
- What should we do with equal keys?
- What about non-numerical data?

Many algorithms support different answers to all these questions, but does that mean you have to re-implement every time?

Slight aside: Parameterizing a Comparison Sort

When we start to sort, we ask (and answer!) a few questions:

- What order do we want our input sorted– increasing, or decreasing?
- Are we sorting just a key, or an entire record?
- What should we do with equal keys?
- What about non-numerical data?

Many algorithms support different answers to all these questions, but does that mean you have to re-implement every time?

We usually do this by providing a ***comparator*** to a sort algorithm.

Slight aside: Parameterizing a Comparison Sort

When we start to sort, we ask (and answer!) a few questions:

- What order do we want our input sorted– increasing, or decreasing?
- Are we sorting just a key, or an entire record?
- What should we do with equal keys?
- What about non-numerical data?

Many algorithms support different answers to all these questions, but does that mean you have to re-implement every time?

We usually do this by providing a ***comparator*** to a sort algorithm.

Replace a hardcoded $<$ or $>$ with a ***function*** that compares a and b

Size of input: Why n^2 vs $n \lg n$ matters

Table: Number of operations for a given input size n

n	$n^2/4$	$n \lg n$
10		

Size of input: Why n^2 vs $n \lg n$ matters

Table: Number of operations for a given input size n

n	$n^2/4$	$n \lg n$
10	25	33

Size of input: Why n^2 vs $n \lg n$ matters

Table: Number of operations for a given input size n

n	$n^2/4$	$n \lg n$
10	25	33
100		

Size of input: Why n^2 vs $n \lg n$ matters

Table: Number of operations for a given input size n

n	$n^2/4$	$n \lg n$
10	25	33
100	2,500	664

Size of input: Why n^2 vs $n \lg n$ matters

Table: Number of operations for a given input size n

n	$n^2/4$	$n \lg n$
10	25	33
100	2,500	664
1,000		

Size of input: Why n^2 vs $n \lg n$ matters

Table: Number of operations for a given input size n

n	$n^2/4$	$n \lg n$
10	25	33
100	2,500	664
1,000	250,000	9,965

Size of input: Why n^2 vs $n \lg n$ matters

Table: Number of operations for a given input size n

n	$n^2/4$	$n \lg n$
10	25	33
100	2,500	664
1,000	250,000	9,965
10,000		

Size of input: Why n^2 vs $n \lg n$ matters

Table: Number of operations for a given input size n

n	$n^2/4$	$n \lg n$
10	25	33
100	2,500	664
1,000	250,000	9,965
10,000	25,000,000	132,877

Size of input: Why n^2 vs $n \lg n$ matters

Table: Number of operations for a given input size n

n	$n^2/4$	$n \lg n$
10	25	33
100	2,500	664
1,000	250,000	9,965
10,000	25,000,000	132,877
100,000		

Size of input: Why n^2 vs $n \lg n$ matters

Table: Number of operations for a given input size n

n	$n^2/4$	$n \lg n$
10	25	33
100	2,500	664
1,000	250,000	9,965
10,000	25,000,000	132,877
100,000	2,500,000,000	1,660,960

Size of input: Why n^2 vs $n \lg n$ matters

Table: Number of operations for a given input size n

n	$n^2/4$	$n \lg n$
10	25	33
100	2,500	664
1,000	250,000	9,965
10,000	25,000,000	132,877
100,000	2,500,000,000	1,660,960

Takeaway: For small inputs, it kinda doesn't matter what sort you choose– do whatever's easiest or satisfies other constraints. But the bigger the input is, the more it matters.

Size of input: Why n^2 vs $n \lg n$ matters

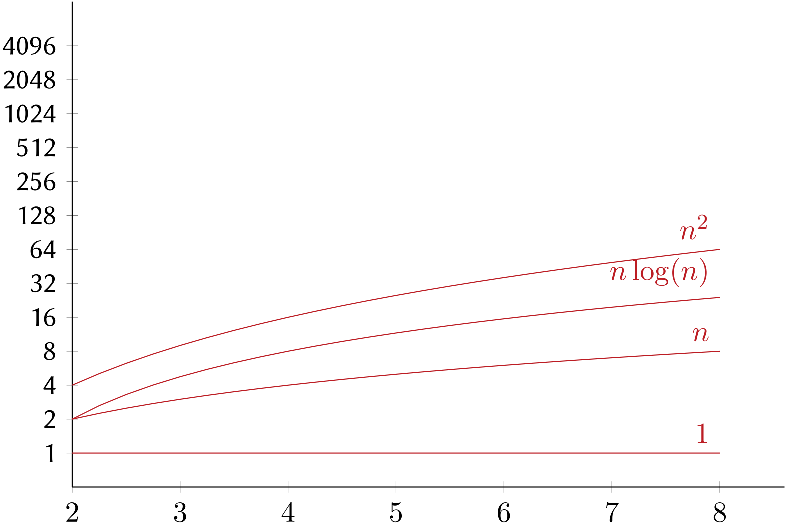
Table: Number of operations for a given input size n

