Lecture 6: Sorting

CS 5006: Algorithms

Adrienne Slaughter

Northeastern University

June 14, 2018

1. Logistics
2. Heap sort
   - A new data structure: Heap
   - Heapsort
   - Priority Queue
3. Quicksort
   - Partitioning
   - The Quicksort Algorithm
   - Analysis: Best Case
   - Analysis: Worst Case
   - Intuition
   - Average case analysis: Solve the recurrence
   - Picking a better pivot
   - Randomizing Quicksort
4. Linear time sorting
   - Bucket Sort
   - Counting Sort
   - Radix Sort
5. Sorting Recap
6. Summary

Agenda

- Quiz
- Logistics
- Sorting

Logistics

- Assignment 4
- Assignment 5
- Assignment 6
- Final
  - In 2 weeks
  - Can cover ALL lectures (including next week, even though there is no homework on it)
  - Questions will come primarily from the book

Topics

So. Much. Sorting.

- Sorting
- Review sorting we've seen
  - Selection sort
  - Bubble sort
  - Merge sort
- Heap sort
- Quicksort
- Linear time sorting

Subsection 1

A new data structure: Heap

Heap Definition

A ***binary heap*** is defined to be a ***complete binary tree*** that satisfies the ***heap ordering property***:

Heap Ordering Property

min-heap: The value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.

max-heap: The value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.

Heap Definition

A ***binary heap*** is defined to be a ***complete binary tree*** that satisfies the ***heap ordering property***:

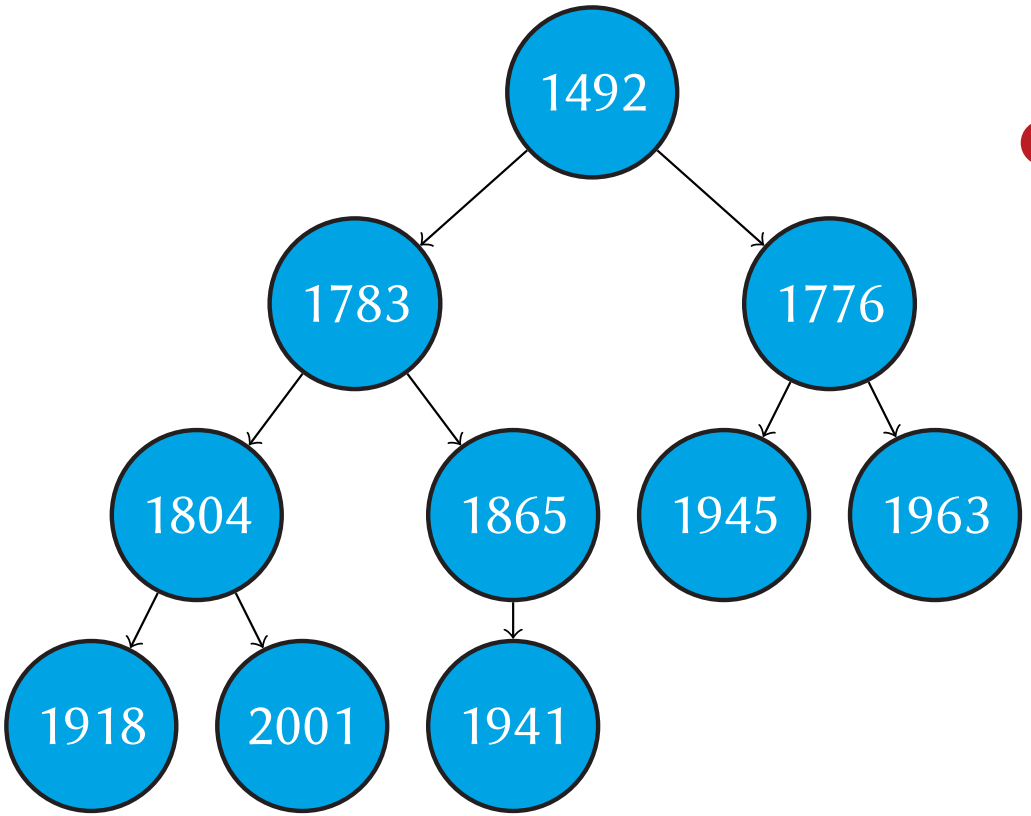
Heap Ordering Property

min-heap: The value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.

max-heap: The value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.

For the next few slides/examples, I'm going to assume we're working with min-heaps.

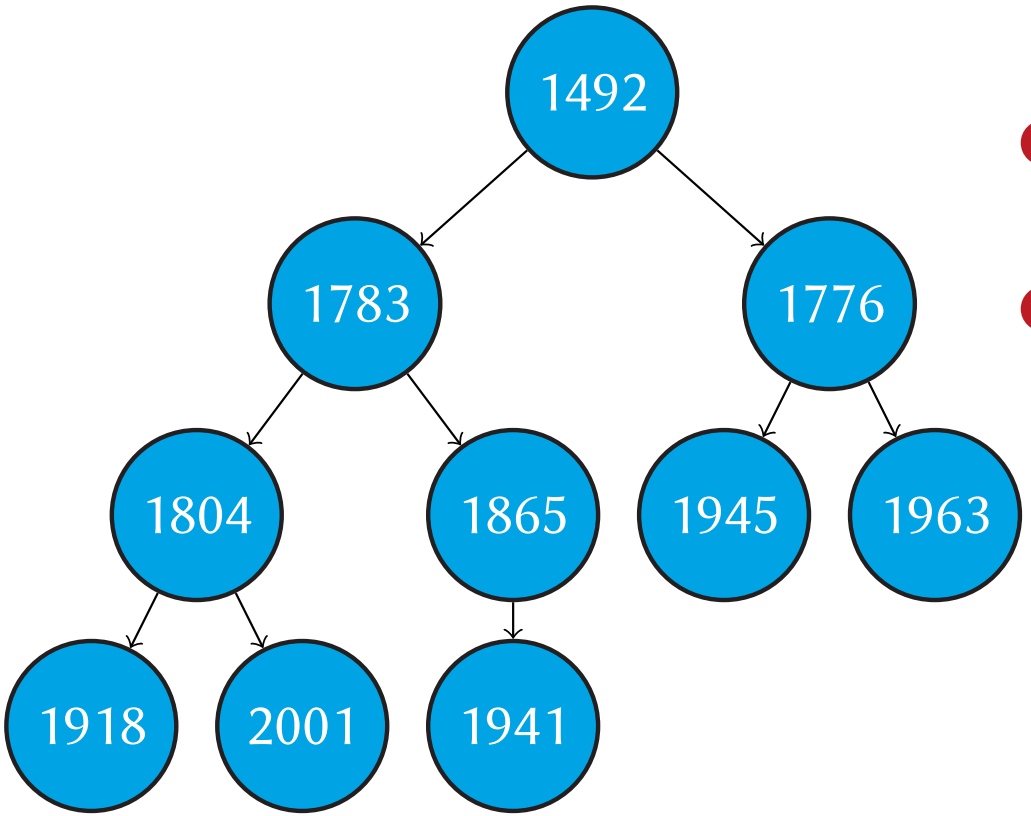
Binary Heaps



Some observations:

1. All leaves are on two adjacent levels.

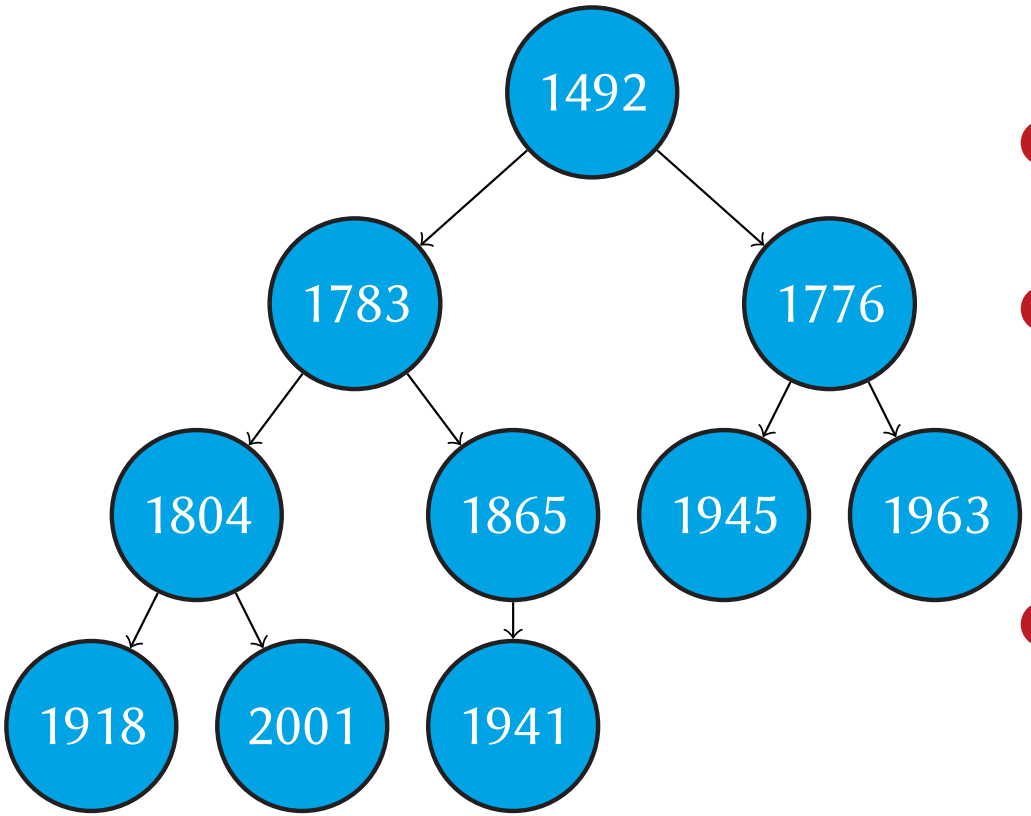
Binary Heaps



Some observations:

1. All leaves are on two adjacent levels.
2. All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.

Binary Heaps



Some observations:

1. All leaves are on two adjacent levels.
2. All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.
3. The value in root is less than or equal to all its children, and the left and right subtrees are again binary heaps.

How do we represent this in code?

Array-Based Heap

The most natural representation of this binary tree would involve storing each value in a node with pointers to its two children. (a la linked list or edge adjacency list)

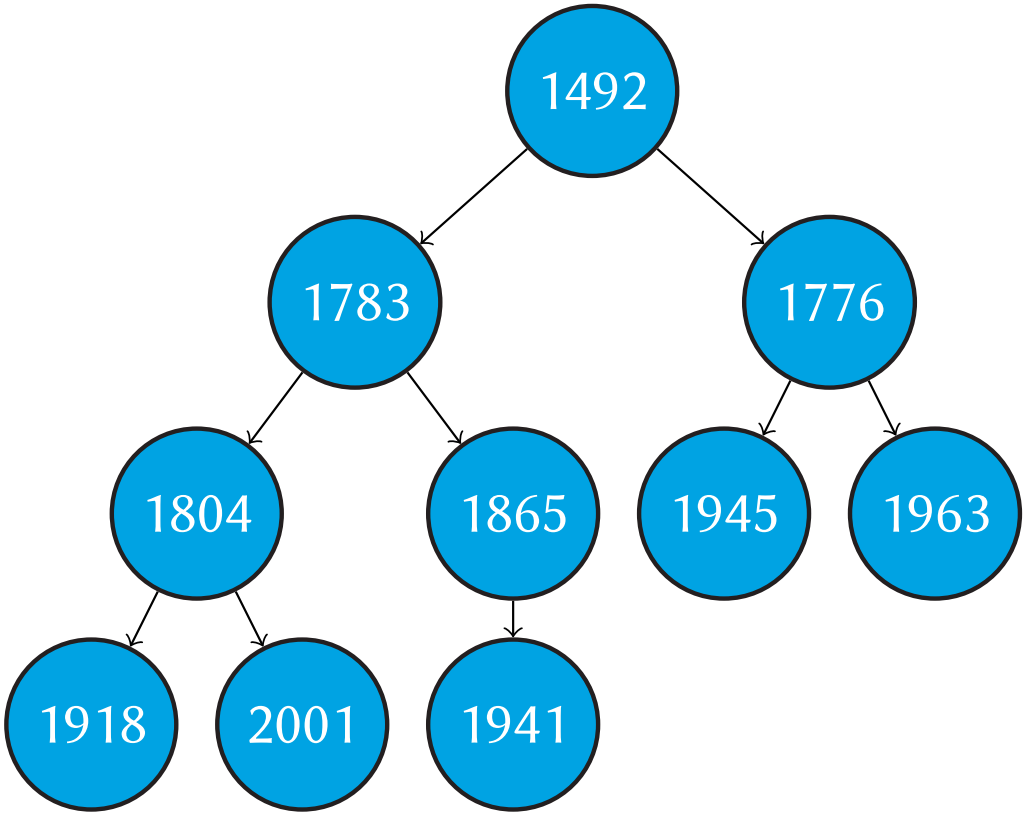
But, it actually is more efficient to store it in an array such that:

The left child of k sits in position $2k$ and the right child in $2k + 1$.

