Sorting theory and broken limits

Serge Kruk

October 12, 2021

We have seen sorts

- $\Theta(n^2)$ (Bubble, insertion, QS bad case)
- $\Theta(n^{\frac{3}{2}})$ (Shell)
- $\Theta(n \log^2 n)$ (Shell)
- $\Theta(n \log n)$ (Heap, Merge, QS)

How fast can we sort?

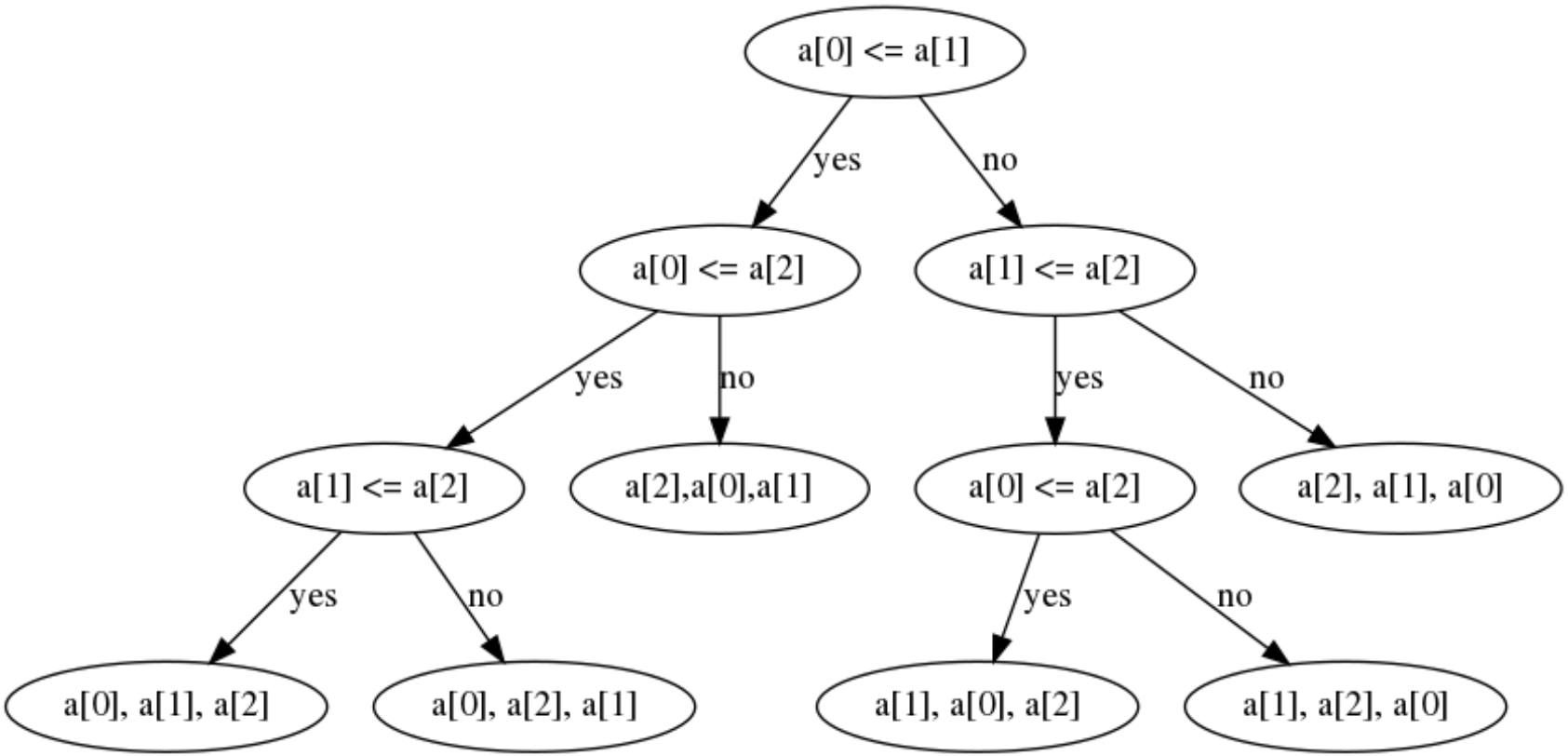
Assumptions:

- Sort is based on comparison.
- We are counting comparisons.
- All of the comparisons are equivalent in cost.

Let us consider a simple case

If we sort an array of 3 elements, how fast can we do it?

All possibilities for sorting a three-element array



Questions

- Is the previous tree optimal (smallest possible height)?

Smallest height?

- There are six permutations on 3 elements.
- Can we have a smaller binary tree with six leaves?

Essential elements of a decision tree

- Every leaf represents one permutation of the elements.
- A path from the root to a leaf is a particular sort.
- The height of the tree is the number of comparison required to sort.
- We therefore need a bound on the height of such a decision tree.

Decision tree for some specific cases

- How big is the smallest tree for a 4-element array?
- How big is the smallest tree for a 5-element array?
- How big is the smallest tree for an n-element array?