n	$n^2/4$	$n \lg n$
10	25	33
100	2,500	664
1,000	250,000	9,965
10,000	25,000,000	132,877
100,000	2,500,000,000	1,660,960

Takeaway: For small inputs, it kinda doesn't matter what sort you choose– do whatever's easiest or satisfies other constraints. But the bigger the input is, the more it matters.

But what would be even better? $O(n)$!!

Growth of Functions



Section 2

Bucket Sort

A sorting problem

I have a bunch of playing cards to sort.

I want them sorted numerically ascending, and suits in the order ♥♦♣♠

How do I do it?

2♥	5♦	K♠	Q♣	A♥	6♣	4♠	J♥	K♦	6♠	···

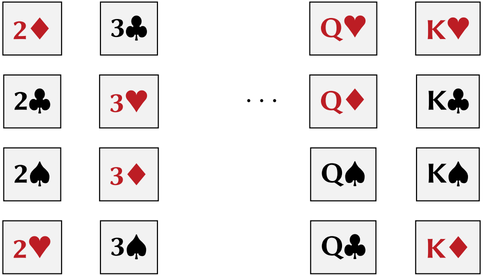
A sorting problem

One approach:

2♦	3♣		Q♥	K♥
2♣	3♥	. . .	Q♦	K♣
2♠	3♦		Q♠	K♠
2♥	3♠		Q♣	K♦

A sorting problem

One approach:



To summarize:

A sorting problem

One approach:

2♦	3♣		Q♥	K♥
2♣	3♥	. . .	Q♦	K♣
2♠	3♦		Q♠	K♠
2♥	3♠		Q♣	K♦

To summarize:

- Put all the cards into the right pile (face value).

A sorting problem

One approach:

2♦	3♣		Q♥	K♥
2♣	3♥	...	Q♦	K♣
2♠	3♦		Q♠	K♠
2♥	3♠		Q♣	K♦

To summarize:

- Put all the cards into the right pile (face value).
- Sort each pile by suit.

Sorting cards

To summarize:

- Put all the cards into the right pile (face value).
- Sort each pile by suit.

Bucket Sort: The Algorithm

This is a type of sorting called ***bucket sort***.
For input A with n elements:

- For each element in A, put it into the correct bucket B
- For each bucket B, use insertion sort to sort the items in the bucket.
- Concatenate the lists, starting with $B[0], B[1], ...B[n-1]$

Bucket Sort: Analysis

Every step of the algorithm is n in worst case, except for the insertion sort line.

Let n_i be the number of elements in bucket $B[i]$.

Expected time to sort the elements in $B[i]$ is $E[O(n_i^2)]$.

This can be re-written: the time to sort the element is $B[i] = O(E[n_i^2]]$

The time to sort all the elements in all of the buckets is:

$$\sum_{i=0}^{n-1} = O(E[n_i^2]) \Rightarrow O\left(\sum_{i=0}^{n-1} E[n_i^2]\right)$$

Bucket Sort: Analysis (cont.)

How do we evaluate $\sum_{i=0}^{n-1} E[n_i^2]$?

Well, we need to use what we know about the distribution of the elements into the buckets.

Let's assume we have n elements and n buckets. The probability p that an element falls into a given bucket $B[i]$ is $1/n$.

This probability follows the binomial distribution, so we know:

Mean: $E[n_i] = np = 1$

Variance: $Var[n_i] = np(1-p) = 1 - 1/n$

Bucket Sort: Analysis (cont.)

