Lower Bound on Comparison Sorts

Key Idea: Decision Trees. Each vertex of the decision tree is a decision for the algorithm to make, so the height of the tree is the worst case running time. A tree with n vertices has height $\Omega(\lg n)$.

Search Lower Bound

Searching for an element in a sorted array of size n: search decision tree has $\geq n$ leaves (the element could be at index 0, 1, ... or $n-1$). Thus, the height is $\Omega(\lg n)$.

Sorting Lower Bound

There are $n!$ orders the array could come in, each of which is a leaf of the decision tree, so the height of the decision tree is $\Omega(\lg(n!))$

$$\begin{aligned}
\lg(1 * 2 * 3 * ... * (n-1) * n) &= \lg(1) + \lg(2) + \lg(3) + ... + \lg(n-1) + \lg(n) \\
&= \sum_{i=1}^{n} \lg(i) \\
&\geq \sum_{i=n/2}^{n} \lg(i) \\
&\geq \sum_{i=n/2}^{n} \lg(n/2) \\
&= \sum_{i=n/2}^{n} (\lg(n) - 1) \\
&= (\frac{n}{2}) \lg(n) - \frac{n}{2} \\
&= \Omega(n \log n)
\end{aligned}$$

Sort Stability

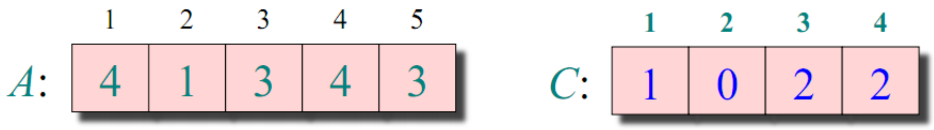
A sorting algorithm is **stable** if elements with the same key appear in the output array in the same order as they do in the input array. That is, it breaks ties between two elements by the rule that whichever element appears first in the input array appears first in the output array. Normally, the property of stability is important when satellite data are carried around with the element being sorted. For example, in order for radix sort to work correctly, the digit sorts must be stable.

Counting Sort

Counting sort is an algorithm that takes an array A of n elements with keys in the range {1, 2, ..., k} and sorts the array in $O(n + k)$ time. It is a stable sort.

In the lecture, we have seen one implementation of counting sort. Here we will show another one mentioned in the textbook (CLRS).

Intuition: Count key occurrences using an auxiliary array C with k elements, all initialized to 0. We make one pass through the input array A, and for each element i in A that we see, we increment $C[i]$ by 1. After we iterate through the n elements of A and update C, the value at index j of C corresponds to how many times j appeared in A. This step takes $O(n)$ time to iterate through A.

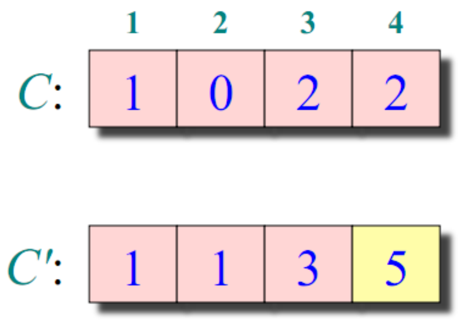


Once we have C, we can construct the sorted version of A by iterating through C and inserting each element j a total of $C[j]$ times into a new list (or A itself). Iterating through C takes $O(k)$ time.

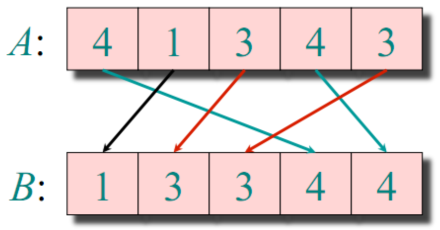
The end result is a sorted A and in total it took $O(n + k)$ time to do so.

However this does not permute the elements in A into a sorted list and is **not stable yet**. If A had two 3s for example, there's no distinction which 3 mapped to which 3 in the sorted result. We just counted two 3s and arbitrarily stuck two 3s in the sorted list. This is perfectly fine in many cases, but you'll see later on in radix sort why in some cases it is preferable to be able to provide a permutation that transforms A into a sorted version of itself.

