

Non-Comparison Based Sorting

How fast can we sort?

Insertion-Sort $O(n^2)$

Merge-Sort, Quicksort (expected), Heapsort : $\Theta(n \lg n)$

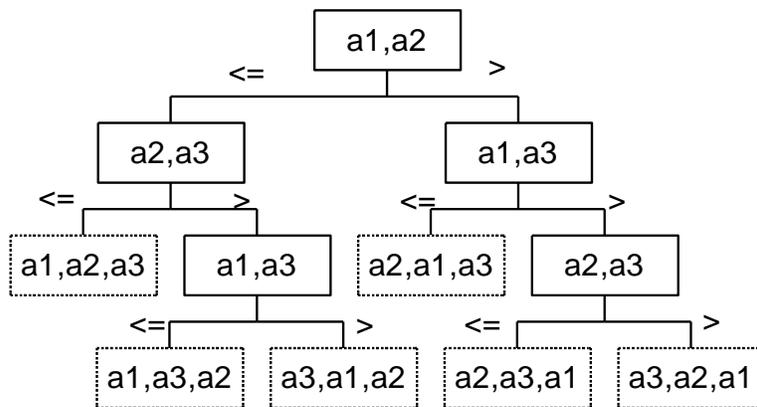
Can we do faster? What is the theoretical best we can do?

So far we have done comparison sorts: A sort based only on comparisons between input elements. $E1 < E2$, $E1 = E2$, $E1 > E2$. We will show that any comparison-based sort MUST make $\Omega(n \lg n)$ comparisons. This means that merge sort and heap sort are optimal.

This is important because it is not always possible that you can prove that your algorithm is the best one possible for a problem!

A decision tree is used to represent the comparisons of a sorting algorithm. Assume that all inputs are distinct. A decision tree compares all possible inputs to each other to determine the sequence of outputs.

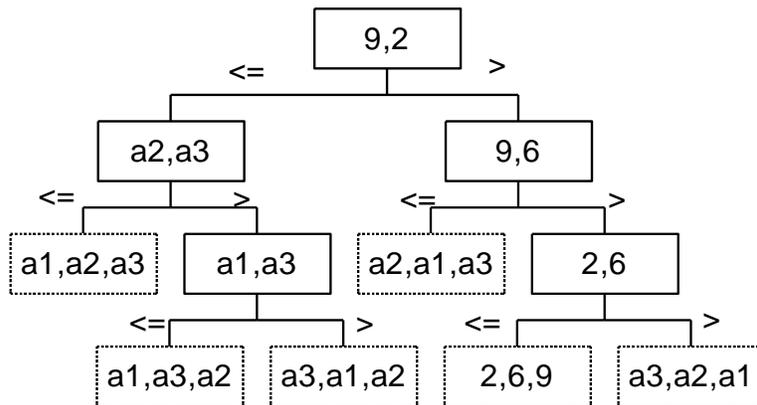
Decision Tree for three elements a_1, a_2, a_3 : If at the root, $a_1 \leq a_2$ go left and compare a_2 to a_3 , otherwise go right and compare a_1 to a_3 . Each path represents a different ordering on a_1, a_2, a_3 .



This type of decision tree will have $n!$ leaves – one for each permutation of the input.

Any comparison-based sorting algorithm will have to go through the steps in the decision tree as a minimum (can do more comparisons if we want to, of course!)

Example of 9,2,6 :



The sorted elements are 2,6,9, in order of a2,a3,a1.

Decision trees can model comparison sorts. For any sorting algorithm:

1. One tree for each input length n
2. An algorithm “splits” at each decision/comparison unwinding the actual execution into a tree path
3. The tree is all possible execution traces

What is the height of the decision tree? This gives us the minimum number of comparisons necessary to sort the input.

For n inputs, the tree must have $n!$ leaves. A binary tree of height h has no more than 2^h leaves:

$$n! \leq 2^h$$

Take log:

$$\lg(n!) \leq h$$

Stirling’s approximation says that $n! > \sqrt{2n\pi} (n/e)^n > (n/e)^n$

$$\lg(n/e)^n \leq h$$

So: $n \lg(n/e) \leq h$

$$n(\lg n - \lg e) \leq h$$

$$n \lg n - n \lg e \leq h$$

This means $h = \Omega(n \lg n)$ and we are DONE! We need to do at least $n \lg n$ comparisons to reach the bottom of the tree.

Does this mean that we can't do any better?? NO! (well, in some cases)
We can actually do some types of sorting in LINEAR TIME.

Counting Sort

This may work in $O(n)$ time. How? Because it uses no comparisons! But we have to make assumptions about the size and nature of the input.

Input: $A[1..n]$ where $A[i] \in \{1..k\}$

Output: $B[1..n]$, sorted

Uses: $C[1..k]$ auxiliary storage

Idea: Using random access array, count up number of times each input element appears and then collect them together.

Algorithm:

```
Count-Sort(A,n)
  for i ← 1 to k do C[i] ← 0           ; Initialize to 0
  for j ← 1 to n do C[A[j]] ++       ; Count
  j ← 1
  for i ← 1 to k do
    if (C[i]>0) then
      for z ← 1 to C[i] do
        B[j]=i
        j++
```

Ex: $A=[1\ 5\ 3\ 2\ 2\ 4\ 9]$
 $C=[0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$
 $C=[1\ 2\ 1\ 1\ 1\ 0\ 0\ 0\ 1]$
 $B=[1\ 2\ 2\ 3\ 4\ 5\ 9]$

This works! How long does it take? $O(n+k)$. If $k=n$, then this runs in $O(n)$ time. However, a bad example would be a input list like $A[1,2,999999999]$.