Using Mean and Variance from the previous slide:

$$\begin{aligned} E[n_i^2] &= Var(n_i) + E^2[n_i] \\ &= 1 - 1/n + 1^2 \\ &= 2 - 1/n \\ &= \Theta(1) \end{aligned}$$

Now, let's plug this into the earlier summation:

$$\sum_{i=0}^{n-1} E[n_i^2] = \sum_{i=0}^{n-1} \Theta(1)$$
$$\Rightarrow O(n)$$

This shows that the insertion sort step is $O(n)$. Therefore, **all** steps of the algorithm are $O(n)$, which makes the entire algorithm $O(n)$.

Bucket Sort: How does it compare?

- Performance will depend on how many buckets you have, compared to the number of inputs.
- If your bucketing can get the problem small enough, it doesn't matter what sort you use in each bucket.
- A good approach to start dealing with stream data– when you keep getting more input and don't know when it'll end.

Section 3

Counting Sort

Counting sort

Overview:

- Take a collection of items for input
- Create a new array, `counts` and initialize it to 0s.
- Go through the input array, and count up the number of each item.
- Go through the `counts` array and modify it by setting each element to it's value plus it's previous value.
- Use the `counts` to fill the output array in sorted order.

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	2	3	1	2	1

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	2	3	1	2	1

0	1	2	3	4	5
0	2	5	6	8	9

$$\texttt{cts}[i] = \texttt{cts}[i] + \texttt{cts}[i-1]$$

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	2	5	6	8	9

Output Array:

0	1	2	3	4	5	6	7	8
-	-	-	-	-	-	-	-	-

Now moving elements into the result array

Start at the end of the array

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	2	5	6	8	9

Output Array:

0	1	2	3	4	5	6	7	8
-	-	-	-	-	-	-	-	-

Now moving elements into the result array

Look at the value of the element

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	2	5	6	8	9

Output Array:

0	1	2	3	4	5	6	7	8
-	-	-	-	-	-	-	-	-

Now moving elements into the result array

Find the value in the counts array

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	2	$\boxed{5}$	6	8	9

Output Array:

0	1	2	3	4	5	6	7	8
-	-	-	-	-	-	-	-	-

Now moving elements into the result array

Use that to determine which index of the output array that element goes into

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	2	$\boxed{5}$	6	8	9

Output Array:

0	1	2	3	4	5	6	7	8
-	-	-	-	-	-	-	-	-

Now moving elements into the result array

Put the element into the output array.

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	2	5	6	8	9

Output Array:

0	1	2	3	4	5	6	7	8
-	-	-	-	2	-	-	-	-

Now moving elements into the result array

Reduce the counts array element.

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	2	4	6	8	9

Output Array:

0	1	2	3	4	5	6	7	8
-	-	-	-	2	-	-	-	-

Now moving elements into the result array

While there are more elements in the array, repeat.

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	2	4	6	8	9

Output Array:

0	1	2	3	4	5	6	7	8
-	-	-	-	2	-	-	-	-

Now moving elements into the result array

While there are more elements in the array, repeat.

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	2	4	6	8	$\boxed{9}$

Output Array:

0	1	2	3	4	5	6	7	8
-	-	-	-	2	-	-	-	-

Now moving elements into the result array

While there are more elements in the array, repeat.

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	2	4	6	8	8

Output Array:

0	1	2	3	4	5	6	7	8
-	-	-	-	2	-	-	-	5

Now moving elements into the result array

Dealing with element 6

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	2	4	6	8	8

Output Array:

0	1	2	3	4	5	6	7	8
-	-	-	-	2	-	-	4	5

Now moving elements into the result array

Dealing with element 5

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	$\boxed{2}$	4	6	7	8

Output Array:

0	1	2	3	4	5	6	7	8
-	1	-	-	2	-	-	4	5

Now moving elements into the result array

Dealing with element 4

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	1	4	6	7	8

Output Array:

0	1	2	3	4	5	6	7	8
-	1	-	2	2	-	-	4	5

Now moving elements into the result array

Dealing with element 3

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	1	3	6	7	8

Output Array:

0	1	2	3	4	5	6	7	8
1	1	-	2	2	-	-	4	5

Now moving elements into the result array

Dealing with element 2

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	0	3	6	$\boxed{7}$	8

Output Array:

0	1	2	3	4	5	6	7	8
1	1	-	2	2	-	4	4	5

Now moving elements into the result array

Dealing with element 1

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	0	3	$\boxed{6}$	6	8

Output Array:

0	1	2	3	4	5	6	7	8
1	1	-	2	2	3	4	4	5

Now moving elements into the result array

Dealing with element 0

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	0	3	6	6	8

Output Array:

0	1	2	3	4	5	6	7	8
1	1	2	2	2	3	4	4	5

Now moving elements into the result array

The Final Sorted Array

Input array:

0	1	2	3	4	5	6	7	8
2	3	4	1	2	1	4	5	2

Counts:

0	1	2	3	4	5
0	0	3	5	6	8

Output Array:

0	1	2	3	4	5	6	7	8
1	1	2	2	2	3	4	4	5

The Final Sorted array

1	1	2	2	2	3	4	4	5

The Final Sorted array

1	1	2	2	2	3	4	4	5

- It's sorted!

The Final Sorted array

1	1	2	2	2	3	4	4	5

- It's sorted!
- It's stable!

The Final Sorted array

1	1	2	2	2	3	4	4	5

- It's sorted!
- It's stable!
- Note: the items that were first in the input are first in the output.

The Final Sorted array

1	1	2	2	2	3	4	4	5

- It's sorted!
- It's stable!
- Note: the items that were first in the input are first in the output.
- Why did we start at the **end** of the array when filling the output array?

The Final Sorted array

1	1	2	2	2	3	4	4	5

- It’s sorted!
- It’s stable!
- Note: the items that were first in the input are first in the output.
- Why did we start at the **end** of the array when filling the output array?
- What are some advantages?

The Final Sorted array

1	1	2	2	2	3	4	4	5

- It's sorted!
- It's stable!
- Note: the items that were first in the input are first in the output.
- Why did we start at the **end** of the array when filling the output array?
- What are some advantages?
- What are some drawbacks?

Counting Sort Analysis

How long does it take to sort with Counting Sort?

Counting Sort Analysis

How long does it take to sort with Counting Sort?

- Initializing the count array takes k

Counting Sort Analysis

How long does it take to sort with Counting Sort?

- Initializing the count array takes k
- The first pass through the input array takes n

Counting Sort Analysis

How long does it take to sort with Counting Sort?

- Initializing the count array takes k
- The first pass through the input array takes n
- The pass through the counts array takes k

Counting Sort Analysis

How long does it take to sort with Counting Sort?

- Initializing the count array takes k
- The first pass through the input array takes n
- The pass through the counts array takes k
- Putting the elements in the output array takes n

Counting Sort Analysis

How long does it take to sort with Counting Sort?

- Initializing the count array takes k
- The first pass through the input array takes n
- The pass through the counts array takes k
- Putting the elements in the output array takes n
- $\Rightarrow O(k + n)$

Section 4

Radix Sort

RAY-dix, because even though Radix Sort is ***rad***, we don't call it rad.

Let’s sort some big numbers

314

100

193

933

721

709

428

591

222

Radix Sort

Using a stable sort:

Sort by the least significant digit (right-most digit)

314
100
193
933
721
709
428
591
222

Radix Sort

Using a stable sort:

Sort by the next least sig digit (middle)

314		100
100		721
193		591
933		222
721	$\longrightarrow$	193
709		933
428		314
591		428
222		709

Radix Sort

Using a stable sort:

Finally, sort by the most significant digit (leftmost digit)

314	100	100
100	721	709
193	591	314
933	222	721
721 $\longrightarrow$	193 $\longrightarrow$	222
709	933	428
428	314	933
591	428	591
222	709	193

Radix Sort

Using a stable sort:

314		100		100		100
100		721		709		193
193		591		314		222
933		222		721		314
721	→	193	→	222	→	428
709		933		428		591
428		314		933		709
591		428		591		721
222		709		193		993

Radix Sort Summary

- We take d passes through the input array, where d is the number of digits we're sorting on!
- Commonly used to sort things like dates with year, month and day.

Section 5

Summary of Sorts

Sorts we've seen

- Insertion Sort
- Mergesort
- Bubble Sort
- Heapsort
- Quicksort
- Bucket Sort
- Counting Sort
- Radix Sort

Insertion Sort

- For each item in a list, put it in the right place earlier in the list, moving elements up as needed.
- Sort the list from left to right such that the left side is always sorted.
- Good for small lists; not good for large lists

Mergesort

- Split the input list in half; sort each half; combine the halves.
- Intro to divide and conquer

Bubble Sort

- Go through the list n times, comparing the first element to the next, swapping if needed, "bubbling" the largest element up to the end of the list with each iteration.