Make it stable: We continue from the point where C is an array where $C[j]$ refers to how many times j appears in A. We transform C to an array where $C[j]$ refers to how many elements are $\leq j$. We do this by iterating through C and adding the value at the previous index to the value at the current index, since the number of elements $\leq j$ is equal to the number of elements $\leq j - 1$ (i.e. the value at the previous index) plus the number of elements that equal j (i.e. the value at the current index). The final result is an array C where the value of $C[j]$ is the number of elements $\leq j$ in A.



Now we iterate through A backwards starting from the last element of A. For each element i we see, we check $C[i]$ to find out how many elements are there $\leq i$. From this information, we know exactly where we can put i in the sorted array. Once we insert i into the sorted array, we decrement $C[i]$ so that if we see a duplicate element, we know that we have to insert it right before the previous i. Once we finish iterating through A, we will get a sorted list as before. This time, we provided a mapping from each element A to the sorted list. Note that since we iterated through A backwards and decrement $C[i]$ every time we see i. we preserve the order of duplicates in A. That is, if there are two 3s in A, we map the first 3 to an index before the second 3. This now makes counting sort **stable**. We will need the stability of counting sort when we use radix sort.



Iterating through C to change $C[j]$ from being the number of times j is found in A to being the number of times an element $\leq j$ is found in A takes $O(k)$ time. Iterating through A to map the elements of A to the sorted list takes $O(n)$ time. Since filling up C to begin with also took $O(n)$ time, the total runtime of this stable version of counting sort is $O(n + k + n)$ = $O(2n + k)$ = $O(n + k)$.

Radix Sort

Example:

Start	After one step	After two steps	After three steps	Sorted
2341 (A)	2341 (A)	2413 (B)	2143 (E)	1243 (D)
2413 (B)	1432 (C)	2413 (F)	1243 (D)	1432 (C)
1432 (C)	2413 (B)	1432 (C)	2341 (A)	2143 (E)
1243 (D)	1243 (D)	2341 (A)	2413 (B)	2341 (A)
2143 (E)	2143 (E)	1243 (D)	2413 (F)	2413 (B)
2413 (F)	2413 (F)	2143 (E)	1432 (C)	2413 (F)

Radix sort doesn't compare elements. It needs the elements it's sorting to be written as strings of characters that are individually comparable, like digits or letters. Starting from the last digit, it sorts the whole list by last digit (any stable sort will give a valid but possibly slow algorithm; we'll use counting sort), then the whole list by next-last digit, and so on until it sorts the whole list by first digit.

This is a valid sort: if $x < y$, then in the first digit where x and y differ, x has a smaller value at that digit than y, so x gets sorted to before y at that step and thereafter stays before it because the remaining sorts are stable. Also, if $x = y$, then x and y maintain their relative order throughout because all the sorts are stable.

Equivalently, we can prove by induction the statement that after k steps, the elements are sorted by the suffix consisting of their last k elements.

Other algorithms' sorting times are measured in comparisons of whole elements; radix sort's time has to be measured in digit comparisons or swaps. If there are n words of length d in an alphabet of size b, then each counting sort takes time $\Theta(n+b)$, for a total time of $\Theta((n+b)d)$. Another common form of that bound: if there are $k = b^d$ possible words, and $b = n$, and $k \leq n^c$, then that's $O(nc)$.

Is Heap Sort Stable?

No. Suppose we use heap sort on the list $\{1a, 1b\}$. When we construct a heap, we put $1a$ on top, so it's the first thing popped from the heap, and so goes at the end of the sorted array.

Is Merge Sort Stable?

It depends on the implementation.

Is the merge sort code given below stable?

```
 1  def merge_sort(A):
 2    n = len(A)
 3    if n==1:
 4      return A
 5    mid = n//2
 6    L = merge_sort(A[:mid])
 7    R = merge_sort(A[mid:])
 8    return merge(L,R)
 9
10  def merge(L,R):
11    i = 0
12    j = 0
13    answer = []
14    while i<len(L) and j<len(R):
15      if L[i]<R[j]:
16        answer.append(L[i])
17        i += 1
18      else:
19        answer.append(R[j])
20        j += 1
21    if i<len(L):
22      answer.extend(L[i:])
23    if j<len(R):
24      answer.extend(R[j:])
25    return answer
```

No, due to the comparison at line 15. If two elements are equal, the element on the right will be put first in the merged array which changes the original ordering. If we change it to `L[i]≤R[j]`, it will be stable.