Recall

Binary tree heght

A binary tree of height k has at most 2^k leaves.

Decision tree for some specific cases

- How big is the tree for a 5-element array?
  - $5! = 120$ between $2^6 = 64$ and $2^7 = 128$ so height 7.
- How big is the tree for an n-element array?

Major theoretical result

Theorem

A decision tree to sort n elements has height $\Theta(n \log n)$.

Corollary

If we sort n elements via comparisons, the runtime is $\Omega(n \log n)$.

Proof

- We have $n!$ permutations of n elements, at least $n!$ leaves.

Proof

- We have $n!$ permutations of n elements, at least $n!$ leaves.
- A binary tree of height h has at most 2^h leaves.

Proof

- We have $n!$ permutations of n elements, at least $n!$ leaves.
- A binary tree of height h has at most 2^h leaves.
- Therefore, we need the smallest h such that $n! \leq 2^h$

$$n! \leq 2^h$$
$$\log(n!) \leq h$$

Proof

- We have $n!$ permutations of n elements, at least $n!$ leaves.
- A binary tree of height h has at most 2^h leaves.
- Therefore, we need the smallest h such that $n! \leq 2^h$

$$n! \leq 2^h$$
$$\log(n!) \leq h$$

- By Stirling's approximation $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta(\frac{1}{n})\right)$

$$n \log\left(\frac{n}{e}\right) + \log\sqrt{2\pi n} \leq h$$
$$n \log n - n \log e \leq h$$

Proof

- We have $n!$ permutations of n elements, at least $n!$ leaves.
- A binary tree of height h has at most 2^h leaves.
- Therefore, we need the smallest h such that $n! \leq 2^h$

$$n! \leq 2^h$$
$$\log(n!) \leq h$$

- By Stirling's approximation $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + \Theta(\frac{1}{n}))$

$$n \log\left(\frac{n}{e}\right) + \log\sqrt{2\pi n} \leq h$$
$$n \log n - n \log e \leq h$$

- Therefore, $h \in \Omega(n \log n)$.

Conclusions

- We need at least $n \log n$ comparisons to sort.
- Heapsort and Mergesort are asymptotically optimal!
- Quicksort, if it avoids the pathological cases, is also optimal.

Questions (hard)

- What are the trade-offs between QS, MS, HS?
- Under what conditions would we pick one over the others?

Hints:

- Space used?
- Parallelizable?
- Worst case important?
- Best case?

Now that we have shown that $O(n \log n)$ is optimal

And now for something completely different . . .

Sorting integers

- If I were to ask you to sort the numbers 1 to 15?

```
def sort_fast_1_100(a):
  # This array only has numbers in [1,..,100]
  n = len(a)
  count = [0]*101
  for i in range(n):
    count[a[i]] += 1
  b = []
  for i in range(101):
    for j in range(count[i]):
      b.append(i)
  return b
print([sort_fast_1_100([1,3,2,2,3,4])==[1,2,2,3,3,4],
```

Sorting integers

- If I were to ask you to sort the numbers 1 to 15?
- If I were to ask you to sort 1000 numbers, all integers in the range 1 to 100?

```
def sort_fast_1_100(a):
  # This array only has numbers in [1,..,100]
  n = len(a)
  count = [0]*101
  for i in range(n):
    count[a[i]] += 1
  b = []
  for i in range(101):
    for j in range(count[i]):
      b.append(i)
  return b
print([sort_fast_1_100([1,3,2,2,3,4])==[1,2,2,3,3,4],
```

Sorting integers

- If I were to ask you to sort the numbers 1 to 15?
- If I were to ask you to sort 1000 numbers, all integers in the range 1 to 100?
- If I were to ask you to sort one million integers, all in the range 1000 to 9999?

```
def sort_fast_1_100(a):
  # This array only has numbers in [1,..,100]
  n = len(a)
  count = [0]*101
  for i in range(n):
    count[a[i]] += 1
  b = []
  for i in range(101):
    for j in range(count[i]):
      b.append(i)
  return b
print([sort_fast_1_100([1,3,2,2,3,4])==[1,2,2,3,3,4],
```

Sorting integers

- If I were to ask you to sort the numbers 1 to 15?
- If I were to ask you to sort 1000 numbers, all integers in the range 1 to 100?
- If I were to ask you to sort one million integers, all in the range 1000 to 9999?
- If I were to ask you to sort a deck of cards?

```
def sort_fast_1_100(a):
  # This array only has numbers in [1,..,100]
  n = len(a)
  count = [0]*101
  for i in range(n):
    count[a[i]] += 1
  b = []
  for i in range(101):
    for j in range(count[i]):
      b.append(i)
  return b
print([sort_fast_1_100([1,3,2,2,3,4])==[1,2,2,3,3,4],
```

Card sorting

Consider the following:

- Take a deck of 52 cards
- $52 \log 52 \approx 300$
- Read one card at a time and put it into one of 13 piles (ace to King)
- Pick up those 13 piles, read one card at a time and put it in one of four piles (four suits)
- Pick up those four piles and you have 52 sorted cards.
- How many operations? $52 + 52 = 104 < 300$!!!!!!!