Heapsort

- Put all the elements in a heap, which ensures the biggest (or smallest) element is at the top of the tree. Remove the root, re-heapify the rest of the tree, and keep doing this. The elements are pulled off in sorted order.
- Sorts in place
- Running time $O(n \lg n)$

Quicksort

- Partition the list around a smartly chosen pivot, then sort the list on each side of the pivot.
- Generally does well on a randomly ordered input, but does poorly on a nearly sorted input.
- Introduced the idea of randomization, both for shuffling the input and choosing the pivot.

Bucket Sort

- Go through the input list, and put each element into an appropriate bucket. Sort each bucket.
- Good if you know the input values are about evenly distributed among the buckets you have.

Counting Sort

- Count the number of each element in a list; use that to take the elements out of the unsorted input and put it into the output array in a sorted manner.

Radix Sort

- Sort all the items first by the least significant digit, then iteratively until you finally sort by the most significant digit.
- Requires a stable sort, but is an approach that can be used with "any" other sort: the algorithm is a general approach, rather than a specific sort.

How to choose a sort

How to choose a sort

- How many items will you be sorting?

How to choose a sort

- How many items will you be sorting?
- Will there be duplicates?

How to choose a sort

- How many items will you be sorting?
- Will there be duplicates?
- What do you know about your data?

How to choose a sort

- How many items will you be sorting?
- Will there be duplicates?
- What do you know about your data?
  - Is it already partially sorted?

How to choose a sort

- How many items will you be sorting?
- Will there be duplicates?
- What do you know about your data?
  - Is it already partially sorted?
  - Do you know the distribution of values?

How to choose a sort

- How many items will you be sorting?
- Will there be duplicates?
- What do you know about your data?
  - Is it already partially sorted?
  - Do you know the distribution of values?
  - Are the items very long or difficult to compare?

How to choose a sort

- How many items will you be sorting?
- Will there be duplicates?
- What do you know about your data?
  - Is it already partially sorted?
  - Do you know the distribution of values?
  - Are the items very long or difficult to compare?
  - Is the range of values very small?

How to choose a sort

- How many items will you be sorting?
- Will there be duplicates?
- What do you know about your data?
  - Is it already partially sorted?
  - Do you know the distribution of values?
  - Are the items very long or difficult to compare?
  - Is the range of values very small?
  - Do you have to worry about disk access?

How to choose a sort

- How many items will you be sorting?
- Will there be duplicates?
- What do you know about your data?
  - Is it already partially sorted?
  - Do you know the distribution of values?
  - Are the items very long or difficult to compare?
  - Is the range of values very small?
  - Do you have to worry about disk access?
  - How much time do you have to write and debug and turn your implementation?

Why so much sorting?

Learning sorts is like learning scales for musicians.

Why so much sorting?

Learning sorts is like learning scales for musicians.

- Lots of other algorithms are built around various sorting algorithms.

Why so much sorting?

Learning sorts is like learning scales for musicians.

- Lots of other algorithms are built around various sorting algorithms.
- Different ideas around algorithms manifest themselves in different sorting algorithms
  - Divide and conquer
  - Randomization
  - Data structures
  - Recursion

Why so much sorting?

Learning sorts is like learning scales for musicians.

- Lots of other algorithms are built around various sorting algorithms.
- Different ideas around algorithms manifest themselves in different sorting algorithms
  - Divide and conquer
  - Randomization
  - Data structures
  - Recursion
- Computers (historically) have spent more time sorting than anything else.
  - Knuth (in 1998!) claimed that more than 25% of cycles were spent sorting data.
  - Sorting is still one of the most ubiquitous computing problems

Why so much sorting?

Learning sorts is like learning scales for musicians.

- Lots of other algorithms are built around various sorting algorithms.
- Different ideas around algorithms manifest themselves in different sorting algorithms
  - Divide and conquer
  - Randomization
  - Data structures
  - Recursion
- Computers (historically) have spent more time sorting than anything else.
  - Knuth (in 1998!) claimed that more than 25% of cycles were spent sorting data.
  - Sorting is still one of the most ubiquitous computing problems
- It's the most thoroughly-studied problems in CS.
  - So many algorithms; each has the particular case where it performs better than other algorithms.