The parent of k is in position $\lfloor k/2 \rfloor$.

Binary Heaps

Our example heap looks like this:



(heap shown as tree)

1	1492
2	1783
3	1776
4	1804
5	1865
6	1945
7	1963
8	1918
9	2001
10	1941

(heap as array)

Can we represent any binary tree in an array ?

This representation is only efficient if the tree is sparse (the number of nodes $n < 2^h$).

Why?

Can we represent any binary tree in an array ?

This representation is only efficient if the tree is sparse (the number of nodes $n < 2^h$).

Why?

All missing internal nodes still take up space in our structure.

That's why heaps need to be as balanced/full at each level as possible.

The array-based representation is also not as flexible to arbitrary modifications as a pointer-based tree

Creating heaps

We create heaps incrementally:

1. Put a new element in the left-most open spot in the array
2. If the new element is less than its parent, swap the positions.
3. Keep checking if the new element is less than the parent, swapping until it's in a position where the parent is less than the new element.

Runtime of creating a heap

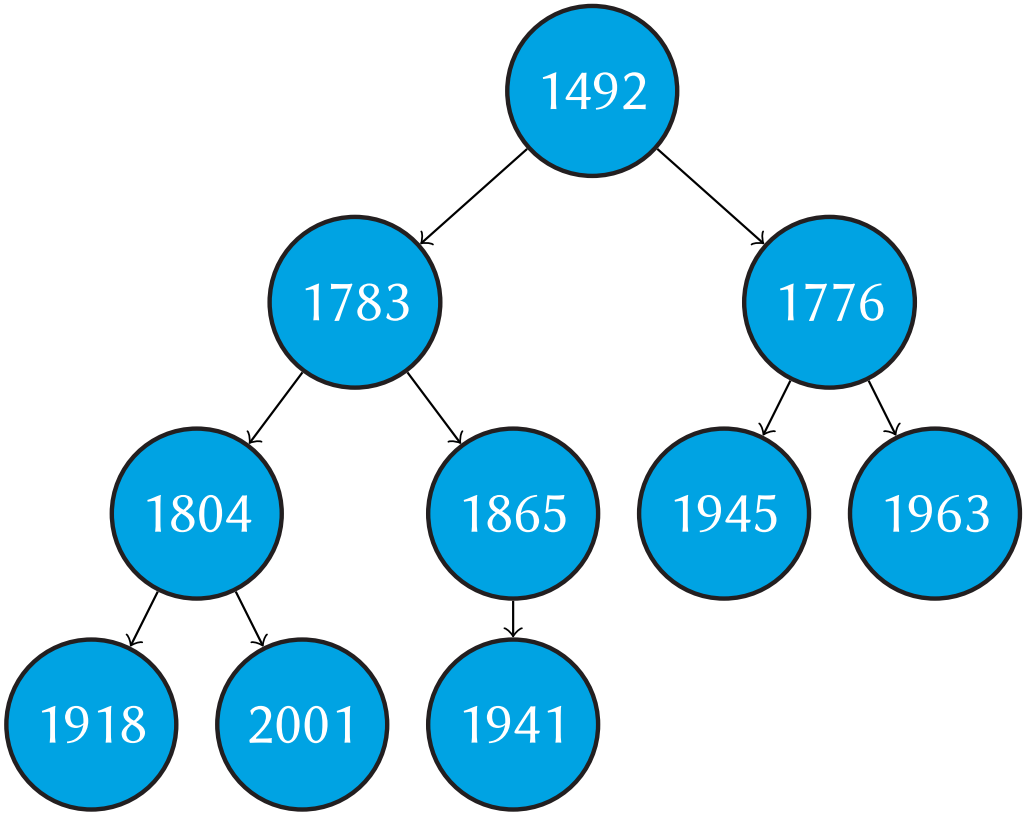
How long does it take to create a heap?

Well, the height h of the n element tree is always $\lfloor \lg n \rfloor$, which means inserting one element takes at max $\lg n$ steps.

Doing n such insertions takes $\Theta(n \log n)$, since the last $n/2$ insertions require $O(\log n)$ time each.

Binary Heaps

Our example heap looks like this:



(heap shown as tree)

1	1492
2	1783
3	1776
4	1804
5	1865
6	1945
7	1963
8	1918
9	2001
10	1941

(heap as array)

Putting some pseudo-code to it

HeapInsert(Heap h, Item val)

```
1  h.lastElem = h.lastElem + 1
2  h.elements[lastElem] = val
3  BubbleUp(h, h.lastElem)
```

Putting some pseudo-code to it

```
HeapInsert(Heap h, Item val)
1   h.lastElem = h.lastElem + 1
2   h.elements[lastElem] = val
3   BubbleUp(h, h.lastElem)

BubbleUp(Heap h, int whichElem)
1   if whichElem == 0
2         return
3   parentElem = whichElem / 2
4   if (h.elements[parentElem] < h.elements[whichElem])
5         swap(h, parentElem, whichElem)
6         BubbleUp(h, parentElem)
```

Improving on Bubble-Up

Robert Floyd found a better way to build a heap, using a merge procedure called ***heapify***.

Given two heaps and a fresh element, they can be merged into one by making the new one the root and bubbling it down:

BuildHeap(A)

```
1   n = |A|
2   for i = ⌊n/2⌋ : 1
3         Heapify(A,1)
```

HEAPIFY(HEAP H, INT STARTELEM, INT MAX)

```
1   while startElem <= max
2         index = startElem
3         leftChild = 2*startElem;
4         rightChild = leftChild + 1;
5         if (leftChild < max AND heap[leftChild] > heap[index])
6               index = leftChild;
7         if (rightChild < max AND heap[rightChild] > heap[index])
8               index = rightChild;
9         if (index == startElem)
10              return;
11        swap(heap,startElem, index);
12        startElem = index;
```

HEAPIFY performs better than $O(n \log n)$, because most of the heaps we merge are small.

In a full binary tree on n nodes, there are at most $\lfloor n/2^{h+1} \rfloor$ nodes of height h, so the cost of building a heap is:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} h/2^h \right)$$

Since this sum is not quite a geometric series, we can't apply the usual identity to get the sum. But it should be clear that the series converges.

Heap Summary

- Creating a heap: $O(n \lg n)$ (using heapify)

Heap Summary

- Creating a heap: $O(n \lg n)$ (using heapify)
- Inserting an element into a heap: $O(n \lg n)$

Heap Summary

- Creating a heap: $O(n \lg n)$ (using heapify)
- Inserting an element into a heap: $O(n \lg n)$
- Removing an element from a heap:

Heapsort

Okay, we know we can easily create a heap.
What can we do with this heap?
Heapify can be used to construct a heap, using the observation that an isolated element forms a heap of size 1.

HEAPSORT(ARRAY A)

```
1   BUILD-HEAP(A)
2   for i = n to 1 do
3         swap(A[1],A[i])
4         n = n - 1
5         Heapify(A,1)
```

Exchanging the maximum element with the last element and calling heapify repeatedly gives an $O(n \lg n)$ sorting algorithm.
Why is it not $O(n)$?

Subsection 3

Priority Queue

Priority Queues

Priority queues are data structures which provide extra flexibility over sorting.

Priority Queue Operations

The basic priority queue supports three primary operations:

Insert(Q, x) Given an item x with value v, insert it into the priority queue Q.

Find-Minimum(Q) or Find-Maximum(Q) Return a pointer to the item whose key value is smaller (larger) than any other key in the priority queue Q.

Delete-Minimum(Q) or Delete-Maximum(Q) Remove the item from the priority queue Q whose key is minimum (maximum).

Each of these operations can be easily supported using heaps or balanced binary trees in $O(\lg n)$.

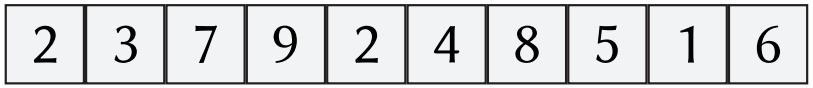
Quicksort Introduction

- Quicksort is the fastest *internal* sorting algorithm.
- Key idea is ***partitioning***
  - **Example:** ***pivot*** about 10
  - **Before:** 17 12 6 19 23 8 5 10
  - **After:** 6 8 5 10 23 19 12 17
- All the elements less than the pivot are to the left of the pivot; all elements greater are to the right.
- The pivot is in the middle.

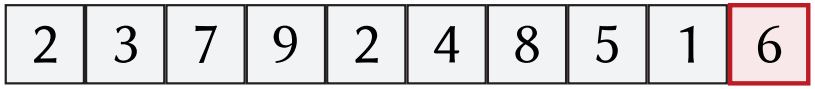
Subsection 1

Partitioning

Partitioning: Graphically



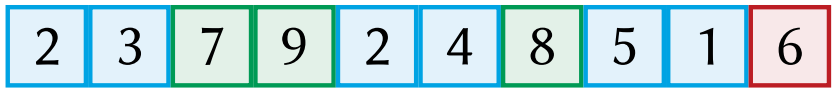
Partitioning: Graphically



Pivot = 6

(Why? Because. It doesn't really matter.)

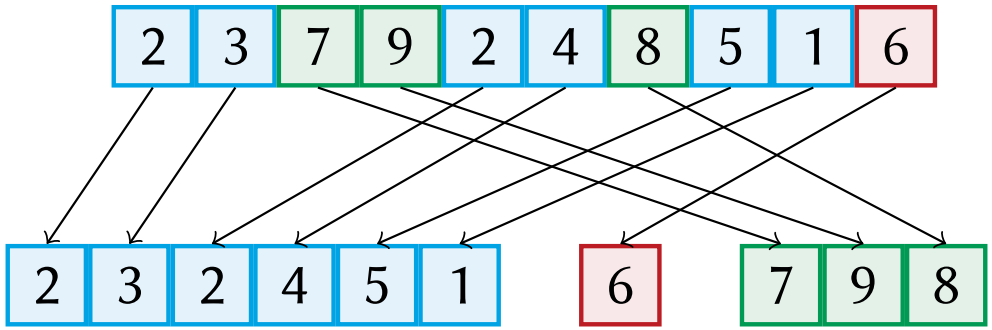
Partitioning: Graphically



Pivot = 6

(Why? Because. It doesn't really matter.)

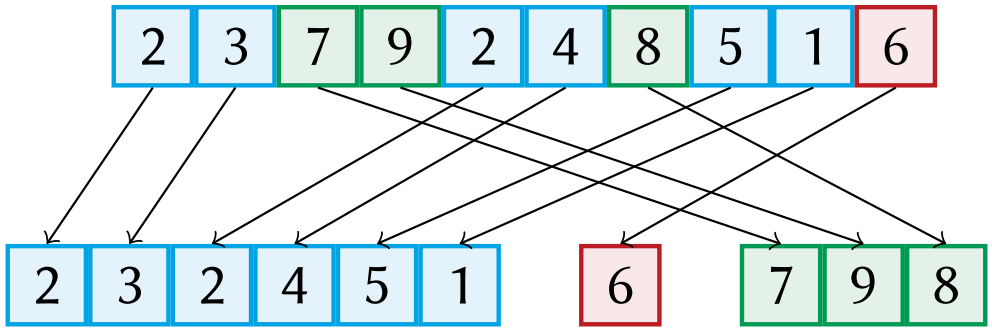
Partitioning: Graphically



Pivot = 6

(Why? Because. It doesn't really matter.)

