Sorting Lower Bound and Sorting Without Comparisons

Overview

We have seen several sorting algorithms so far, all of which have worst-case running time at least $\Omega(n \lg n)$. The natural question to ask is: Can we do better than $\Theta(n \lg n)$ in the worst-case? We introduce the concept of lower bound, and show that sorting lower bound in the comparison model of computation is $\Omega(n \lg n)$. We describe a couple of different ways to sort which do not use the comparison model and under certain assumptions achieve linear time.

1 Comparison-based Sorting lower bound

- We have seen several sorting algorithms. All have worst-case running time at least $\Omega(n \lg n)$. The natural question to ask is: **Can we do better than $\Theta(n \lg n)$ in the worst-case?** In other words, is there a sorting algorithm whose worst-case running time is asymptotically better than $\Theta(n \lg n)$; Or, can we show that it is impossible to sort faster than $\Theta(n \lg n)$ in the worst case? If we could, then this would establish a *lower bound* for sorting.

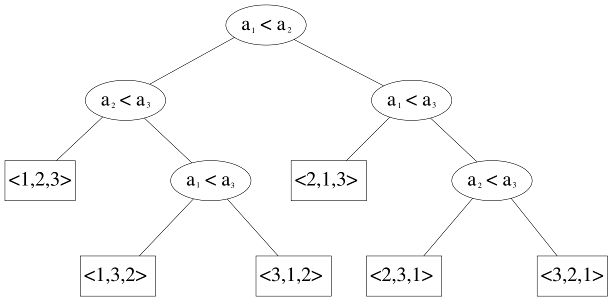
> A *lower bound* for a problem is a lower bound on the worst-case running time of any algorithm that solves that problem.

- In order to state that $L(n)$ is a lower bound for a problem, we must prove that *any* algorithm for that problem must take at least $L(n)$ (or $\Omega(L(n))$) in *the worst case*.
- To prove a lower bound of $\Omega(n \lg n)$ for sorting, we would have to prove that no sorting algorithm could possibly be faster, in the worst-case, then $\Theta(n \lg n)$.
- But..how to prove a lower bound, without going through all possible algorithms?!?!
- To try and prove a lower bound we have to go back to what is an "algorithm". The power of an algorithm depends on the primitive instructions we allow, which in turn depend on the machine model. How fast we can solve a problem depends on the model of computation. The more powerful the machine model, the smaller the lower bound. (Ultimately, if our model allowed *sort* as a primitive operation, then we could sort in one instruction, hence constant time).
- Today we'll show, that any sorting algorithm that uses only comparisons takes $\Omega(n \lg n)$ in the worst case. We'll model such algorithms using the *decision tree* (or *comparison*) model.

1.1 The Decision Tree/Comparison Model

- All sorting algorithms we have seen so far use *only comparisons* to gain information about the input.
- We will prove that such algorithms have to perform $\Omega(n \log n)$ comparisons. To prove this bound, we need a formal model of an algorithm that uses only comparisons.

- **Comparison/decision tree model**: an algorithm is represented only by the input comparisons it performs. Imagine that we remove all other instructions from the algorithm and only keep the comparison between the input elements.
- For simplicity we assume all comparisons are of the form $a_i \leq a_j$ (other comparison can be re-written to be in this form)
- Here's how an algorithm sorting 3 elements a_1, a_2, a_3 might look like:



The first comparison performed by this hypothetical algorithm is $a_1 \leq a_2$. If this is true, it follows on the left brach and continues with $a_2 \leq a_3$. If that's true as well, it concludes that $a_1 \leq a_2 \leq a_3$, and outputs $< a_1, a_2, a_3 >$ as sorted. If the result of $a_2 \leq a_3$ is false (in other words: $a_1 \leq a_2, a_3 < a_2$), it cannot conclude sorted order yet; it performs one more comparison, namely $a_1 \leq a_3$. If this is true, then it concludes that sorted order is a_1, a_3, a_2; if false, it concludes that sorted order is a_3, a_1, a_2. And so on.

Note that executing the algorithm on a particular input corresponds to performing the comparisons starting from the root and following a path to one of the leaves. Each leaf corresponds to the specific permutation that sorts that particular input. In this case, for an array of 3 elements, there are 6 leaves corresponding to the 3! possible permutations of the input.