Applications of Sorting

It turns out that sorting makes a bunch of other problems really easy to solve:

Applications of Sorting

It turns out that sorting makes a bunch of other problems really easy to solve:

- **Searching:** Binary search is great, but requires sorted data. Once data is sorted, it's easy to search.

Applications of Sorting

It turns out that sorting makes a bunch of other problems really easy to solve:

- **Searching:** Binary search is great, but requires sorted data. Once data is sorted, it's easy to search.
- **Closest pair:** Given a set of m numbers, how do you find the pair of numbers that have the smallest difference between them?

Applications of Sorting

It turns out that sorting makes a bunch of other problems really easy to solve:

- **Searching:** Binary search is great, but requires sorted data. Once data is sorted, it's easy to search.
- **Closest pair:** Given a set of m numbers, how do you find the pair of numbers that have the smallest difference between them?
- **Element uniqueness:** Are there any duplicates in a given set of n items? (A special case of the closest pair)

Applications of Sorting

It turns out that sorting makes a bunch of other problems really easy to solve:

- **Searching:** Binary search is great, but requires sorted data. Once data is sorted, it's easy to search.
- **Closest pair:** Given a set of m numbers, how do you find the pair of numbers that have the smallest difference between them?
- **Element uniqueness:** Are there any duplicates in a given set of n items? (A special case of the closest pair)
- **Frequency distribution:** Given a set of n items, which element occurs the largest number of times in the set? Note, this enables not just calculating frequencies, but can also support the question "How many times does item k occur?".

Applications of Sorting

It turns out that sorting makes a bunch of other problems really easy to solve:

- **Searching:** Binary search is great, but requires sorted data. Once data is sorted, it's easy to search.
- **Closest pair:** Given a set of m numbers, how do you find the pair of numbers that have the smallest difference between them?
- **Element uniqueness:** Are there any duplicates in a given set of n items? (A special case of the closest pair)
- **Frequency distribution:** Given a set of n items, which element occurs the largest number of times in the set? Note, this enables not just calculating frequencies, but can also support the question "How many times does item k occur?".
- **Selection:** What is the kth largest number in an array? If the array is sorted, lookup is constant.

Let's solve a problem

Give an efficient algorithm to determine whether two sets (of size m and n) are disjoint. Analyze the worst-case complexity in terms of m and n, considering the case where m is substantially smaller than n.

Let's solve a problem

Give an efficient algorithm to determine whether two sets (of size m and n) are disjoint. Analyze the worst-case complexity in terms of m and n, considering the case where m is substantially smaller than n.

1. **Sort the big set first:** Sort the big set in $O(n \lg n)$. Now, do a binary search with each of the m elements in the second set to see if it exists in the big set. The total time will be $O((n + m) \log n)$.

Let's solve a problem

Give an efficient algorithm to determine whether two sets (of size m and n) are disjoint. Analyze the worst-case complexity in terms of m and n, considering the case where m is substantially smaller than n.

1. **Sort the big set first:** Sort the big set in $O(n \lg n)$. Now, do a binary search with each of the m elements in the second set to see if it exists in the big set. The total time will be $O((n+m)\log n)$.
2. **Sort the little set first:** Sort the small set in $O(m \log m)$. Do a binary search with each of the n elements in the big set to see if it exists in the little one. $O((n+m)\log m)$

Let's solve a problem

Give an efficient algorithm to determine whether two sets (of size m and n) are disjoint. Analyze the worst-case complexity in terms of m and n, considering the case where m is substantially smaller than n.

1. **Sort the big set first:** Sort the big set in $O(n \lg n)$. Now, do a binary search with each of the m elements in the second set to see if it exists in the big set. The total time will be $O((n+m)\log n)$.
2. **Sort the little set first:** Sort the small set in $O(m \log m)$. Do a binary search with each of the n elements in the big set to see if it exists in the little one. $O((n+m)\log m)$
3. **Sort both sets:** When both sets are sorted, we can compare the smallest elements of the two sets and discard the smaller if they don't match. Repeat this on the smaller sets, testing for duplication in linear time after sorting. Total time is $O(n \log n + m \log m + n + m)$

Which is best?

Which is best?

- Small set sorting is faster than sorting a bigger set ($\log m < \log n$)

Which is best?

- Small set sorting is faster than sorting a bigger set ($\log m < \log n$)
- Also, $(n + m) \log m$ must be asymptotically smaller than $n \log n$ since $n + m < 2n$ when $m < n$.

