Sorting and Shuffling

Dov Kruger

Department of Electrical and Computer Engineering
Rutgers University

February 28, 2024

If you have a dictionary of words, is it easier to find a word if they are in known order?
For an array of n numbers, is it easier to find one if they are in order?

Sorting is a class of algorithm that **puts a list into a desired order**

For our purposes it does not matter what the order is, so we will choose **ascending order**

There are an astounding number of different sorting algorithms

- Highly studied problem
- Common need in early computing
- We will study 6 algorithms to gain insight into different approaches to problem solving

Sorting Algorithms

Slow and Useless, but Instructive

- Bubblesort
- Selection Sort

$O(n^2)$ so useless for Large Datasets, but fastest for small or almost-sorted data

- Insertion Sort

$O(n \ log \ n)$

- Quicksort
- Heapsort
- Mergesort

Further Study

We will not have time to study these, but two further algorithms to look into are

- Radix Sorting

Faster than $O(n\ log\ n)$ when the number of unique values is smaller than n

- Spreadsort (Newer hybrid algorithm that is faster than quicksort)

Swapping Elements of an Array

Most of the algorithms in this lesson sort by swapping elements that are out of order

How to do it?

- Use a temp variable
- Use inverse operations (see notes for details!)
- Use XOR which is its own inverse (see notes for details!)

```
swap(a, b)
  temp ← a
  a ← b
  b ← temp
end
```

Bubblesort

One of the simplest ways to sort is to compare adjacent elements

- If they are out of order, swap them

10	9	8	7	6	5	4	3	2	1
9	10								

Bubblesort, part 2

One of the simplest ways to sort is to compare adjacent elements

- Repeat moving through the list

10	9	8	7	6	5	4	3	2	1
9	8	10							
		7	10						
			6	10					
				5	10				
					…				
					4	3	2	1	10

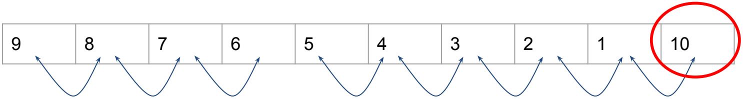
Complexity of Bubblesort Pass

One pass of bubblesort requires

- n-1 comparisons
- For each pair out of order, a swap (n-1 max)

The result is one element guaranteed in the right place

O(n)



Complexity of Bubblesort: $n * (n-1) = O(n^2)$

In order to guarantee the entire array is sorted, n-1 passes are required

Not n, the last time there is only 1 element which cannot be out of order

Example: n=5

pass 1

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

sorted part after each pass is circled

remaining number cannot be out of order

Modifications to Bubblesort: Early Termination

Worst case $O(n^2)$ but is it possible to stop early?

If the array is partially sorted, yes.

In any pass, if no swaps are made, then sorting is done

Example:

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

no swaps in pass 2,
algorithm can stop early

Modified Bubblesort Pseudocode

```
modifiedBubblesort(x)  //  O(n^2)
  for j ← x.length − 1 to 0      //O(n)
    done ← true
    for i ← 0 to j  //average is n/2     O(n)
      if x[i] > x[i+1]
       swap(x[i], x[i+1])
       done ← false
      end
    end
    if done
      return   // Ω(n)
    end
  end
end
```

Selection Sort

Completely different approach to bubble sort, but same worst-case complexity

No way to end early

Overview

- Find the biggest element (selection)
- Put into the last position
- Repeat for (n-1) remaining elements

SelectionSort

```
selectionSort(a)
  for i ← length(a)−1 downto 1 //n−1 = O(n)
    max ← a[0]
    maxpos ← 0
    for j ← 1 to i   // average is n/2 = O(n)
      if a[j] > max
        maxpos ← j
        max ← a[j]
      end
    end
    swap(a[maxpos], a[i]) // move biggest element t
  end
end
```

Complexity of Selection Sort

The selection sort finds the biggest (or smallest element) O(n)

It does not swap each time, but that is just a bigger constant
Complexity is

$$(n-1)+(n-2)+\ \dots\ +1 = n(n-1)/2 = \frac{1}{2}(n^2-n) = O(n^2)$$

It is not fundamentally better than bubblesort, and though faster, cannot end early

$$O(n^2) = \Omega(n^2) \rightarrow \theta(n^2)$$

Insertion Sort

Not better than Bubblesort or selection sort by complexity

Lower constant, very efficient

Best algorithm for small datasets and almost-sorted data

Works the way sorting a hand of cards works

- Pick up a card (1st is by definition in the correct order)
- Pick up each new card and insert into correct position
- Build up the sorted list

Insertion Sort Example

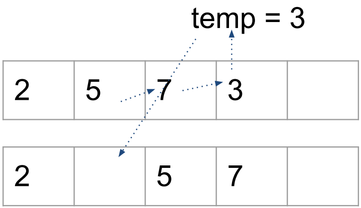
First element is already sorted

Add second element. If out of order insert before first

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

How to Insert

Insertion sort has a low constant because swapping is not necessary
Numbers move over until the new number fits into place
Roughly twice as fast as bubblesort Example: insert 3



- Store the new value into a temporary variable
  temp $\leftarrow$ 3
- iterate backward moving each element right until reaching a number $<$ 3
- Put temp into the correct location

Faster Performance: Swap Elements Further Apart

Insert is the best of the $O(n^2)$ sorts.