- In general, each comparison-based sorting algorithm corresponds to a *decision tree*: binary tree where each internal node is labeled $a_i \leq a_j$; Each leaf of the tree represents a possible sorted permutation
- An execution of the algorithm on a particular input corresponds to a root-to-leaf path in this tree; it corresponds to identifying the permutation that sorts that input.
- The tree must be able to sort any possible input. Put differently, for each input the tree must be able to compute the permutation of the input that gives sorted order.
- Worst case number of comparisons performed corresponds to maximal root-leaf path (=height of tree).
- Therefore lower bound on height $\Rightarrow$ lower bound on sorting.
- If we could prove that any decision tree on n inputs must have height $\Omega(n \lg n) \Rightarrow$ any sorting algorithm (that uses only comparisons) must make at least $\Omega(n \lg n)$ comparisons in the worst case.

1.2 Sorting Lower Bound in the Decision Tree/Comparison Model

$\boxed{\textbf{Theorem:}\ \text{Any decision tree sorting } n \text{ elements has height } \Omega(n \log n).}$

Proof:

- Assume n distinct elements
- There must be $n!$ leaves (one for each of the $n!$ permutations of n elements)
- A tree of height h has at most 2^h leaves
- Therefore the height h must be so that $2^h \geq n!$
- We get

$$\begin{aligned}
2^h &\geq n! \\
h &\geq \log(n!) \\
h &\geq \log(n(n-1)(n-2)\cdots(2)) \\
&= \log n + \log(n-1) + \log(n-2) + \cdots + \log 2 \\
&= \sum_{i=2}^{n} \log i \\
&= \sum_{i=2}^{n/2-1} \log i + \sum_{i=n/2}^{n} \log i \\
&\geq 0 + \sum_{i=n/2}^{n} \log \frac{n}{2} \\
&= \frac{n}{2} \cdot \log \frac{n}{2} \\
&= \Omega(n \log n)
\end{aligned}$$

It follows that:

$\boxed{\text{Any sorting algorithm that uses only comparisons takes } \Omega(n \log n) \text{ in the worst case.}}$

2 Sorting in linear time

- We know that it is not possible to sort n elements faster than $\Omega(n \lg n)$ in the worst case **when using only comparisons**.
- Are there other ways to sort? What else could we use besides comparisons?
- To see what else we might use, let's think of the following problem:

  Exercise: Assume you have an array of elements such that they are all integers in the range 0..999. Can you sort them in $O(n)$ time?

  Take a few minutes and try to come up with an algorithm.
- As we can see from this example, it is possible to sort using something other than comparisons of the input, but we crucially exploit special properties of the input (such as integers in a small range).
- The most common linear-time sorting algorithms algorithms are bucket sort and counting sort. which we discuss below. Both these algorithms handle the same sort of input you saw in the exercise: integers in a certain range.
- It is important to distinguish how a sorting algorithm handles elements with the same value; to this end we introduce a new property called "stability":
- $\boxed{\text{A sorting algorithm is called \textit{stable} if it preserves the relative order of equal elements}}$
- Stability is important when sorting by multiple keys.
- **Example:** We have a set of people: Ann Johnson, Lily Smith, Zach Johnson, which we want to sort by last name, and, for people with same last name, by first name (this is called *lexicographic order*).
- We can implement this order by doing two passes of [your favorite sorting algorithm]:

  1. First sort by FirstName;
  2. Then sort by LastName.

  In this case, sorting by first name would give:

  1. Ann Johnson, 2. Lily Smith, 3. Zach Johnson,

  Then sorting by last name would give:

  { Ann Johnson, Zach Johnson } , Lily Smith

  For the second pass of sorting, both Ann and Zach have same last name; if the sorting algorithm is stable, it leaves the Johnson's in the same order in which it found them, that is, Ann first then Zach. Because the names are sorted by first name after the first pass, if the sorting algorithm is stable, this ensures that people with same last name appear in the order of their first names.

2.1 Bucket sort

- Input: integers in the range $\{0, .., N\}$, for some $N \geq 2$.
- Note that in general the elements could contain other data besides the integer part. For simplicity we will only consider the integer part and ignore the rest of the object.
- Note that we denote n the *size* of the input, and N the *range* of the input.

```
BUCKET-SORT($A$)
1  // Input: array A[] of $n$ elements assumed to be integers in range $0..N$
2  // Output: array B[] sorted order
3  Create an array $C[]$ of size $N + 1$ //the buckets
4  Create the output array $B[]$ of size $n$
5  For $i = 0$ to $n - 1$
6        add $A[i]$ to $C[A[i]]$ at the end
7  For $i = 0$ to $N$
8        traverse $C[i]$ and append its elements in order to $B[]$
9  return sorted array $B$
```

- We think of array B as an array of "buckets" or lists, where element $B[k]$ holds the list of elements with key k. For e.g. if element 24 occurs in the input, we insert it in $B[24]$
- Analysis: $O(n + N)$ time and $O(N + n)$ extra space.
- Bucket-sort is stable if we append the elements at the *end* of the bucket.

