Linear Time Sort

CS 5633 Analysis of Algorithms

Computer Science
University of Texas at San Antonio

September 30, 2024

Lower Bound for Comparison Sort

Why Some Algorithms Sort Faster?

- All of the sorting algorithms we have considered so far have been **comparison sorts**. That is, we only use comparisons to determine the relative order of elements.
- Algorithm design is essentially searching for a procedure that eliminates unnecessary operations.
- For sorting, it is essentially eliminating unnecessary comparisons.
  - If we already know $A[i] < A[j]$ and $A[j] < A[k]$, there is no need to compare $A[i]$ and $A[k]$.
  - If we have
    1. Two arrays, $B \in A$ and $C \in A$.
    2. Any element of B is smaller than $A[j]$.
    3. Any element of C is greater than $A[j]$.
    4. Then we do not have compare any elements of B and C.
    5. In the ideal case, $|B| = |C| = \frac{|A|}{2}$.

Why Some Algorithms Sort Faster? cont.

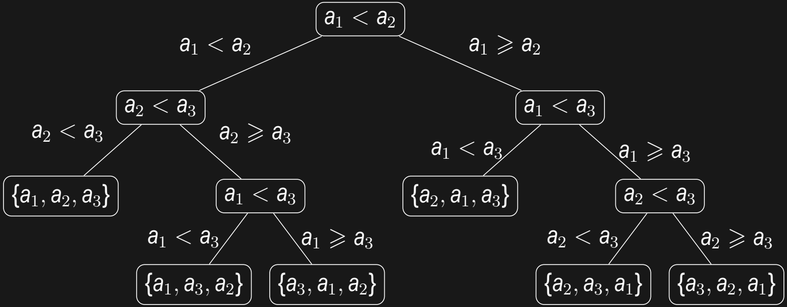
- Insert sort and bubble sort only eliminate a few unnecessary comparisons.
- Quick Sort can eliminate many unnecessary comparisons, if it finds a good pivot.
- Merge sort always eliminates many unnecessary comparisons (we will see that it eliminate all unnecessary comparisons).

The Lower-Bound of Comparison Sort Performance

- The best algorithm we have seen in terms of worst-case running time is $O(n \log n)$.
- Can we do better? That is, is it possible to design a comparison-based sorting algorithm which has worst-case running time $o(n \log n)$ (e.g. $O(n)$ or $O(n \log \log n)$)?
- We will use a **decision tree** to help us answer this question.

The Decision Tree For Comparison Sort

- A **decision** tree is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.
- Example, the insertion sort decision tree when sorting array $A = \{a_1, a_2, a_3\}$.



Properties of The Decision Tree For Comparison Sort

1. Each node represents a comparison.
2. Always a binary tree as there are only two out comes for a comparison.
3. Leaf node is a permutation of the input array. One of the leaf node represents the sorted array based on the input array.
4. The length of a path to a leaf node represents the number of comparisons to reach that permutation.
5. The height of the tree is the maximum number of comparisons needed to sort an input. It is also the worst case performance of this decision tree's algorithm, i.e., the algorithm's time complexity.

The Lower Bound of Comparison Sort

- Now, let's consider an imaginary best sorting algorithm's decision tree.
    1. It is a binary tree.
    2. Each leaf node is a permutations of input array $A[1:n]$. Since there are $n!$ permutations, there are at mostly $n!$ leaf.
    3. As this sorting algorithm is the best, it's time complexity is then the lower bound of comparison sort.
    4. This algorithm's time complexity is its decision tree's height.
- For any binary tree, with l leaf nodes, it is height h is at least $\log_2 l$. That is $h \geqslant \lg l$.

The Lower Bound of Comparison Sort cont.