Partitioning: Graphically

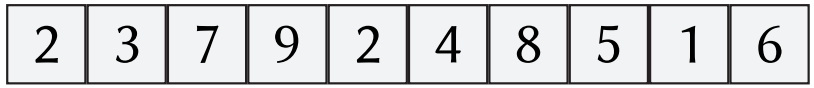


- All the elements less than the pivot are to the left of the pivot; all elements greater are to the right.
  - The pivot is in the middle.
- Now that we've partitioned around this pivot, it is in place in the final order!

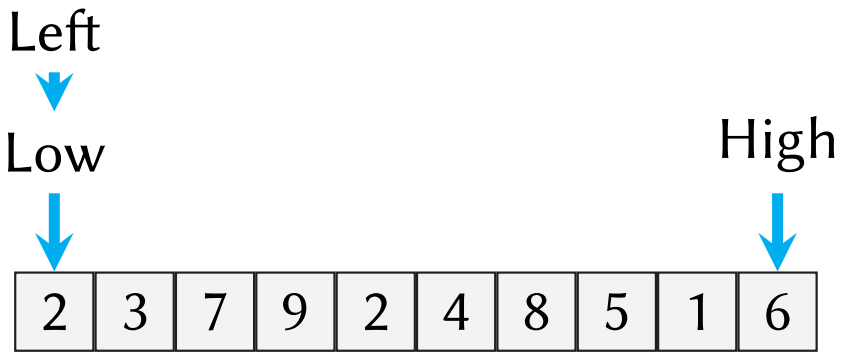
Notes on partitioning

- The partition can be done in one linear scan
- We consider 3 sections of the array:
  - Less than pivot
  - Greater than pivot
  - Unexplored

Partitioning: The Algorithm, Graphically

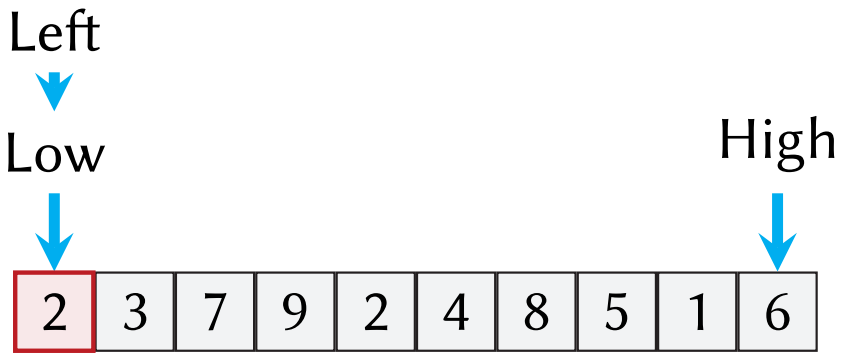


Partitioning: The Algorithm, Graphically



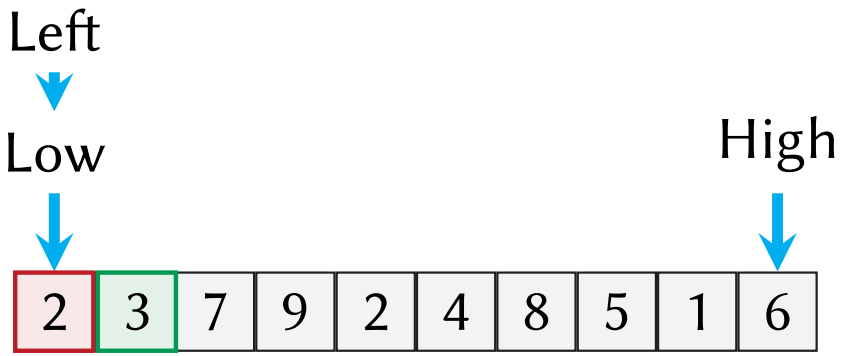
Start looking at the entire array; place some markers

Partitioning: The Algorithm, Graphically



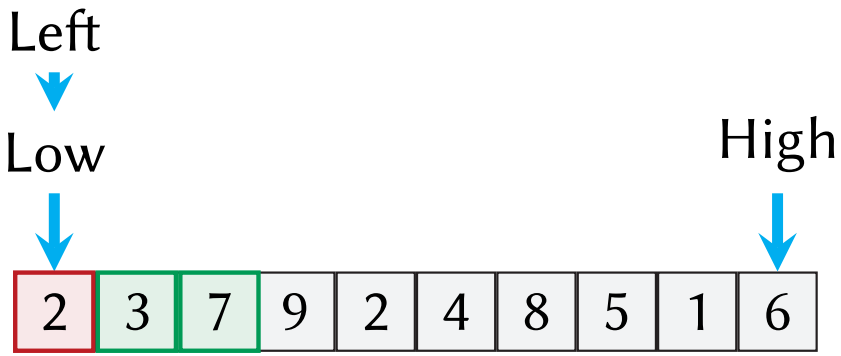
Decide that the pivot is whatever Low is pointing at

Partitioning: The Algorithm, Graphically



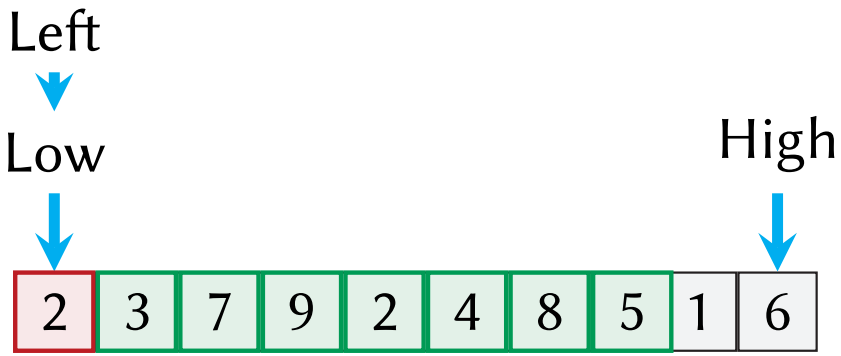
Start comparing the pivot to every other element in the array

Partitioning: The Algorithm, Graphically



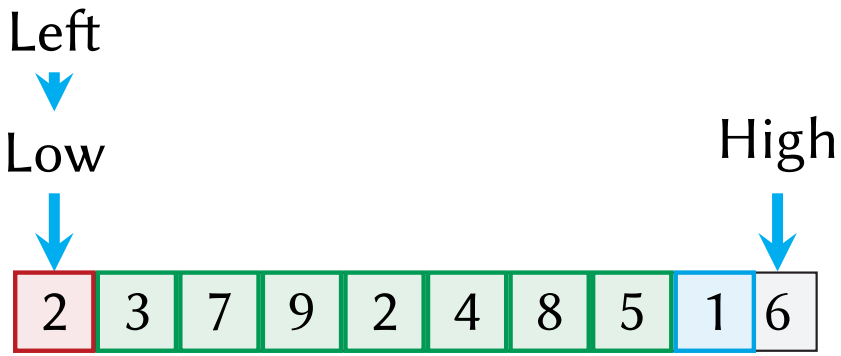
Element is bigger than pivot, so keep going.

Partitioning: The Algorithm, Graphically



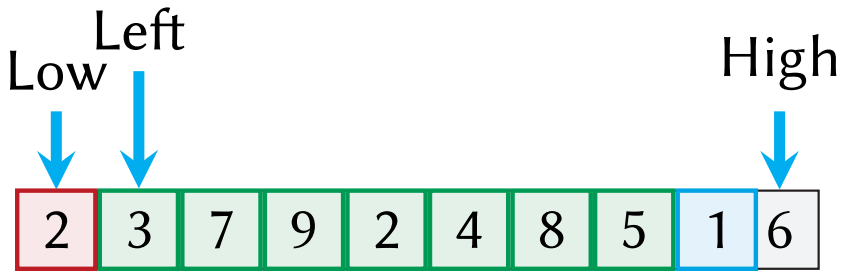
All these elements are bigger, so let's jump ahead.

Partitioning: The Algorithm, Graphically



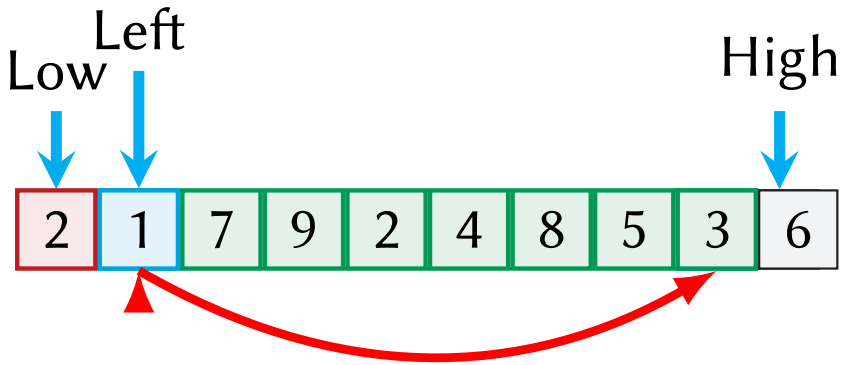
We've found an element smaller than the pivot, so do something.

Partitioning: The Algorithm, Graphically



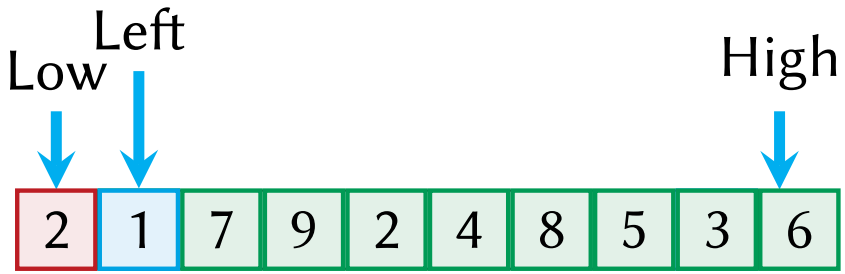
Move the left marker up

Partitioning: The Algorithm, Graphically



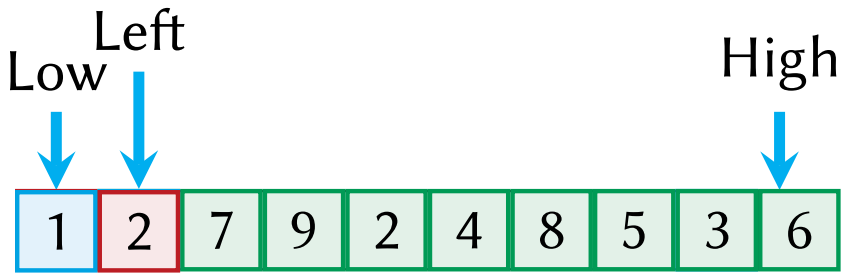
Swap the current (smaller) element with the element that Left is pointing at.

Partitioning: The Algorithm, Graphically



Last element is bigger than the pivot, so ignore it.

Partitioning: The Algorithm, Graphically



To finish up:

Swap the Left with the original low, which ensures the pivot is in the correct place.

Think and convince yourself that if there were more than one element on the left hand side, swapping the pivot with Left would still leave you with a properly partitioned array.

Why partition?

Partitioning consists of at most n swaps, so it takes time linear of n.
What does it give us?

Why partition?

Partitioning consists of at most n swaps, so it takes time linear of n.
What does it give us?

1. The pivot element ends up in the position it retains in the final sorted order.

Why partition?

Partitioning consists of at most n swaps, so it takes time linear of n.
What does it give us?

1. The pivot element ends up in the position it retains in the final sorted order.
2. After a partitioning, no element flops to the other side of the pivot in the final sorted order.

Why partition?

Partitioning consists of at most n swaps, so it takes time linear of n.
What does it give us?