To do better, you must move values quickly to where they need to be

We will cover three better algorithms

- Quicksort
- Heapsort
- Mergesort

All have their advantages. None is best in all cases

All are $O(n\ log\ n)$

Comparing $O(n \log n)$ to $O(n^2)$

How different is $O(n^2)$ and $O(n \log n)$?

n	n^2	$n\ log\ n$ (approx)
10	100	13
100	10000	700
10^3	10^6	10^4
10^6	10^{12}	20×10^6
10^9	10^{18}	30×10^9

Quicksort

Pick a value (the pivot)

Place all values $<$ the pivot to the left all values $>$ to the right
Recursively call quicksort on each half

The key to quicksort is in picking the right pivot

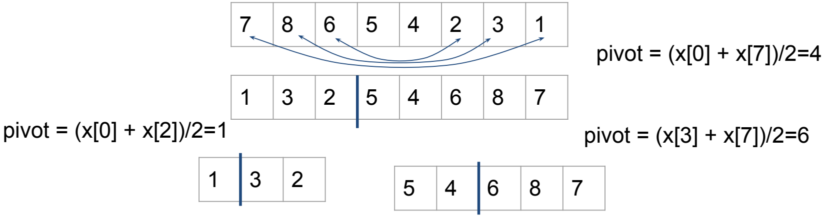
If picked badly, the sort can be $O(n^2)$

Tony Hoare, 1970:
https://www.youtube.com/watch?v=pJgKYn0lcno

Quicksort: Example of Optimal Partitioning

The following example shows quicksort at its best

Each pass splits the list into two parts, which are then sorted recursively



Lomuto Partitioning Variant

The Lomuto Partitioning variant is slightly more complicated

- Pick a pivot
- Move to one side (for example, right)
- Find an element on alternating sides that has to move and swap with pivot
- End up with the pivot in the middle
- Split the list in 3 parts ($<$, the pivot, and $\geq$)
- Leave the pivot in place and partition only the two sides

Problem with Quicksort

Badly picked pivots can wreak havoc

Most examples on the internet are wrong, and extremely slow

There are three not-terrible choices

- Random pivot - this is the best
- Average of first and last element
- Average of the first, last and middle element

Only the randomly chosen pivot can survive all data as we will see

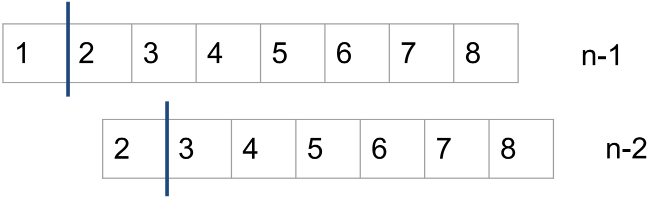
- Even that is uncomfortably up to chance!

Worst-case: Pivot is First or Last Element

If the pivot is selected as the first or last element

- Sorting sorted or nearly-sorted data will select the smallest or largest element
- This is effectively selection sort $O(n^2)$ with more overhead

Example: use first element as pivot (splits into 1, n-1)



Worst Case: Pivot is the average of first and last

Not as disastrously likely as choosing the first or last, but still can be pathological

Problem: construct a dataset for which this pivot choice will result in $O(n^2)$ runtime

1	3	5	7	8	6	4	2

7	5	3	1	2	4	6	8

Worst Case: Pivot is Average of First, Last and Middle

Surely no dataset can be found? Actually, it's not hard

1	4	7	2	6	8	5	3

Only Reliable Choice of Pivot: Random

Notice that prior examples might pick values not in the list at all

Picking a random pivot guarantees that the value selected is in the list

Even here, if the random number generator has problems...

Most implementations fall back to another method in case of trouble (heapsort)

Quiksort

Invented by Tony Hoare 1960

https://www.bl.uk/voices-of-science/interviewees/tony-hoare/audio/tony-hoare-inventing-quicksort

Heapsort

Heapsort relies on

- Turn the list into a **heap** efficiently
- Remove the largest (maxheap) or smallest (minheap) and put in correct position
- Repeat until the list is completely sorted

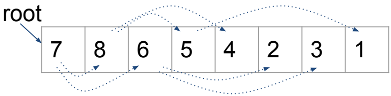
Definition of a maxheap

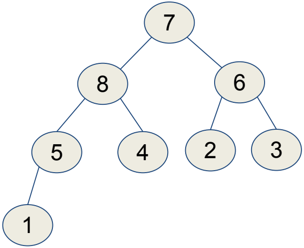
A binary tree in which each node $\geq$ its children