Which is best?

- Small set sorting is faster than sorting a bigger set ($\log m < \log n$)
- Also, $(n + m) \log m$ must be asymptotically smaller than $n \log n$ since $n + m < 2n$ when $m < n$.
- Sorting the small set is the best option.

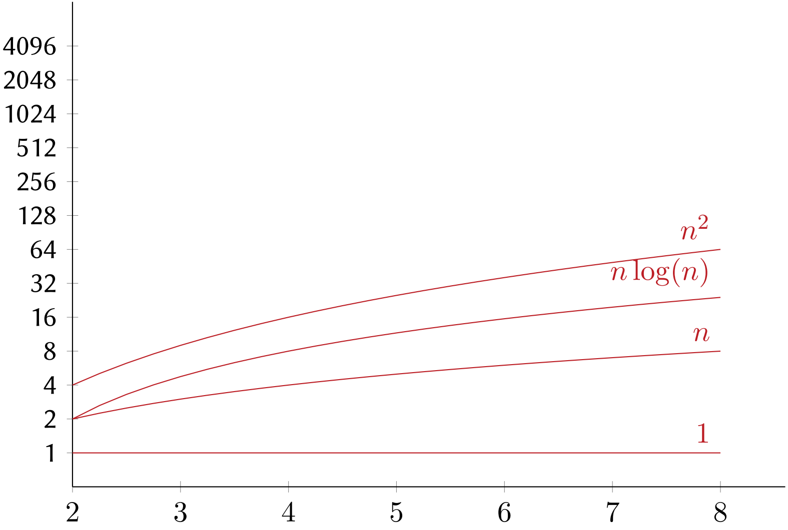
Which is best?

- Small set sorting is faster than sorting a bigger set ($\log m < \log n$)
- Also, $(n+m)\log m$ must be asymptotically smaller than $n \log n$ since $n + m < 2n$ when $m < n$.
- Sorting the small set is the best option.
- **Best approach:** Use hashing. Build a hash table containing elements of both sets. When there's a collision, check that they are actually identical.

Sorting Algorithms and their runtime

Algorithm	Worst	Average	Best
Insertion sort	$O(n^2)$	$\Theta(n^2)$	$\Omega(n)$
Mergesort	$O(n \lg n)$	$\Theta(n \lg n)$	$\Omega(n \lg n)$
Heapsort	$O(n \lg n)$	$\Theta(n \lg n)$	$\Omega(n \lg n)$
Quicksort	$O(n^2)$	$\Theta(n \lg n)$	$\Omega(n \lg n)$
Selection sort	$O(n^2)$	$\Theta(n^2)$	$\Omega(n^2)$
Counting sort	$O(n + k)$	$\Theta(n + k)$	$\Omega(n + k)$
Radix sort	$O(nk)$	$\Theta(nk)$	$\Omega(nk)$
Bucket sort	$O(n^2)$	$\Theta(n + k)$	$\Omega(n + k)$

Growth of Functions



Takeaways

- Comparison sorts can never do better than $n \lg n$
- Other sorts can do n
- Sorting is important
- Sorting usually helps make other problems easier

Topics for Midterm

- Basics
  - What is a VM? OS? vi/emacs?
  - Common git commands
  - How to compile (gcc)
  - Makefile, targets
  - Basic Linux commands (cp, mv, rm, cd, ls, ...)
- C
  - loops– for vs while, translate from one to the other
  - runtime of code chunks
  - function signature
  - function prototype
  - function definition
  - return type
  - header files
  - structs
  - . vs −¿
  - arrays, index
  - string terminator
  - how to get the address of a struct
  - heap vs stack
  - enum
  - declare, initialize array
  - valgrind– what is it? how to use it?
  - stack, queue, linked lists, graphs (adjacency list, matrix)
  - stack, queue, linked list implementations
  - typical function names/operators for basic data structures

Topics for Midterm

- Trees
  - binary tree vs heap vs BST
  - root/sibling/parent/child/internal/leaf/level vs height
  - BFS, DFS
  - insert/remove nodes in binary tree, heap, BST
  - implement as “linked list” vs array
- Graphs
  - List vertices, edges
  - Draw graph from various reps
  - Adjacency list vs matrix
  - Djikstra’s
  - DAG
  - topological ordering
  - strongly connected components