1. The pivot element ends up in the position it retains in the final sorted order.
2. After a partitioning, no element flops to the other side of the pivot in the final sorted order.

At this point, we can sort the elements to the left of the pivot and the right of the pivot independently, giving us a recursive sorting algorithm!

Subsection 2

The Quicksort Algorithm

Quicksort Pseudocode

```
1 Sort(A):
2   Quicksort(A, 1, n)
3 Quicksort(A, low, high):
4   if (low < high)
5     pivot-location = Partition(A,low,high)
6     Quicksort(A,low, pivot-location - 1)
7     Quicksort(A, pivot-location+1, high)
```

Quicksort C Implementation

```
1 quicksort(int s[], int low, int high){
2   int p; // index of partition
3
4   if ((high-low) > 0){
5     p = partition(s, low, high);
6     quicksort(s, low, p-1);
7     quicksort(s, p+1, h);
8   }
9 }
```

Partition Pseudocode

```
1 Partition(A,low,high)
2   pivot = A[low]
3   leftwall = low
4   for i = low+1 to high
5     if (A[i] < pivot) then
6       leftwall = leftwall+1
7       swap(A[i],A[leftwall])
8   swap(A[low],A[leftwall])
```

Partition C Implementation

```
1  int partition(int s[], int low, int high){
2    int i; // counter
3    int p; // pivot element
4    int firsthigh; // divider partition for pivot
5
6    p = h;
7    firsthigh = low;
8    for (i=1; i<high; i++){
9      if (s[i] < s[p]){
10       swap(&s[j], &s[firsthigh]);
11       firsthigh++;
12     }
13   swap(&s[p], &s[firsthigh]);
14
15   return(firsthigh);
16   }
17 }
```

Quicksort Animation

Subsection 3

Analysis: Best Case

Quicksort Analysis: Best Case

We know it's correct, because we can see that each element ends in the correct position.

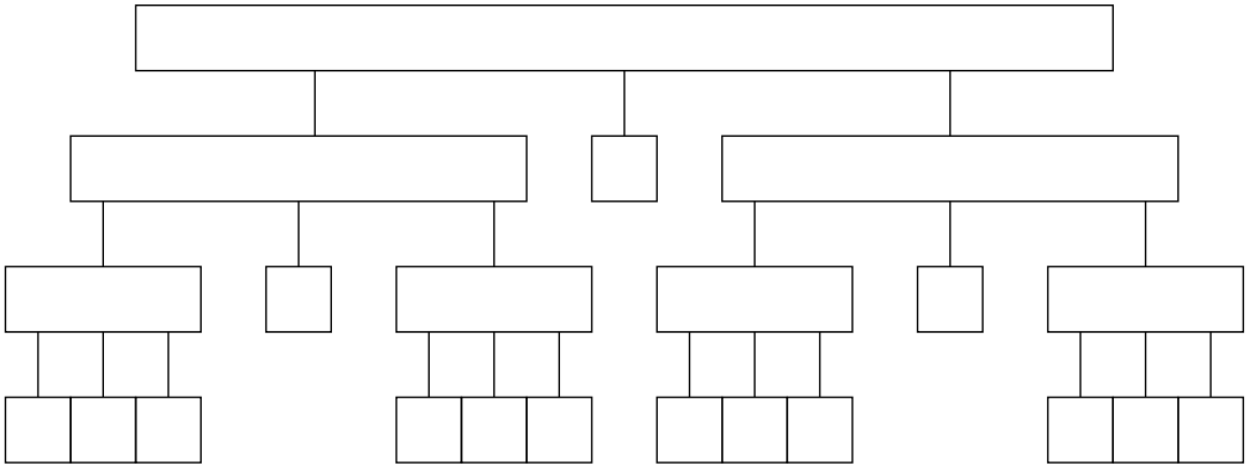
How long does it take?

The best case for divide-and-conquer algorithms comes when we split the input as evenly as possible. Thus in the best case, each subproblem is of size $n/2$.

The partition step on each subproblem is linear in its input size.

Thus the total effort in partitioning the 2^k problems of size $n/2^k$ is $O(n)$.

Best Case Recursion Tree



The total partitioning on each level is $O(n)$, and it take $\lg n$ levels of perfect partitions to get to single element subproblems.

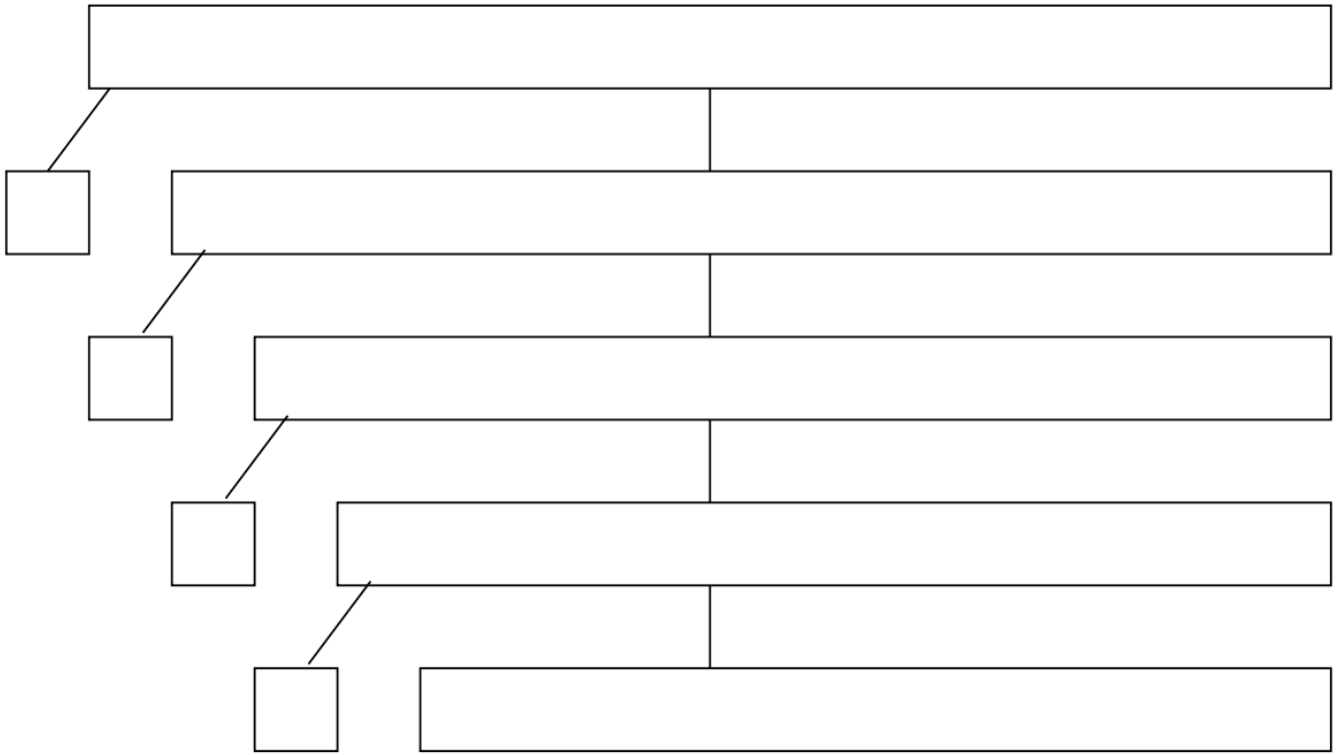
When we are down to single elements, the problems are sorted.

Thus the total time in the best case is $O(n \lg n)$.

Subsection 4

Analysis: Worst Case

Worst Case



Suppose instead our pivot element splits the array as unequally as possible.

Instead of $n/2$ elements in the smaller half, we get zero, meaning that the pivot element is the biggest or smallest element in the array.

Now we have $n - 1$ levels, instead of $\lg n$, for a worst case time of $\Theta(n^2)$, since the first $n/2$ levels each have $\geq n/2$ elements to partition.

To justify its name, Quicksort had better be good in the average case. Showing this requires some intricate analysis.

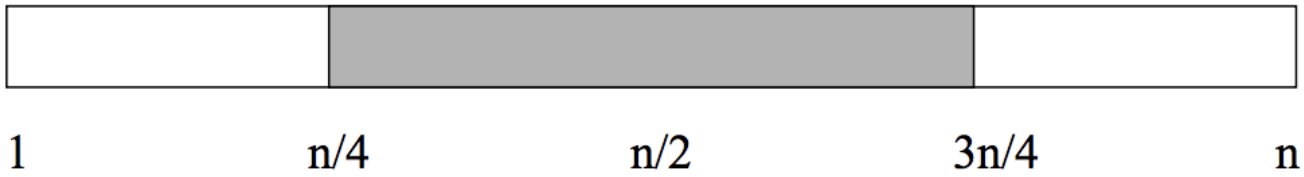
One take away: If you break a job into pieces, you make more progress by making the pieces of equal size!

Subsection 5

Intuition

Intuition: Average case for quicksort

Suppose we pick the pivot element at random in an array of n values:



Half the time, the pivot element will be from the middle half of the sorted array.

Whenever the pivot element is from positions $n/4$ to $3n/4$, the larger remaining subarray contains at most $3n/4$ elements.

How many good partitions?

If we assume that the pivot element is always in this range, what is the maximum number of partitions we need to get from n elements down to 1 element?

$$\left(\frac{3}{4}\right)^{l} \cdot n = 1 \tag{1}$$

$$\rightarrow n = \left(\frac{4}{3}\right)^{l} \tag{2}$$

$$\lg n = \lg\left((4/3)^{l}\right) \tag{3}$$

$$\lg n = l \cdot \lg(4/3) \tag{4}$$

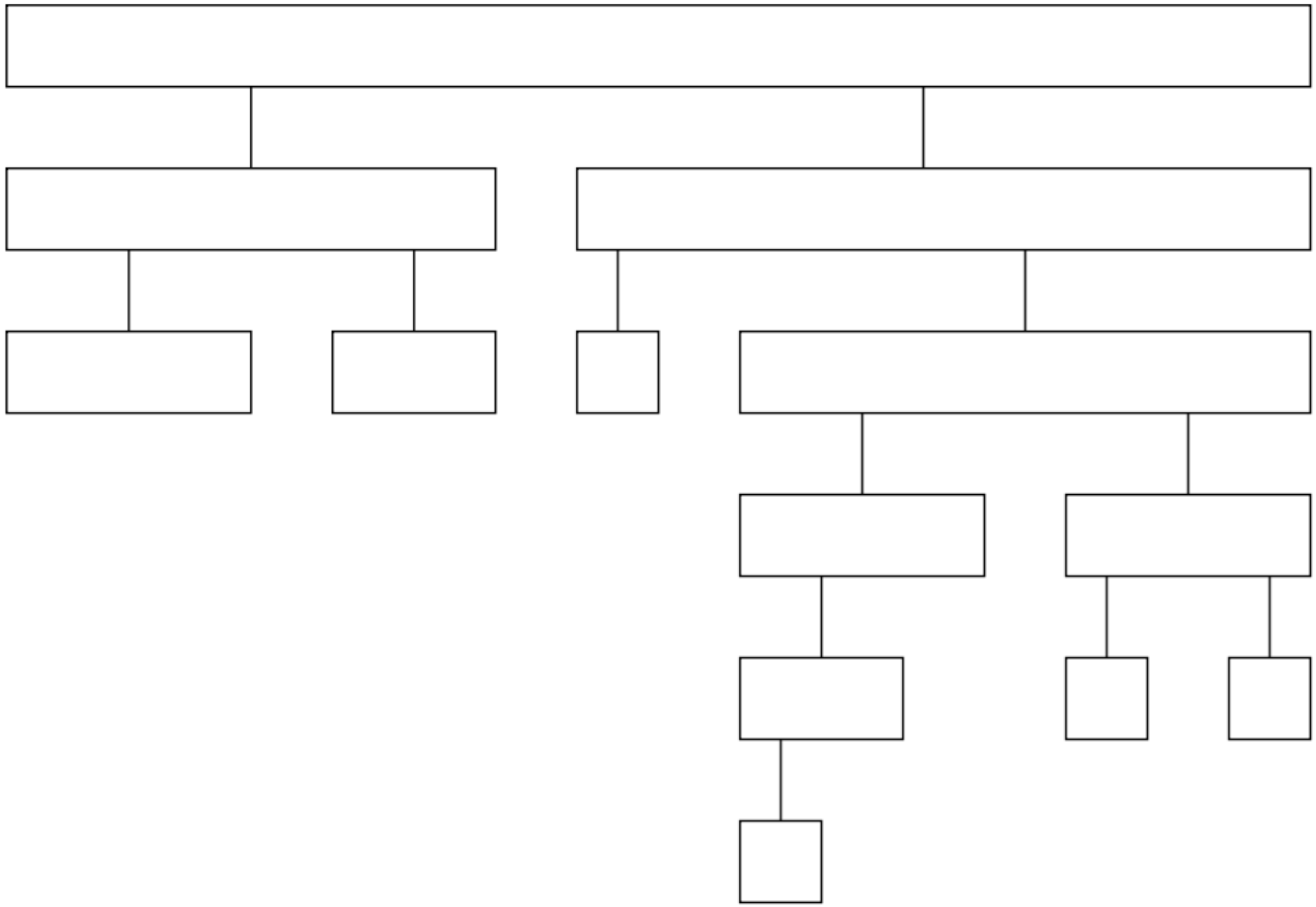
Therefore $l = \lg(4/3) \cdot \lg(n) < 2 \lg n$ good partitions suffice.