- Consequently, for this best algorithm's decision tree with $n!$ leaves, it is height h is at least $\log_2 n!$. That is

$$
\begin{aligned}
h &\geqslant \lg n! \\
&= \lg(n(n-1)(n-2)\ldots 1) \\
&= \lg n + \lg(n-1) + \ldots + \lg 1 \\
&= \sum_{i=1}^{n} \lg i \\
&= \sum_{i=1}^{n/2-1} \lg i + \sum_{i=n/2}^{n} \lg i \\
&\geqslant 0 + \sum_{i=n/2}^{n} \lg \frac{n}{2} = \frac{n}{2} \cdot \lg \frac{n}{2} = \Omega(n \lg n)
\end{aligned}
\qquad (1)
$$

The Lower Bound of Comparison Sort cont.

- Based on the Equation (1), the best sorting algorithm's minimum execution time is $\Omega(n \lg n)$.
- The lower bound for comparison sort is then $\Omega(n \lg n)$.
- As the worst-case execution time of merge sort is $O(n \lg n)$, merge sort is the asymptotically optimal comparison sort.

Linear Time Sort

Linear Time Sort

- Comparison sort does not require any special properties on the input data.
- However, sorting can be much easier if we know the inputs have some special properties.
  - For example, the range of the input values.
- Knowing special properties allows us to design linear time ($O(n)$) time sorting algorithms.

Counting Sort

- If we know the range of the input values, we can simply count how many time each value occurs.
- After counting, we create the sorted array based on the counts of occurrence.
- This idea is called Counting Sort.

Algorithm 1: Counting Sort

```
Input: Array $A[1:n]$, with $0 \leqslant A[i] \leqslant k$
Output: Sorted Array $B[1:n]$
1  Array $C[0..k]$ = {0};
2  for $j$=1 to $n$ do
3      $C[A[j]]$ ++; // $C[i]$ has the number of the elements equal to $i$ ;
4  for $i$=1 to $k$ do
5      $C[i] = C[i] + C[i-1]$;
6      // $C[i]$ has the number of the elements less and equal to $i$ ;
7  for $j$=n down to 1 do
8      $B[C[A[j]]] = A[j]$; // create $B$ based on $C[i]$'s count;
9      $C[A[j]] - -$;
```

An Example of Counting Sort

- Let A be $\{4, 1, 3, 4, 3\}$.
- After loop at line 2 we have $C = \{0, 1, 0, 2, 2\}$.
  - That is, we have zero 0, one 1, zero 2, two 3 and two 4.
- After loop at line 4 we have $C = \{0, 1, 1, 3, 5\}$.
- For loop at line 7 we have,
  - After iteration $j = 5$: $B = \{0, 0, 3, 0, 0\}$ and $C = \{0, 1, 1, 2, 5\}$.
  - After iteration $j = 4$: $B = \{0, 0, 3, 0, 4\}$ and $C = \{0, 1, 1, 2, 4\}$.
  - After iteration $j = 3$: $B = \{0, 3, 3, 0, 4\}$ and $C = \{0, 1, 1, 1, 4\}$.
  - After iteration $j = 2$: $B = \{1, 3, 3, 0, 4\}$ and $C = \{0, 0, 1, 1, 4\}$.
  - After iteration $j = 2$: $B = \{1, 3, 3, 4, 4\}$ and $C = \{0, 0, 1, 1, 3\}$.

Counting Sort Run Time

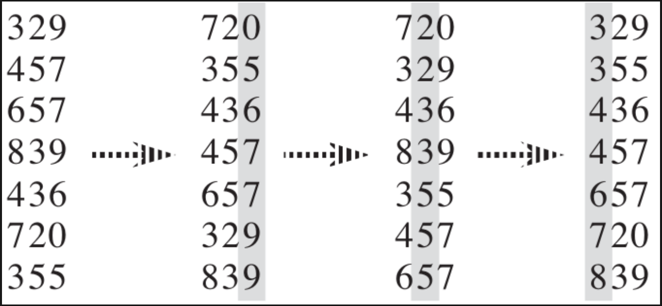
- Loop at line 2 has run time $O(n)$.
- Loop at line 4 has run time $O(k)$.
- Loop at line 7 has run time $O(n)$.
- Therefore, the total run time of Counting Sort is $O(n+k)$.