An array can be considered a binary tree with

Root is element 0

Each node at element i has children at 2i+1 and 2i+2



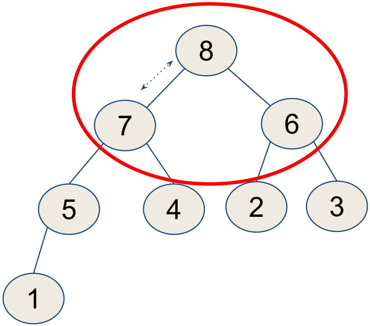


Start from the bottom (location n/2 or n/2-1)

```
int end = n/2−1
for i ← end downto 1
  makesubtree(x, i, n)
end
makesubtree(x, i, n)
  if x[2i+1] > x[2i+2]
    if x[i] < x[2i+1]  //left child is bigger
      swap(x[i], x[2i+1])
      makesubtree(x, 2i+1, n)
    end
  else // right child is bigger
    if x[i] < x[2i+2]
      swap(x[i], x[2i+2])
      makesubtree(x, 2i+2, n)
    end
```

Heapsort, step 1: Calling Makesubheap

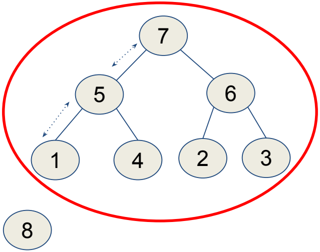
In this example, only the top subtree is not a heap already



Heapsort, step 2: Move First Element to the End

Once the largest value is moved to the end, shrink the tree by 1

The resulting tree is not a heap. Next step, make it a heap again.



sorted part

1	7	6	5	4	2	3	8

Complexity of Heap Sort

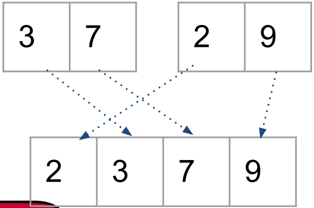
- Size of tree: n elements
- Depth of tree: $\log_2 n$
- Moving each element up to the correct location: $O(\log n)$
- Total cost of moving all n elements: n/2 $\log_2 n = O(n \log n)$
- Moving 1 element and reforming the heap: $1 + \log_2 n = O(\log n)$
- Repeating n times: $O(n \log n)$

Mergesort

Mergesort is the opposite of quicksort

- Start with individual elements which are by definition in order
- Merge sorted lists of size n into size 2n
- Requires extra storage

Example: Merging two sorted lists n=2



Mergesort Driver: Bottom-up

Simplest way to think about Mergesort

- Allocate n extra storage
- Merge all groups of size=1 into size=2 onto extra storage
- swap extra and list, double size and keep going

```
mergesort(list, n)
  tmp ← new int[n];
  for k ← 1, k < n; k ← 2k
    for i ← 0, i < n; i ← i + 2k
      merge(temp, list+i, list+i+k, n)
    end
    swap(list, tmp)
  end
```

Test Yourself

Mergesort Complexity

The first merge is n groups each of size 1: $1 * n = n$
The second merge is n/2 groups each of size 2: $2 * n/2 = n$
This happens $\log_2 n$ times. Therefore $O(n \log n)$

Radix sort

When the number of different values is much less than the number of elements to be sorted, it is possible to achieve better results

Extreme example: Only two values (0 and 1)

```
sort(list)
  countzero ← 0
  for i ← 0 to list.length
    if list[i] = 0
      countzero++
    end
  end
  for i ← 0 to countzero-1
    list[i] ← 0
  for i ← count to list.length
    list[i] ← 1
```

see https://cppsecrets.com/users/
1442971101051149897110100971154850485464103109971051084699
C00-boostsortspreadsortspreadsort.php

Shuffling

Shuffling is the opposite of sorting
If sorting is finding the one desired order...
Shuffling is taking a sorted list and scrambling it so any order is equally likely
We will cover 3 methods, only one is good

- Bad Shuffle
- Slow Shuffle
- Fischer-Yates

Bad Shuffle

This algorithm is unfair because not all permutations are equally likely

It looks good, with every location swapped for a random one

However, the first element is swapped with a random one, so later swapped again

If we conduct experiments, we can see that values are not equally distributed.

```
badShuffle(a)
  for i ← 0 to length(a)−1
    r ← random(0, length(a)−1)
    swap(a[i], a[r])
  end
end
```

Analyzing this one is hard because the inner loop involves random numbers

We can establish a firm upper bound of $O(n^2)$

```
badShuffle(a)
  out ← new array[size(a)]
  for i ← 0 to length(a)−1    //O(n)
    do                                //O(?)
      r ← random(0, length(a)−1)
    while a[r] = −1                   // problem: thi
    out[i] ← a[r]
    a[r] ← −1
  end
end
```

Fischer-Yates

```
FischerYates(a)
  for i ← length(a)−1 to 1
    r ← random(0, i)
    swap(a[i], a[r])
  end
end
```