One disadvantage of the current algorithm: it is not **stable**

An algorithm is stable if the occurrences of a value I appear in the same order in the output as they do in the input. That is, ties between two numbers are broken by the rule that whichever number appears first in the input array appears first in the output array.

Why do we want a stable algorithm? If the thing we are sorting is just a key of a record (perhaps a zip code, or a job indicating priority where we want the first one in to have precedence) then stability may be important.

Ex: $A[3\ 5a\ 9\ 2\ 4\ 5b\ 6]$
Sorts to $A[2\ 3\ 4\ 5a\ 5b\ 6\ 9]$

and not to A[2 3 4 5b 5a 6 9]

Can modify algorithm to make it stable:

```

Stable-Count-Sort(A,n)
  for i ← 1 to k do C[i] ← 0           ; Initialize to 0
  for j ← 1 to n do C[A[j]] ++       ; Count
  for I ← 2 to k do
    C[i] ← C[i]+C[i-1]               ; Sum elements so far
                                     ; C[I] contains num elements ≤ I
  for j ← n downto 1 do
    B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]]-1

```

Example: A=[1 5 3 2 2 4 9]
 C=[0 0 0 0 0 0 0 0]
 C=[1 2 1 1 1 0 0 0 1]
 C=[1 3 4 5 6 6 6 6 7]
 B=[. 9]
 C=[1 3 4 5 6 6 6 6 6]
 B=[. . . . 4 . 9]
 ...
 B=[1 2 2 3 4 5 9]

This version is stable, since we fetch from the original array.

Radix Sort

Works like the punch-card readers of the early 1900's. Only works on input items with digits!

Idea somewhat counterintuitive: Sort on the least significant digit first.

```

Radix-Sort(A,d,n)           ; A is an n element array, each element d digits long
  for i ← 1 to d
    Use a stable sort to sort array A on digit i

```

Example:

A				
492		031		102
299		492		204
102		102		031
031	→	204	→	835
996		835		492
204		996		996
835		299		299
				031
				102
				204
				299
				492
				835
				996

Sort must be stable so numbers chosen in the correct order! Assumes that lower order digits are already sorted to work.

If each digit is not large, counting sort is a good choice to use for the sort method. If k is the maximum value of the digit, then counting sort takes $\Theta(k + n)$ time for one pass. We have to make d passes, so the total runtime is $\Theta(dk + dn)$.

If d is a constant and k is smaller than $O(n)$, Radix-Sort runs in $O(n)$ linear time!

Radix or Counting sorts are simple to code and the method of choice if the input is of the right form.

Bucket Sort

Similar to count sort, but uses a “bucket” to hold a range of inputs. Works for real numbers!

Like the other sorts, bucket sort is fast because it assumes something about the input:

1. Input is randomly generated
2. Input elements randomly distributed over the interval $[0..1]$. In many cases we can divide by some “max” value to force the input key for comparison to be between 0 and 1. This assumption means that elements are generated with uniform probability over $[0..1]$ or that each element has the same likelihood of being generated.

Idea:

1. Divide $[0..1]$ into n equal sized parts or “buckets”
2. Put each of the n inputs into one of the buckets. Some buckets may be empty and some may have more than 1 element.
3. Sort each bucket.
4. To produce output, go through the buckets in order, listing the elements in each.

Linked Lists is a good mechanism for storing the buckets.

```
Bucket-Sort(A,n)
for i ← 1 to n do
    Insert A[i] into list B[nA[i]]
for i ← 0 to n-1 do
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ... B[n-1] together in order
```

Buckets are automatically numbered in this case from $0..n-1$

All the lines but line 5 take $O(n)$ time in the worst case.

Line 5 is insertion sort which takes $O(n^2)$ time but since the input is generated uniformly we don't expect any bucket to have many elements in it so Insertion-Sort should be called on very small lists.

Example:

$A=[0.44 \ 0.12 \ 0.73 \ 0.29 \ 0.67 \ 0.49]$

Bucket I will get the values between I/n and $(I+1)/n$ since buckets are numbered from 0 to $n-1$.

B

0..0.16	→ 0.12
0.16..0.33	→ 0.29
0.33..0.50	→ 0.44 → 0.49
0.50..0.66	→
0.66..0.83	→ 0.73 → 0.67
0.83..1	→

Sort the buckets with insertion sort and then combine buckets to get:

0.12 0.29 0.44 0.49 0.67 0.73

Informal Argument on the average time:

Since any element in A comes from $[0..1]$ with an equal probability then the probability that an element e is in bucket $B[I]$ is $1/n$ (each bucket covers $1/n$ of $[0..1]$).

This means that the average number of elements that end up in bucket $B[I]$ is 1. There is a little more to the analysis than this, but the basic idea is that the distribution of the input will cause the calls to Insertion-Sort to be on very short lists and so the other steps in the algorithm will use more time. The average running time of Bucket-Sort is then $T(n)=O(n)$.