How many bad partitions?

If we pick an arbitrary element as pivot, how often will it generate a decent partition?

Since any number ranked between $n/4$ and $3n/4$ would make a decent pivot, we get one half the time on average.

If we need $2 \lg n$ levels of decent partitions to finish the job, and half of random partitions are decent, then on average the recursion tree to quicksort the array has $\approx 4 \lg n$ levels.



Since $O(n)$ work is done partitioning on each level, the average time is $O(n \lg n)$.

Subsection 6

Average case analysis: Solve the recurrence


2 3 2 4 5 1
6
7 9 8

Average Case analysis of Quicksort

To do a precise average-case analysis of quicksort, we formulate a recurrence given the exact expected time $T(n)$[1]:

$$T(n) = (T(p-1) + T(n-p)) + (n-1)$$

$$T(n) = \sum_{p=1}^{n} \frac{1}{n} (T(p-1) + T(n-p)) + (n-1)$$

Each possible pivot p is selected with equal probability.
The number of comparisons needed to do the partition is $n-1$.

[1] Equivalent to (8.3) in CLRS

Short aside: Useful Harmonic Numbers H_n

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n$$

Solving the Recurrence

Start with the recurrence

$$T(n) = \left(\sum_{p=1}^{n} \frac{1}{n} \left(T(p-1) + T(n-p) \right) \right) + n - 1 \tag{5}$$

(8)

Solving the Recurrence

Algebra. When you consider how the sum expands, you see how you can pull that 2 out.

$$T(n) = \left(\sum_{p=1}^{n} \frac{1}{n} \left(T(p-1) + T(n-p) \right) \right) + n - 1 \tag{5}$$

$$T(n) = \frac{2}{n} \left(\sum_{p=1}^{n} T(p-1) \right) + n - 1 \tag{6}$$

(8)

Solving the Recurrence

Multiply by n

$$T(n) = \left(\sum_{p=1}^{n} \frac{1}{n} \left(T(p-1) + T(n-p) \right) \right) + n - 1 \tag{5}$$

$$T(n) = \frac{2}{n} \left(\sum_{p=1}^{n} T(p-1) \right) + n - 1 \tag{6}$$

$$nT(n) = 2 \left(\sum_{p=1}^{n} T(p-1) \right) + n(n-1) \tag{7}$$

$$\tag{8}$$

Solving the Recurrence

Apply to $n-1$

$$T(n) = \left(\sum_{p=1}^{n} \frac{1}{n} \left(T(p-1) + T(n-p) \right) \right) + n - 1 \tag{5}$$

$$T(n) = \frac{2}{n} \left(\sum_{p=1}^{n} T(p-1) \right) + n - 1 \tag{6}$$

$$nT(n) = 2 \left(\sum_{p=1}^{n} T(p-1) \right) + n(n-1) \tag{7}$$

$$(n-1)T(n-1) = 2 \left(\sum_{p=1}^{n-1} T(p-1) \right) + (n-1)(n-2) \tag{8}$$

...continuing to solve the recurrence

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1) \tag{9}$$

(14)

...continuing to solve the recurrence

Rearranging terms

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1) \tag{9}$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \tag{10}$$

(14)

...continuing to solve the recurrence

Substituting $a_n = A(n)/(n+1)$ gives us

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1) \tag{9}$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \tag{10}$$

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} \tag{11}$$

(14)

...continuing to solve the recurrence

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1) \tag{9}$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \tag{10}$$

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} \tag{11}$$

$$a_n \approx 2\sum_{i=1}^{n} \frac{1}{(i+1)} \approx 2\ln n \tag{12}$$

(14)

...continuing to solve the recurrence

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1) \tag{9}$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \tag{10}$$

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} \tag{11}$$

$$a_n \approx 2\sum_{i=1}^{n} \frac{1}{(i+1)} \approx 2\ln n \tag{12}$$

$$A(n) = (n+1)a_n \approx 2(n+1)\ln n \approx 1.38n\lg n \tag{13}$$

$$\tag{14}$$

...continuing to solve the recurrence

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1) \tag{9}$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \tag{10}$$

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} \tag{11}$$

$$a_n \approx 2\sum_{i=1}^{n} \frac{1}{(i+1)} \approx 2\ln n \tag{12}$$

$$A(n) = (n+1)a_n \approx 2(n+1)\ln n \approx 1.38n\lg n \tag{13}$$

$$A(n) = \Theta(n\lg n) \tag{14}$$

Subsection 7

Picking a better pivot

Pick a better pivot

Having the worst case occur when they are sorted or almost sorted is ***very bad***, since that is likely to be the case in certain applications.

To eliminate this problem, we have three options to pick a better pivot:

1. Use the middle element of the subarray as pivot.
2. Use a *random* element of the array as the pivot.
3. Take the median of three elements (first, last, middle) as the pivot. (Why should we use median instead of the mean?)

Whichever of these three rules we use, the worst case remains $O(n^2)$

Subsection 8

Randomizing Quicksort

Is Quicksort really faster than Heapsort?

When Quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort.

This is because the operations in the innermost loop are simpler.

The difference between heapsort and quicksort is limited to a multiplicative constant factor. This means the details of how you implement each algorithm can make a big difference.

Randomized Quicksort

Suppose you are writing a sorting program, to run on data given to you by your worst enemy. Quicksort is good on average, but bad on certain worst-case instances.

If you used Quicksort, what kind of data would your enemy give you to run it on? Exactly the worst-case instance, to make you look bad.

But instead of picking the median of three or the first element as pivot, suppose you picked the pivot element at random.

Now your enemy cannot design a worst-case instance to give to you, because no matter which data they give you, you would have the same probability of picking a good pivot!

Randomized Guarantees

Randomization is a very important and useful idea. By either picking a random pivot or scrambling the permutation before sorting it, we can say:

"With high probability, randomized quicksort runs in $\Theta(n \lg n)$time."

Where before, all we could say is:

"If you give me random input data, quicksort runs in expected $\Theta(n \lg n)$ time."

Importance of Randomization

Since the time bound how does not depend upon your input distribution, this means that unless we are extremely unlucky (as opposed to ill prepared or unpopular) we will certainly get good performance.

Randomization is a general tool to improve algorithms with **bad worst-case but good average-case** complexity.

The worst-case is still there, but we almost certainly won't see it.

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

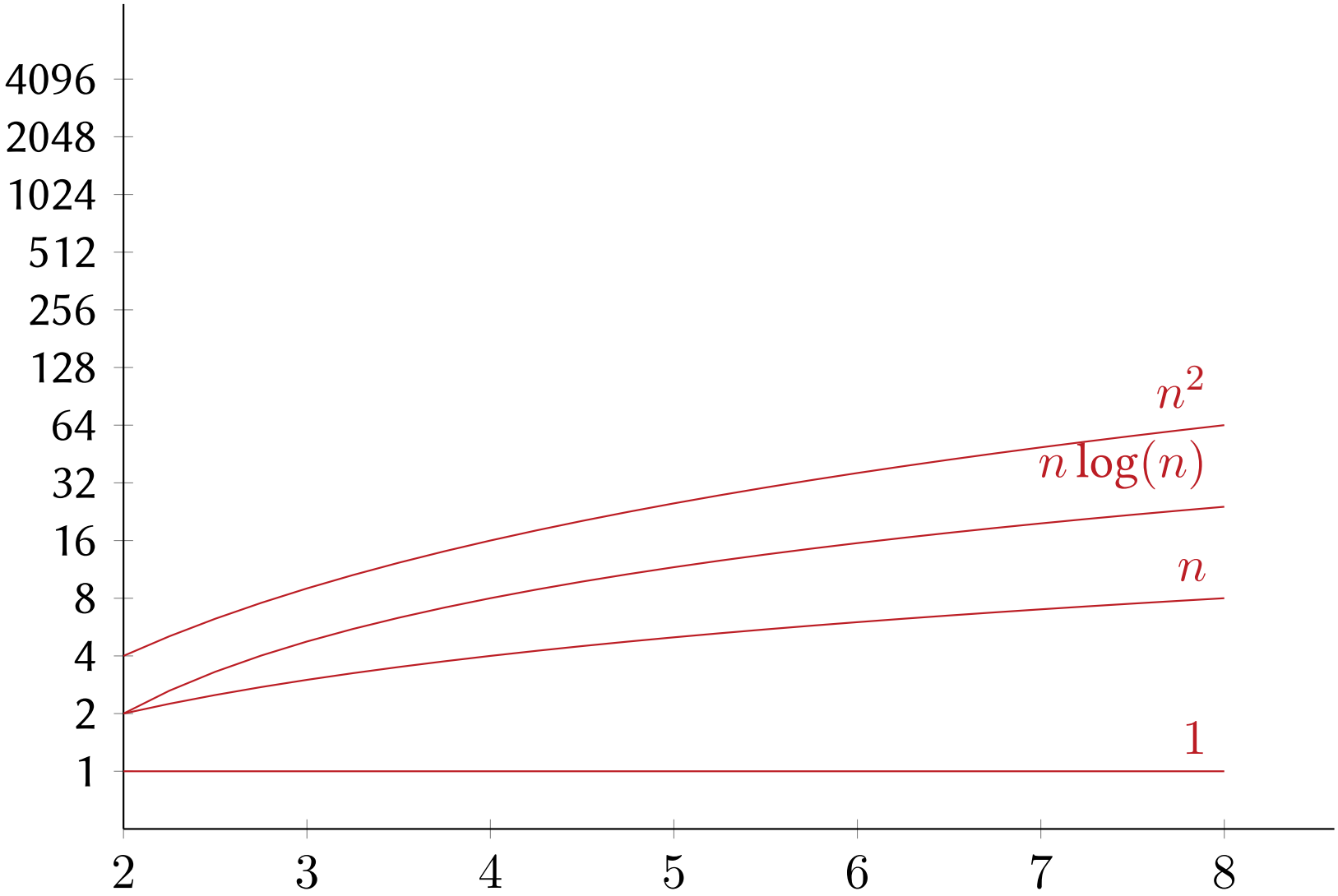
Table: Number of operations for a given input size n

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 4

Linear time sorting

A sorting problem

I have a bunch of playing cards to sort.