Counting Sort Properties

- Note that we did not ever directly compare any two elements of A.
- Also note that if $k = O(n)$, then counting sort takes $\Theta(n)$ time (which is better than the $\Omega(n \log n)$ lower bound of comparison sorting algorithms.
- In the worst case, k can be arbitrarily large, and therefore the running time of the algorithm can be arbitrarily large (with respect to n).
- Counting sort is a **stable** sort. That is, if the same value appears multiple times, then their input order is preserved in the output array.

Radix Sort

- To better handle extremely large values (large *k*), let's consider digit-by-digit sort.
- Idea: sort the input on the *least-significant digit* first (i.e. digits from right to left) using a stable sorting algorithm (like counting sort).
- An example of Radix Sort:



329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix Sort Run Time

- Assume there are n numbers, each with d digits, and each digit has a range of $[0, k]$
- To sort one digit, we need $O(n + k)$ time.
- To sort all digits, we need $O(d(n + k))$ time.
- Let's consider a case of sorting decimal (10-based) numbers:
  - The array has n elements.
  - Each digit of a decimal number can be 0 to 9, so $k = 10$.
  - To represent n distinctive numbers, we need roughly $\log_{10} n$ digits.
  - Therefore, the total run time is $O(\log_{10} n(n + 10)) = O(n \lg n)$, which is even worse than counting sort and getting close to the run-time of comparison sort.

Optimal Radix Sort

- Note that we can convert a decimal number to any other base (i.e. binary or hexadecimal) and sort these numbers instead (sorted order of binary/hex will be the same as sorted order for decimal).
- Note that if we convert the number to binary, then the number of digits for radix sort increases, but the range of values for counting sort decreases ($k = 2$).
- Note that if we convert the number to hexadecimal, then the number of digits for radix sort decreases, but the range of values for counting sort increases ($k = 16$).
- What base should we use to minimize the running time of the algorithm?

Optimal Radix Sort cont.

- We can view the problem as sorting n binary numbers of b bits each.
  - The maximum value is $2^b - 1$ then.
- Consider some positive integer $r \leq b$. We can view each number as having b/r digits. Each digit is in the range $[0, 2^r - 1]$. Then the algorithm would make b/r calls to counting sort with $k = 2^r$.
  - E.g., for hexadecimal representation, r is 4. That is, 4 bits represent one hexadecimal digit.
- What choice of r should we make to minimize the running time of the algorithm.

Optimal Radix Sort cont.

- Counting sort takes $\Theta(n+k)$ time, and our $k = 2^r$ which implies each call to counting sort takes time $\Theta(n+2^r)$.
- Since there are b/r calls to counting sort, we have the overall running time is $\Theta(\frac{b}{r}(n+2^r))$. What choice of r minimizes this function?
  - $r = \lg n$ minimize this function.
  - Intuitively,
    - The goal is to get close to linear run time.
    - For $\frac{b}{r}$, we would like it to be smaller than 1 (or any constant), so $r \geqslant b$.
    - For $(n+2^r)$, we would like it to be at most $2n$ (or $\Theta(n)$), so $r \leqslant \lg n$.
    - When n is sufficiently large, $r = \lg n$ gives a linear result.

Optimal Radix Sort cont.

- So with $r = \lg n$, we get that the running time of radix sort when the input is n numbers of b bits each is $\Theta(\frac{bn}{\lg n})$.
- Therefore, if the range of values is a constant (and therefore b is a constant), then the algorithm runs in $o(n)$ time.
- If the range of values is from 0 to $n^d - 1$, then $b = d \lg n$. The algorithm would have running time $\Theta(dn)$.
- If the range of values is arbitrarily large, then the running time of radix sort is arbitrarily large.