Another example

- Consider the array 36,9,0,25,1,64,16,81,4
- Put in bins according to last digit

0	1	2	3	4	5	6	7	8	9
0	1			64	25	36			9
	81			4		16			

- Read by column put in bins by first digit

0	1	2	3	4	5	6	7	8	9
0	16	25	36			64		81	
1									
4									
9									

- Read by column: 0,1,4,9,16,25,36,64,81

Faster sort

```
def binsort(a):
    # Sort an array of dates in YYYYMMDD format
    # stored as e.g. [[2,0,1,5,0,1,1,1],[2,0,1,5,1,2,1,0],[2,0,1,41,1
```

Faster sort

```
def binsort(a):
    bins = [a]
    for l in range(len(a[0])-1,-1,-1):
        binsTwo = [[] for _ in range(10)]
        for bin in bins:
            for e in bin:
                binsTwo[e[l]].append(e)
        bins = binsTwo
    return [e for bin in bins for e in bin]
```

Little tests

```
a=[[2,0,1,5,0,1,1,1],[2,0,1,5,1,2,1,0],
   [2,0,1,4,1,1,0,6],[2,0,1,4,1,1,0,5]]
binsort(a)
```

2	0	1	4	1	1	0	5
2	0	1	4	1	1	0	6
2	0	1	5	0	1	1	1
2	0	1	5	1	2	1	0

Trace this execution on the 10 buckets

0	1	2	3	4	5	6
20151210	20150111				20141105	20141106
20141105	20151210					
20141106	20150111					
	20141105	20151210				
	20141106					
	20150111					
20150111	20141105					
	20141106					
	20151210					
				20141105	20150111	
				20141106	20151210	
	20141105					
	20141106					
	20150111					
	20151210					

Efficiency

- We cannot count comparisons; we have none.
- Counting appends.

Efficiency

- Assume that we are sorting n elements, each with k digits.
- We have three `for` loops but, looking at their contents, we have n elements.

$$\begin{aligned} T(n) &= \sum_{l=0}^{l=k} \sum_{i=0}^{n-1} 1 \\ &= kn \end{aligned}$$

Correctness

LI: At step `l` of the outer loop, looking at the elements in the bins, in order of the bins (0-9), the elements are sorted according to the last `l` digits.

Radix sort

```
def radixSort(a):
    # Sorting an array a of elements with 'd' digits
    for i in range(len(a[0], -1, -1)):
        sortStably(a, lambda a0,a1 : a0[i] < a1[i])
```

Each sortStably is some other sort algorithm (often bucket).

Stable sort

Stability

A sort is stable if two elements with equivalent keys maintain their relative position after the sort.

Questions

- Is MergeSort stable?
- Is HeapSort stable?
- Is QuickSort stable?

Conclusions

- Yes, we can sort faster than $n \log n$.
- To do so requires that we use knowledge of the data.
- Multiple implementation, each adapted to data.
- Authors do not agree on names.
- Current "best" sort `learned sort` (Uses AI)

Subsets of sorting problems (sub-sorts?)

- What if you wanted the smallest and largest elements of an array?
- What if you wanted the k smallest and k largest elements of an array?
- What if you wanted the k largest elements of an array?

Min and max

Problem:

Given an array a, find the smallest and largest elements.

Min and Max

```
def minandmax(a):
    min = max = a[0]
    for i in range(1,len(a)):
        if a[i] < min:
            min = a[i]
        elif a[i] > max:
            max = a[i]
    return min,max
```

How many comparisons?

Efficiency

Between n and $2n$ comparisons. Can we do better?

Min and Max (better version)

```
def minandmax2(a):
    min, max = a[0],a[1] if a[0] < a[1] else a[1],a[0]
    for i in range(2,len(a),2):
        if a[i] < a[i+1]:
            if a[i] < min:
                min = a[i]
            if a[i+1] > max:
                max = a[i+1]
        else:
            if a[i+1] < min:
                min = a[i+1]
            if a[i] > max:
                max = a[i]
    return min,max
```

- Is it correct?
- How many comparisons?

Efficiency

Always $\frac{3n}{2}$ comparisons.

Homework/Test questions

- What does it mean for a sort to be stable?
- If we could find the median of an array in $O(n)$, could we improve quicksort runtime?
- Can we make all sorting algorithms seen previously stable?

Project

Problem

Given an (unsorted) array of n elements where each element is of the form AAA999AA99A9
where the A are characters in the range 'a' to 'z' and the 9 are characters in the range '0' to '9'.
Your task is to implement a sort of this array that will be as fast (asymptotically) as possible. (Much better than $n \log n$)