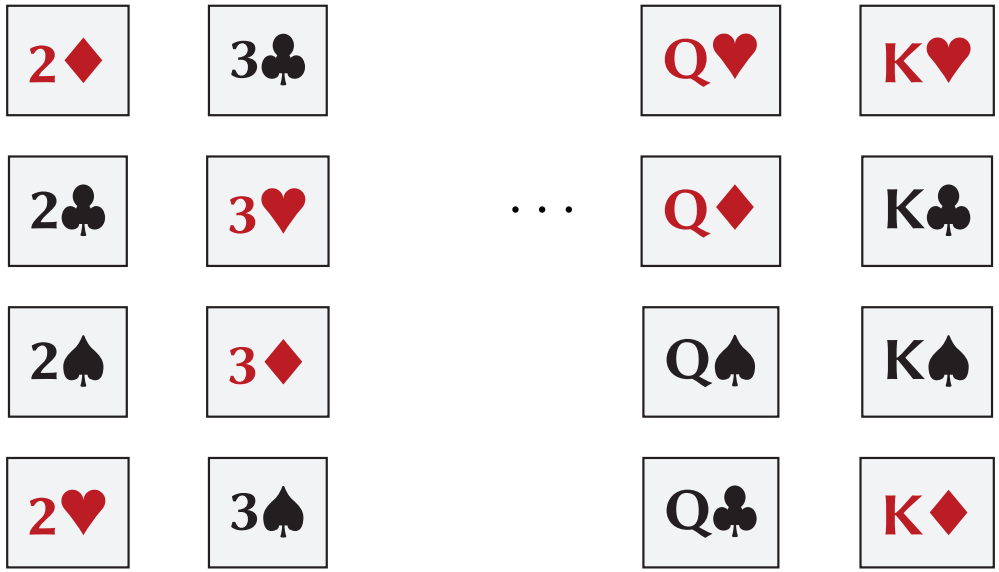
I want them sorted numerically ascending, and suits in the order ♥♦♣♠

How do I do it?

2♥	5♦	K♠	Q♣	A♥	6♣	4♠	J♥	K♦	6♠	···

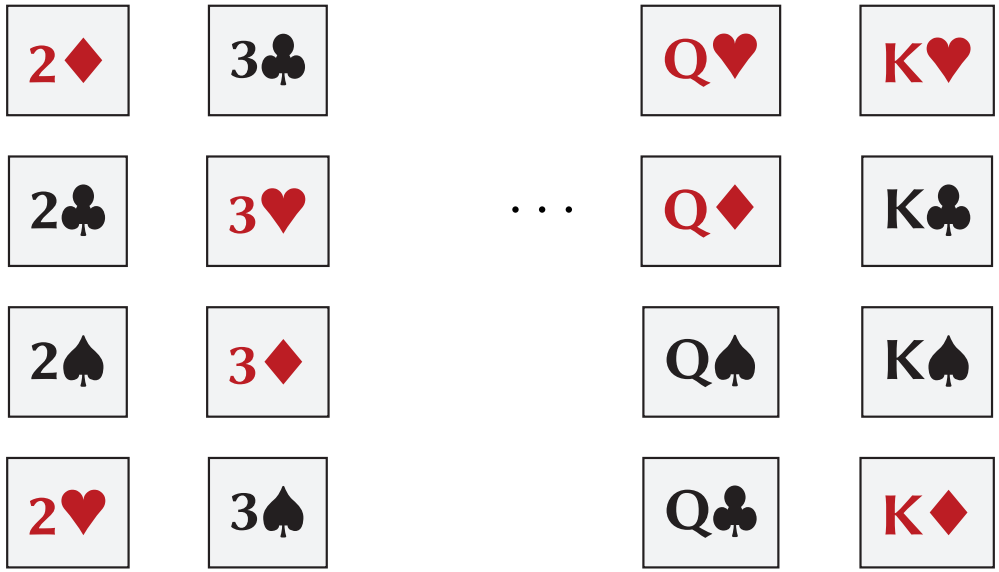
A sorting problem

One approach:



A sorting problem

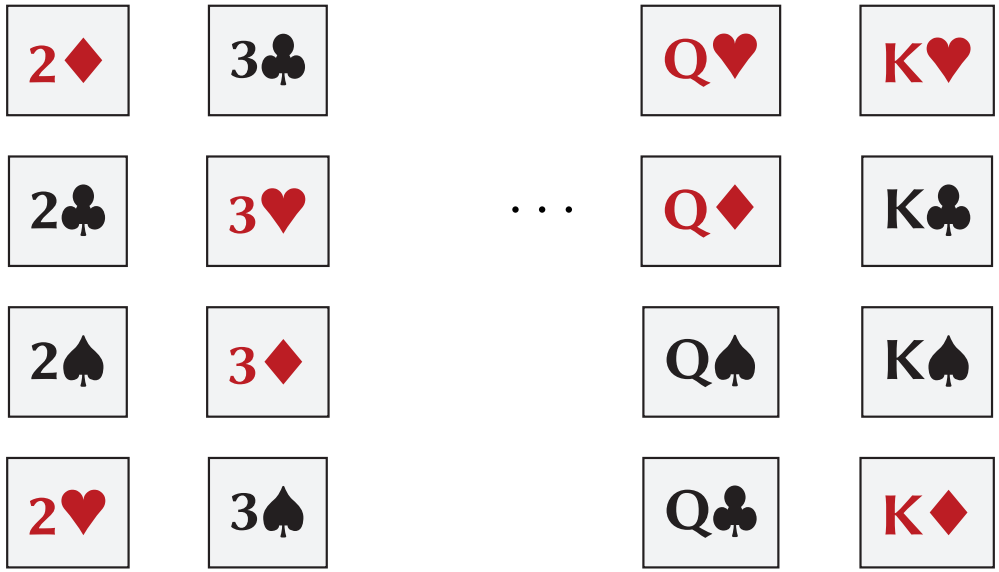
One approach:



To summarize:

A sorting problem

One approach:

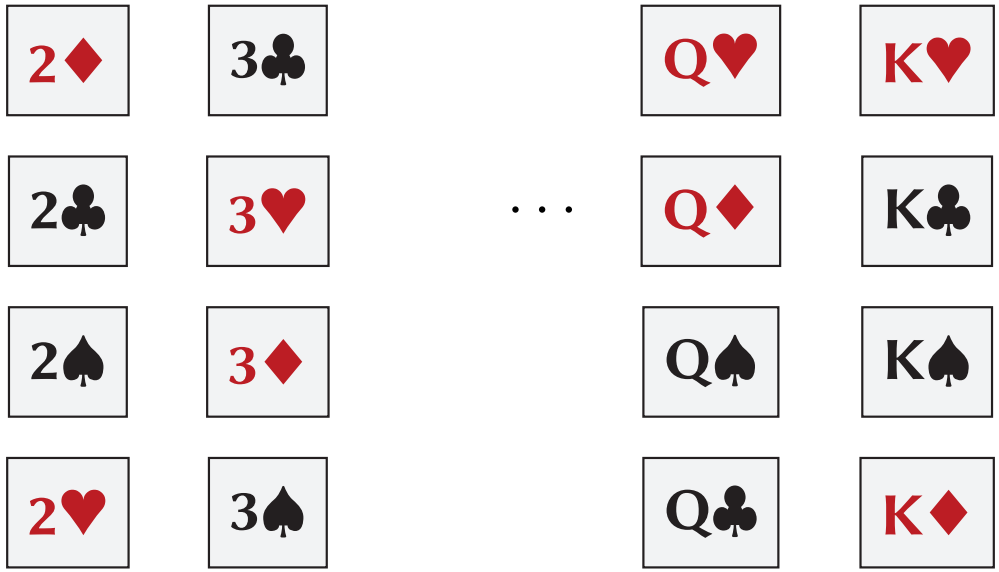


To summarize:

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

A sorting problem

One approach:



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 4

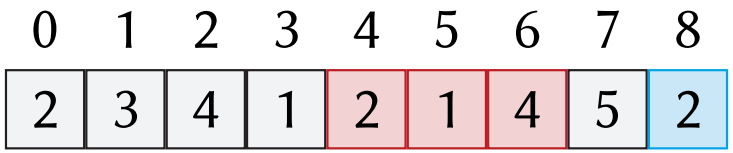
Linear time sorting

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:



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

`cts[`i`] = cts[`i`] + 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

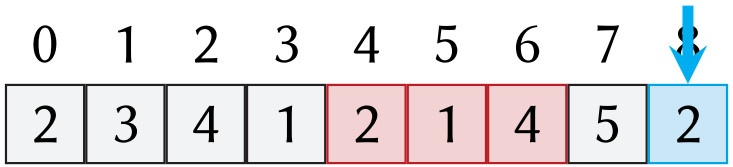
Output Array:

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

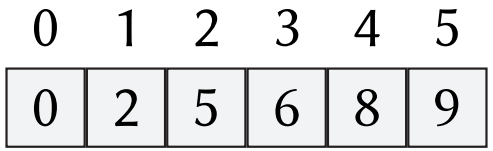
Now moving elements into the result array

Start at the end of the array

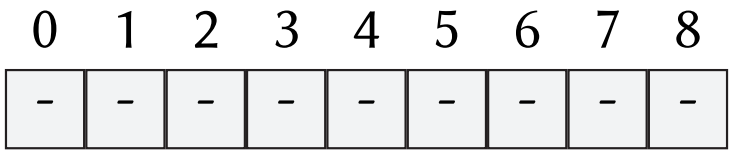
Input array:



Counts:



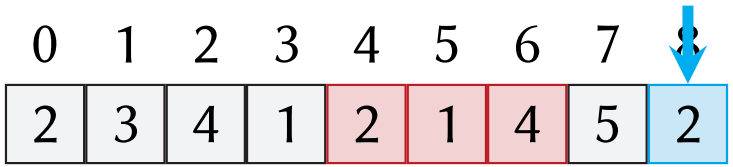
Output Array:



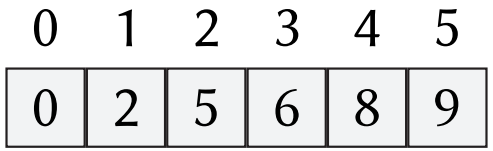
Now moving elements into the result array

Look at the value of the element

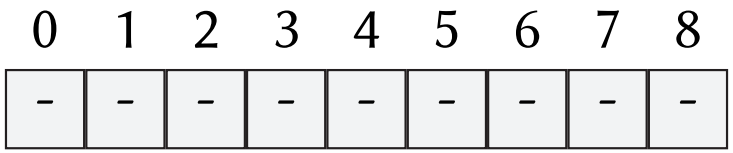
Input array:



Counts:



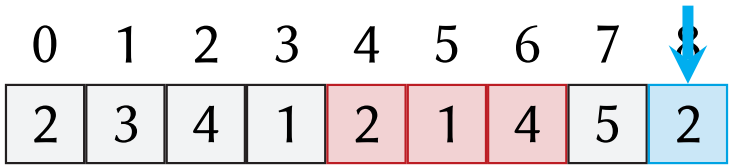
Output Array:



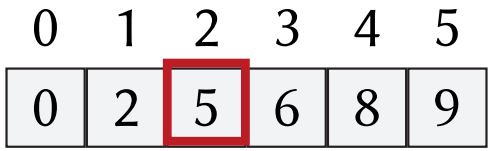
Now moving elements into the result array

Find the value in the counts array

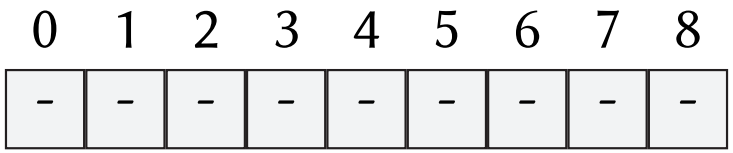
Input array:



Counts:



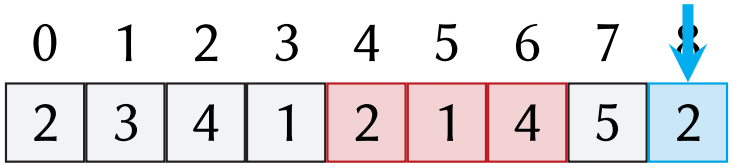
Output Array:



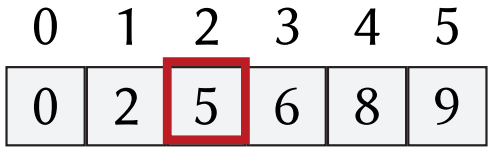
Now moving elements into the result array

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

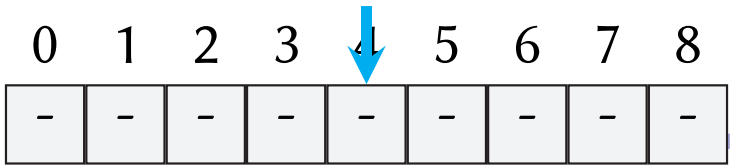
Input array:



Counts:



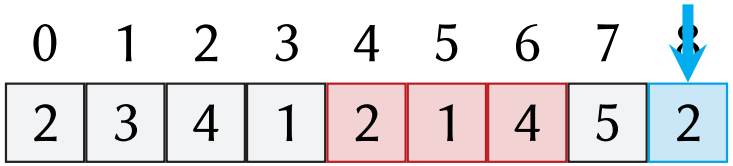
Output Array:



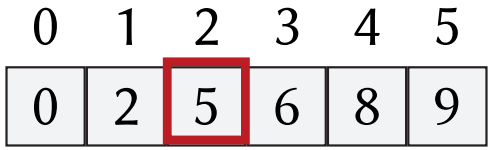
Now moving elements into the result array

Put the element into the output array.