2.2 Counting sort

- Input: integers in the range $\{0, .., N\}$, for some $N \geq 2$.
- Note that in general the elements could contain other data besides the integer part. For simplicity we will only consider the integer part and ignore the rest of the object.
- Counting-sort uses the same idea as bucket sort, except that, instead of creating buckets (with linked lists), it stores everything in an array. It uses less space and is more elegant.

```
Counting-Sort($A$)
 1  // Input: array A[..] of $n$ elements assumed to be integers in range 0..$N$
 2  // Output: array B[...] sorted order
 3  Create an auxiliary array $C[]$ of size $N + 1$
 4  Create the output array $B[]$ of size $n$
 5
 6  // initialize the counts
 7  For $i = 0$ to $N$
 8       $C[i] = 0$
 9
10  For $i = 0$ to $n - 1$
11       $C[A[i]] + +$
12  // Claim: After this loop $C[i]$ represents how many elements in $A$ are $== i$
13
14  For $i = 1$ to $N$
15       $C[i] = C[i] + C[i - 1]$
16  // Claim: After this loop $C[i]$ represents how many elements in $A$ are $<= i$
17  // $C[i]$ tells us where value $i$ will be in the sorted order of $A$!
18
19  For $i = n - 1$ down to 0
20       // put $A[i]$ at index $C[A[i]]$ in $B[]$ and adjust for 0-based
21       $B[C[A[i]] - 1] = A[i]$
22       decrement $C[A[i]]$ //previous value $i$ will be right before
23  return array B
```

- Analysis: $O(n + N)$ time and $O(N + n)$ extra space.
- Counting-sort is stable. This is assured by the last loop (lines 8-9) which goes backwards.

3 Bucket-sort, Counting-sort or Quicksort?

- Bucket-sort uses more space which makes it less efficient than Counting-sort, by a constant factor (that is, they are asymptotically the saame, but the constants in the $\Theta()$ for Counting-Sort is smaller than the constant in the $\Theta()$ in Bucket-Sort).
- How does $\Theta(n + N)$ compare with $\Theta(n \lg n)$? Put differently, when is Counting-sort efficient over e.g. Quicksort or Heapsort?
  - The bottom-line is, if the range is small, you probably want to use Counting-sort.
    For e.g. sort n=10,000 elements with keys which are integers in range $\{0, 1, .., 10\}$. Counting-sort will be faster than Quicksort (by a lot).
  - If the range N is large and n is small, Quicksort will be faster
    For e.g. sort n=1,000 elements with keys in range $\{0, 1, 2, ..., 10^6\}$. Counting-sort will be slower than Quicksort.
  - If the range is large, and n is large, it can depend.
- Could we use Counting-sort/Bucket-sort to sort (regular) integers?

Answer: Let's analyze. An integer is represented on a machine with 4 bytes=32 bits, and its range is approx. $[-2^{31}, 2^{31}]$; that's approx. 4 billion values. Counting-Sort will need to store an auxiliary array of 2^{32}=4 billion elements!! Therefore for very small n, Quicksort will be (much) faster than Counting-Sort. For large n, $n > 2^{32}$, Counting-Sort will be faster. Where is the cross-over point? Consider $n = 2^{25}$. Quicksort runs in $O(25 \cdot n) \sim 2^{30}$ on the average (because $\lg n = 25 \sim 2^5$). Counting-sort will run in one pass $O(n + 2^{32})$ but require an additional array of 4 billion integers. They'll be close. To see which one is better in practice will need an experimental evaluation.

4 Study questions and practice problems

1. What is the smallest possible depth of a leaf in a decision tree for a comparison-based sort of n elements?
2. Consider the array
$$A = [7, 1, 3, 4, 7, 5, 2, 1, 4, 5, 7, 9, 4, 3]$$
of $n = 15$ elements in the range $\{0, .., 9\}$. Step through Counting-sort.
3. Argue that Counting-sort is stable.
4. Suppose that we were to rewrite the last "for" loop in Counting-sort as: for j=0 to n-1 (instead of: for j=n-1 down to 0). Would the algorithm work (i.e. sort properly)? What would change?
5. For each algorithms below, consider whether it is stable:

   A. Mergesort B Quicksort C Heapsort D Insertion sort
6. Assume you have n elements in the range $\{-20, -19, ..., ..., 19, 20\}$. How would you modify Counting-sort to sort this array?
7. Assume you have $n = 1,000$ integers in the range $\{0, .., 50\}$ and you need to sort them. Would you use Counting-sort or Quicksort, and why?
8. Describe one scenario when Counting-sort is more efficient than Quicksort.
9. Describe one scenario when Quicksort is more efficient than Counting-sort.
10. Describe one scenario when you cannot use Counting-sort.