Contents

1 Lower Bound on Sort

Comparison-based algorithms: Insertion sort, merge sort, etc... all rely on sorting by comparing elements to one another.

These algorithms can be represented by a *decision tree*. Each node of the decision tree is a comparison, children are comparison you should make after each possible outcome.

Every comparison-based sorting algorithm can be represented by a decision tree. Any algorithm that relies on comparing elements to each other can be broken down like this.

A path in the tree is a trace of the algorithm.

So the worst case running time is height of tree since we might have to trace all the way down it.

Theorem: Any tree that can sort n elements must have height $\Omega(n \log n)$.

Proof: There must be $n!$ leaves since we have to be able to get every possible ordering. The decision tree is binary so we must have that for height h, $2^h \geq n!$. So

$$h \geq \lg(n!) \geq \log((n/e)^n) \geq n\log(n) - n\log(e) = O(n \log n)$$

. The last step is by Stirling's Theorem, which says $n! \approx (n/e)^n$.

2 Counting Sort

Counting sort can beat comparison sort bound *because* it doesn't use comparisons. In order not to use comparisons, we must have a little bit of "extra" knowledge. Namely: given an array A to sort, we need to be able to map every element that might appear in A uniquely to integers in $[1, k]$ where k is a small integer.

Input: A array to be sorted

Output: B sorted array of elements of A

Pass 1: Create array C of length k. $C[i]$ stores the number of times i appears in A. For each i, $C[A[i]] = C[A[i]] + 1$

Pass 2: For each entry i in C, put $C[i]$ i's in B.

This takes $O(n+k)$.

Problem: B is not *stably* sorted. Two equal keys may swap their relative orders. We would like to avoid that.

New algorithm: Create C', which stores in $C'[i]$ number of numbers in A less than or equal to i. Fill in C' after filling in C just by keeping running total.

Now, for $j = length[A]$ downto 1, place $A[j]$ at $C'[A[j]]$ in B and decrement $C'[A[j]]$. Now the sorting is stable.

Example: We know we are sorting numbers from 1 to 8

$$\begin{aligned} A &= [2,7,5,3,5,4] \\ C &= [0,1,1,1,2,0,1,0] \\ C' &= [0,1,2,3,5,5,6,6] \end{aligned}$$

Stepping through the second pass (0 in B indicates nothing there yet):

1.
$$\begin{aligned} B &= [0,0,0,0,0,0] \\ C' &= [0,1,2,3,5,5,6,6] \end{aligned}$$

2.
$$\begin{aligned} B &= [0,0,4,0,0,0] \\ C' &= [0,1,2,2,5,5,6,6] \end{aligned}$$

3.
$$\begin{aligned} B &= [0,0,4,0,5,0] \\ C' &= [0,1,2,2,4,5,6,6] \end{aligned}$$

4.
$$\begin{aligned} B &= [0,3,4,0,5,0] \\ C' &= [0,1,1,2,4,5,6,6] \end{aligned}$$

5.
$$\begin{aligned} B &= [0,3,4,5,5,0] \\ C' &= [0,1,1,2,3,5,6,6] \end{aligned}$$

6.
$$\begin{aligned} B &= [0,3,4,5,5,7] \\ C' &= [0,1,1,2,3,5,5,6] \end{aligned}$$

7.
$$\begin{aligned} B &= [2,3,4,5,5,7] \\ C' &= [0,0,1,2,3,5,5,6] \end{aligned}$$

3 Radix Sort

Digit-by-digit sort of list of numbers. Sort on least-significant digit first using a *stable* sort.

Why least significant? Example: Most significant

33		33		52
55	$\rightarrow$	55	$\rightarrow$	33
52		52		55

Oops!

Why do we need a stable sort? Because we don't want to mess up orderings caused by earlier digits in later digits!

Example: Sort 33, 55, 52

33		52		33
55	$\rightarrow$	33	$\rightarrow$	52
52		55		55

We can only guarantee that 52 comes before 55 if we can guarantee the sort on the second digit is stable!

Running Time: Sorting n words of b bits each: Each word has b/r 2^r-base digits. Example: 32-bit word is has 4 8-bit digits. Each counting sort takes $O(n + 2^r)$, we do b/r sorts. Choose $r = \log n$, gives running time $O(nb/\log n)$.

Correctness: By induction on number of digits.

Base Case: If all numbers have one digit, we sort them correctly by definition of the sorting algorithm.

Inductive Step: Assume we can sort numbers with $k-1$ digits correctly. Now consider sorting numbers of k digits. The first $k-1$ sorts, sort the last $k-1$ digits of the numbers correctly by the inductive hypothesis. The kth sort sorts the last k digits correctly. Now consider two numbers n_1 and n_2. Assume, WLOG that $n_1 < n_2$. If the kth digits of n_1 and n_2 are different, n_1 precedes n_2 in the sorting order since we just sorted on the kth digit. If the kth digits are the same, then n_1 and n_2 are in the same relative order as they were before the sort since the sort is stable. Since the kth digits are the same, we must have that the last $k-1$ digits of n_1 is smaller than the last $k-1$ digits of n_2 so n_1 preceded n_2 in $k-1$ sort by the inductive hypothesis.

4 Graph Overview

Graph $G = (V, E)$, V vertices, E edges. Can be *directed* or *undirected*. Upper bound on edges for directed graph: $n(n-1)$, for undirected: $n(n-1)/2$

Unless otherwise specified, no loops and only one directed edge from v_1 to v_2.

Example: Robot navigation.

Rooms connected together. Can do different things in different rooms. Adjacency defined by which states can transition to which other states. Searching this graph gives possible plans for getting from one state to another.

Adjacency lists: Use $|V|$ linked lists. Could use a hash table.

Directed graph only stores *outgoing* neighbors (although many people do store both for convenience in two different lists).

Adjacency matrix: $a_{ij} = 1$ if that edge exists. Can use for directed or undirected.

Powers of matrix are interesting: A^n represents whether there is a path of length n between i and j. Note: Only gives paths of length n, may be a path of length less than n.

Tradeoffs: e is the maximum number of edges originating at any vertex

Data Structure	Add edge	Adjacency checking	Visit all neighbors	Remove edge
List	$O(1)$	$O(e)$	$O(e)$	$O(e)$
Matrix	$O(1)$	$O(1)$	$O(n)$	$O(1)$

Also possible to represent graph *implicitly*: Just implement Adj(u) that returns all neighbors of u.

Example: Infinite grid. Squares are adjacent to top, bottom, left, right. Impossible to write down adjacency matrix, easy to implement Adj.