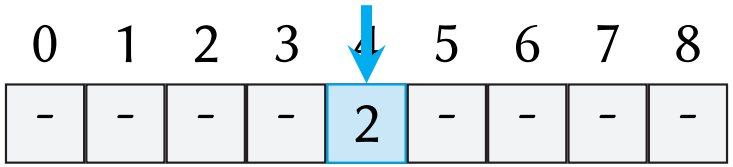
Input array:



Counts:



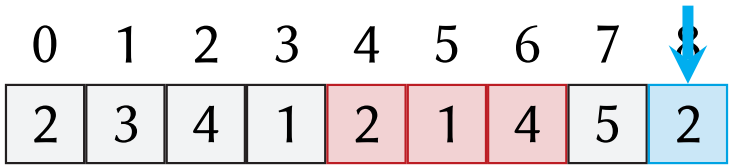
Output Array:



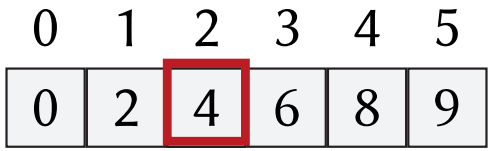
Now moving elements into the result array

Reduce the counts array element.

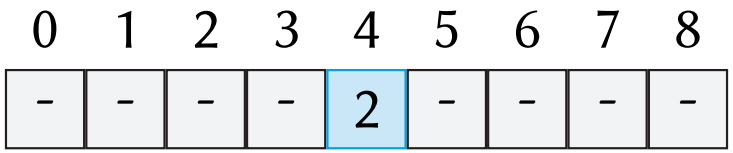
Input array:



Counts:



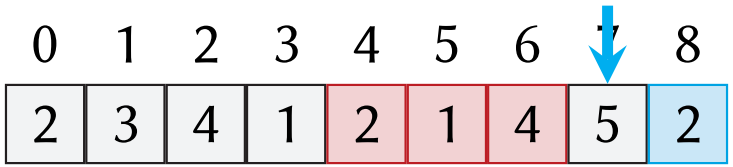
Output Array:



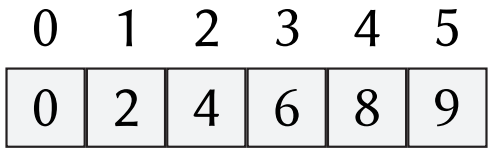
Now moving elements into the result array

While there are more elements in the array, repeat.

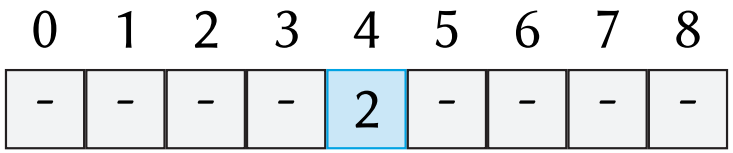
Input array:



Counts:



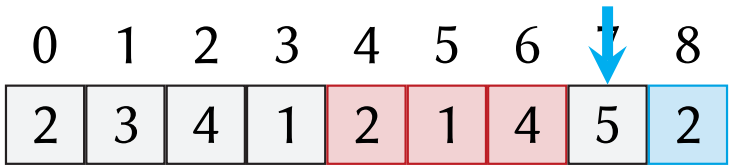
Output Array:



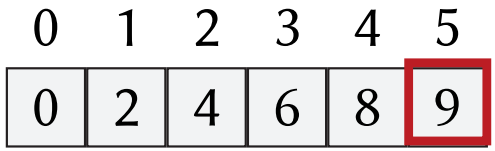
Now moving elements into the result array

While there are more elements in the array, repeat.

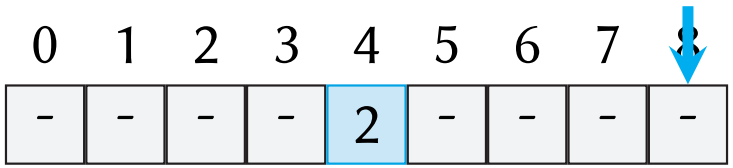
Input array:



Counts:



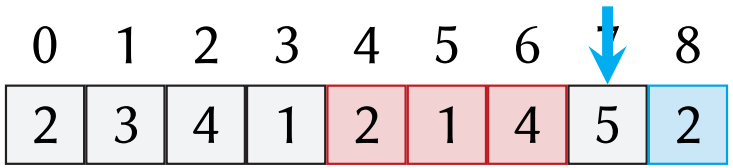
Output Array:



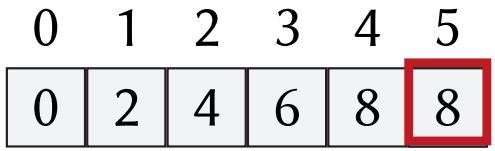
Now moving elements into the result array

While there are more elements in the array, repeat.

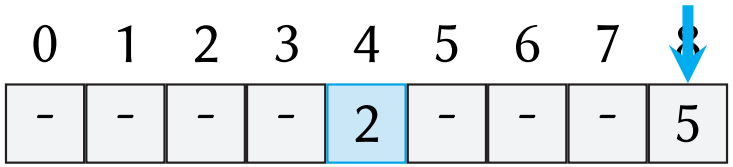
Input array:



Counts:



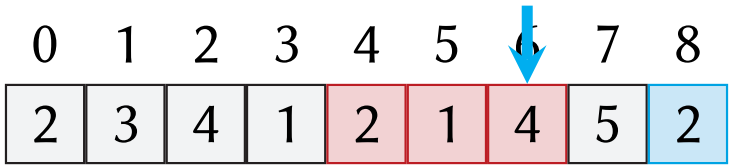
Output Array:



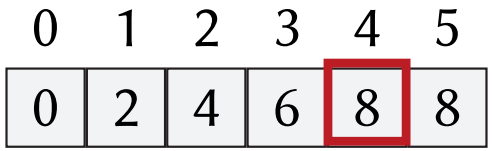
Now moving elements into the result array

Dealing with element 6

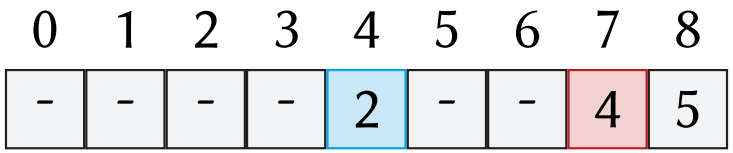
Input array:



Counts:



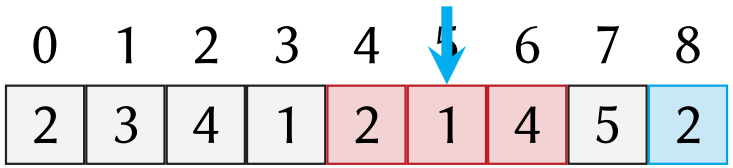
Output Array:



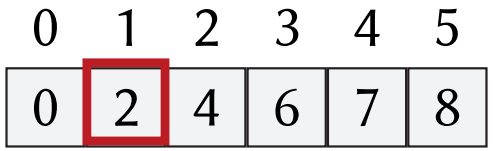
Now moving elements into the result array

Dealing with element 5

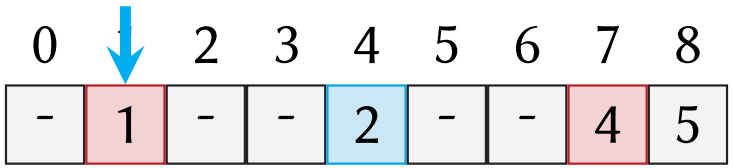
Input array:



Counts:



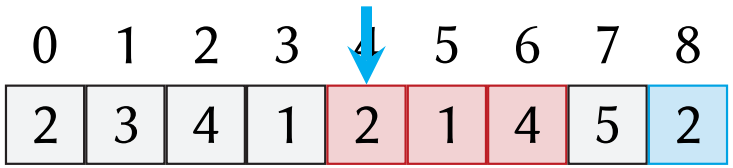
Output Array:



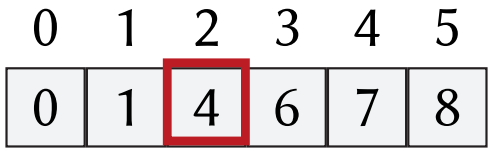
Now moving elements into the result array

Dealing with element 4

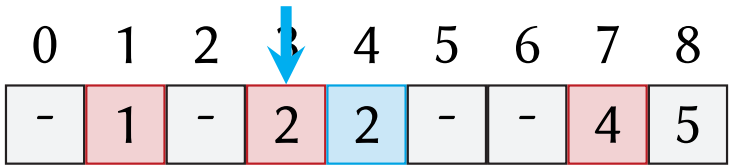
Input array:



Counts:



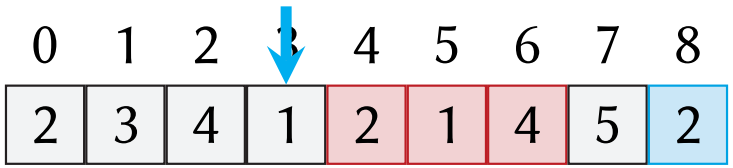
Output Array:



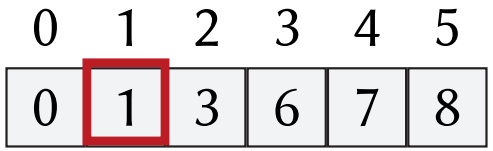
Now moving elements into the result array

Dealing with element 3

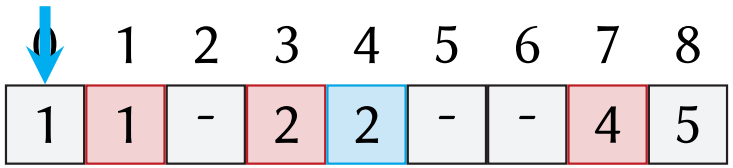
Input array:



Counts:



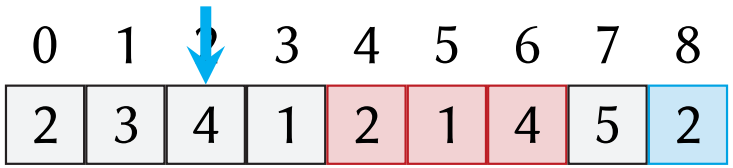
Output Array:



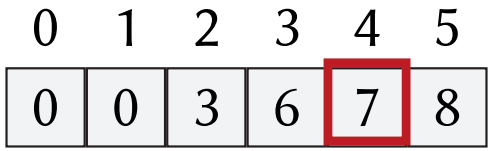
Now moving elements into the result array

Dealing with element 2

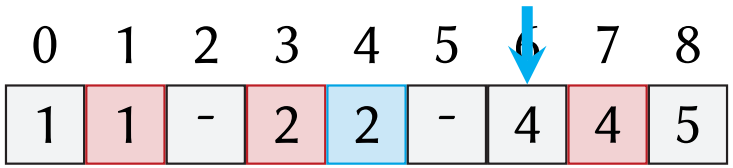
Input array:



Counts:



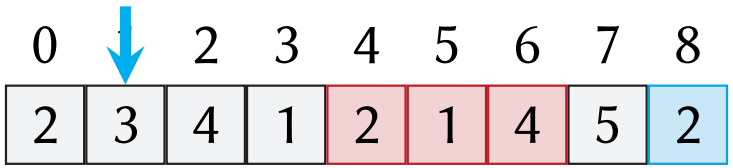
Output Array:



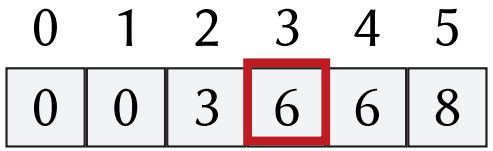
Now moving elements into the result array

Dealing with element 1

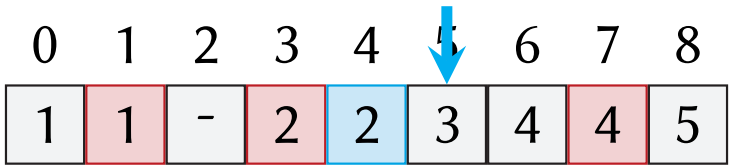
Input array:



Counts:



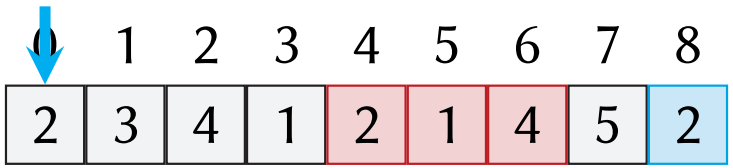
Output Array:



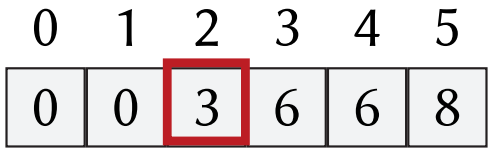
Now moving elements into the result array

Dealing with element 0

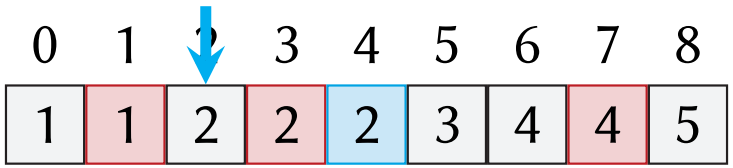
Input array:



Counts:



Output Array:



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

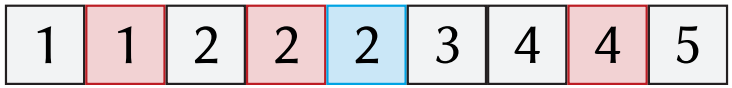
Counts:

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

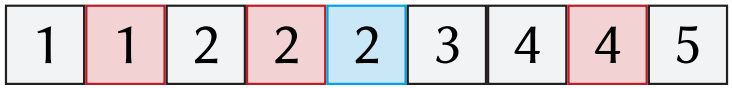
Output Array:

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

The Final Sorted array

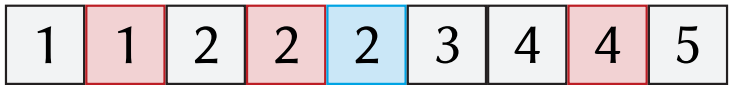


The Final Sorted array



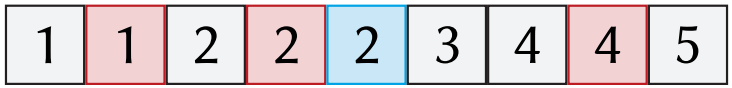
- It's sorted!

The Final Sorted array



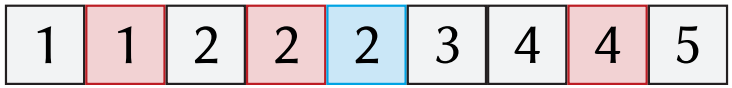
- It's sorted!
- It's stable!

The Final Sorted array



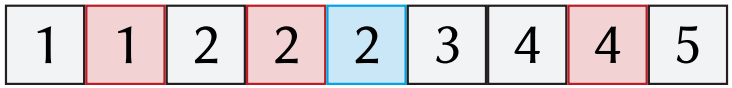
- It's sorted!
- It's stable!
- Note: the items that were first in the input are first in the output.

The Final Sorted array



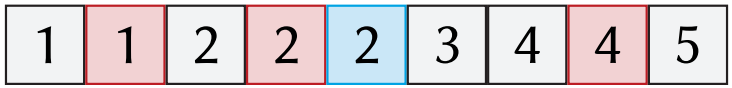
- It's sorted!
- It's stable!
- Note: the items that were first in the input are first in the output.
- Why did we start at the **end** of the array when filling the output array?

The Final Sorted array



- 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



- 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

Linear time sorting

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:

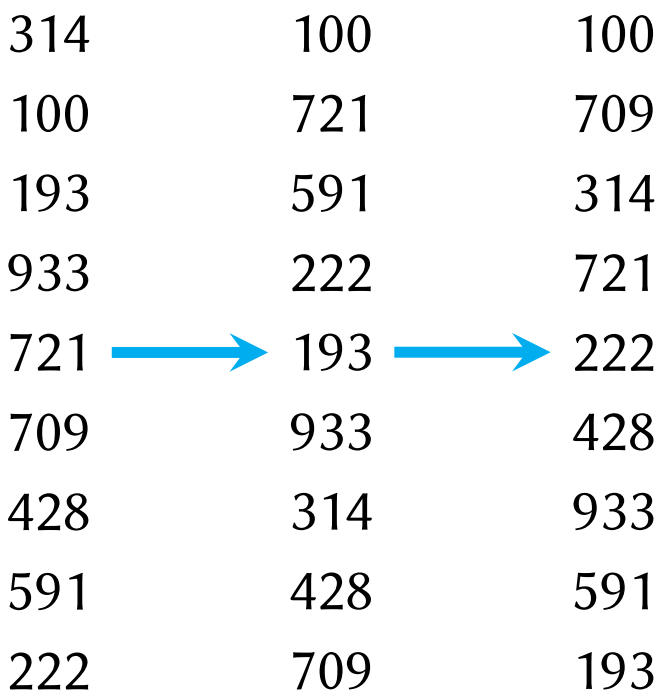
Sort by the next least sig digit (middle)

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

Radix Sort

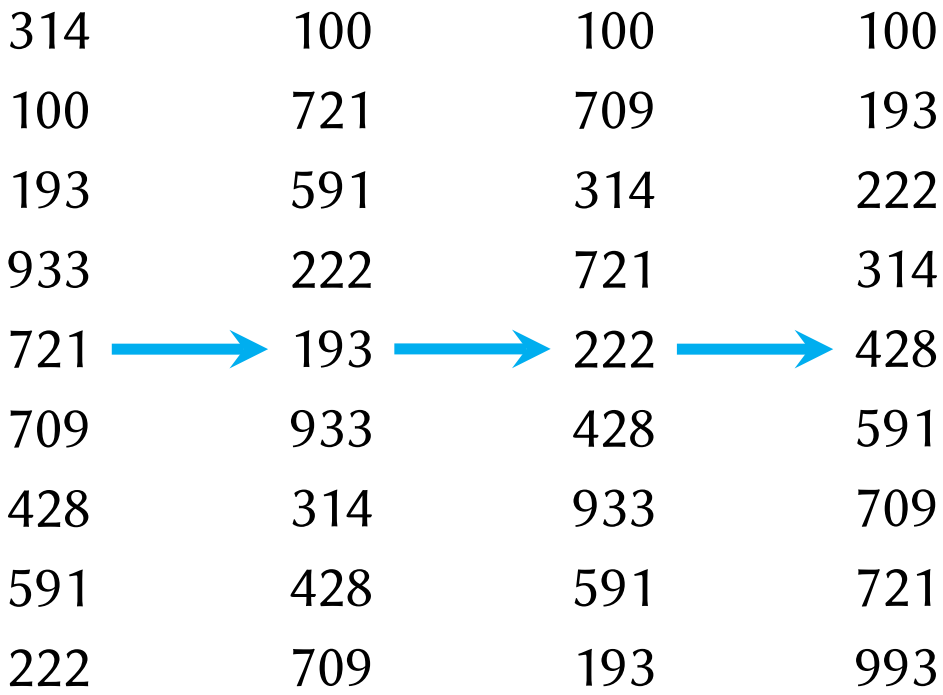
Using a stable sort:

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



Radix Sort

Using a stable sort:



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.

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.

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	$O(n^2)$	$\Theta(n + k)$	$\Omega(n + k)$

Sorting Algorithms and their runtime

Algorithm	Worst	Average	Best
Insertion sort	$O(n^2)$	$O(n^2)$	$\Omega(n)$
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	$O(n^2)$	$\Theta(n+k)$	$\Omega(n+k)$

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?

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

Section 6

Summary

Summary

What problems did we work on today?

- Sorting

What approaches did we use?

- Improving algorithm runtime by using a new data structure
- Non-comparison sorts

Problems

Algorithm Design Techniques

Algorithms

Tools/Skills

Problems

- Sorting
- Stable matching
- Inversions
- Single-Source Shortest Path
- Strongly connected components
- Interval Scheduling
- Scheduling to minimize lateness
- Variable Encoding

Algorithm Design Techniques

Algorithms

Tools/Skills

Problems

- Sorting
- Stable matching
- Inversions
- Single-Source Shortest Path
- Strongly connected components
- Interval Scheduling

Algorithm Design Techniques

- Divide and Conquer
- Greedy
- Graphs

Algorithms

- Scheduling to minimize lateness
- Variable Encoding

Tools/Skills

Problems

- Sorting
- Stable matching
- Inversions
- Single-Source Shortest Path
- Strongly connected components
- Interval Scheduling
- Scheduling to minimize lateness
- Variable Encoding

Algorithm Design Techniques

- Divide and Conquer
- Greedy
- Graphs

Algorithms

- Mergesort
- Insertion Sort
- Merge-And-Count
- Bucket sort
- Quicksort, Partition
- Counting sort
- Radix sort
- Heap Sort

Tools/Skills

Problems

- Sorting
- Stable matching
- Inversions
- Single-Source Shortest Path
- Strongly connected components
- Interval Scheduling
- Scheduling to minimize lateness
- Variable Encoding

Algorithms

- Mergesort
- Insertion Sort
- Merge-And-Count
- Bucket sort
- Quicksort, Partition
- Counting sort
- Radix sort
- Heap Sort

Algorithm Design Techniques

- Divide and Conquer
- Greedy
- Graphs

Tools/Skills

- Mathematical Induction
- Inductive proofs
- Recurrences
  - Substitution
  - Recursion Tree
  - Master Method
- Asymptotic Notation
- Summations
- Logs

Oh look, we've learned so much it doesn't fit on one page.

Our list of skills and algorithms has gotten too long...

Algorithms

- Mergesort
- Insertion Sort
- Merge-And-Count
- Bucket sort
- Quicksort, Partition
- Counting sort
- Radix sort
- Heap Sort
- Djikstra's
- Huffman
- ...

Tools/Skills

- Mathematical Induction
- Inductive proofs
- Recurrences
  - Substitution
  - Recursion Tree
  - Master Method
- Asymptotic Notation
- Summations
- Logs
- Proving greedy vs optimal
- Exchange argument
- Stays ahead argument
- Graph traversals
- Improving algorithms by using data structures

1. Logistics
2. Heap sort
   - A new data structure: Heap
   - Heapsort
   - Priority Queue
3. Quicksort
   - Partitioning
   - The Quicksort Algorithm
   - Analysis: Best Case
   - Analysis: Worst Case
   - Intuition
   - Average case analysis: Solve the recurrence
   - Picking a better pivot
   - Randomizing Quicksort
4. Linear time sorting
   - Bucket Sort
   - Counting Sort
   - Radix Sort
5. Sorting Recap
6. Summary