Chapter 8 - Sorting in Linear Time [Cormen 2001]

- We have now seen several algorithms that can sort n number in $O(n \lg n)$ time

- Q: Do you remember which ones?
  - In the worst-case scenario { MergeSort, HeapSort
  - In the average-case scenario: Quicksort
    (Recall that in the worst-case scenario Quicksort is $\Theta(n^2)$)

- These algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements

- These algorithms are called comparison sorts

8.1 - Lower bounds for sorting

- Comparison sort model:

  Only uses comparisons between ~~models~~ elements to gain order information about an input sequence $\langle a_1, a_2, \cdots, a_n \rangle$

- That is, given two elements a_i and a_j we perform one of the tests:
  - $a_i < a_j$
  - $a_i \leq a_j$
  - $a_i = a_j$ (this one is in fact useless)
  - $a_i \geq a_j$
  - $a_j > a_j$

  These can all be reduced to comparisons of the form $a_i \leq a_j$

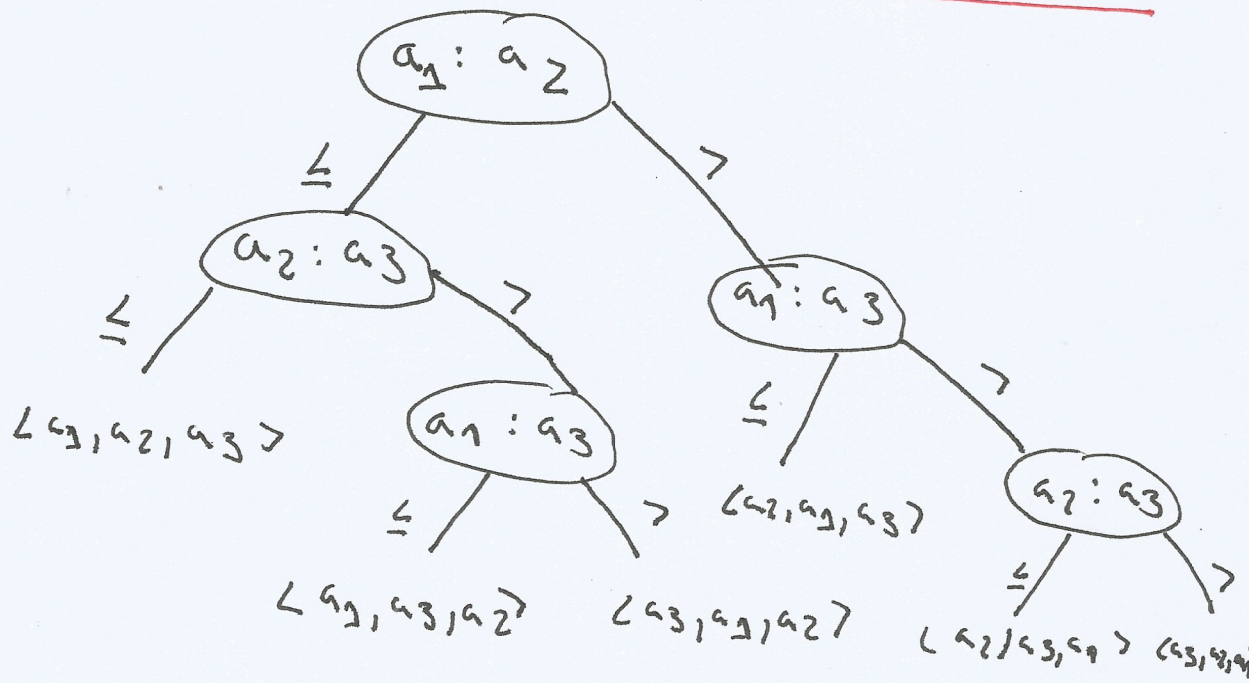
- We may not inspect the values of the elements or gain order information about them in any other way

8.1.1 - The decision-tree model

- All comparison sorts can be viewed abstractly in terms of decision trees
- 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.
- Control, data movement and all other aspects of the algorith... are ignored
- In a decision tree, each internal node is annotated by $i:j$ for some i and j in the range $1 \leq i, j \leq n$ (the array A to be sorted goes from index 1 to n)

- Each leaf is annotated by a permutation $\pi(1), \pi(2), \ldots, \pi(n)$
  - What is a permutation?
    - In this case it corresponds to a change of the elements of the array that will result in the sorted version of the array.
- The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf.
- At each internal node a comparison is made: $a_i \leq a_j$

Example: InsertionSort on an array of three elements



- When we come to a leaf, the sorting algorithm has established the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$

- Any sorting algorithm must be able to produce each permutation of its input

Question: How many permutations exist?

For an array of size 2: $\langle a_1, a_2 \rangle$ $\langle a_2, a_1 \rangle$

For an array of size 3: $\langle a_1, a_2, a_3 \rangle$, $\langle a_1, a_3, a_2 \rangle$
$\langle a_3, a_1, a_2 \rangle$ $\langle a_2, a_1, a_3 \rangle$
$\langle a_2, a_3, a_1 \rangle$ $\langle a_3, a_2, a_1 \rangle$

For any array:

$\langle$ ___ , ___ , ... , ___ $\rangle$

We can choose from n elements

We can choose from $n-1$ elements

We can choose from 1 element

$\therefore n!$ possible permutations

- This means that the decision tree must have at least $n!$ leafs

8.1.2 A lower bound for the worst case

- Length of the longest path from the root of a decision tree to any of its leaves represents the worst-case number of comparisons

- Consequently, the worst-case number of comparisons for a comparison sort equals the height of its decision tree.
- A lower bound on the heights of all decision trees is therefore a lower bound on the running time of any comparison sort algorithm.
- **Question:**

  How can we calculate such a lower bound?

Well we know that:

- Minimum number of leaves should be the total
- Maximum number of leaves for a tree of height h is 2^h
- I.e. $n! \leq l \leq 2^h$ where l is the number of leaves
- By taking logarithms on both sides:

$$\log(n!) \leq \log(2^h)$$

$$\Leftrightarrow \quad h \geq \log(n!)$$

- $\log(n!) = \log[n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1]$

$$= \log(n) + \log(n-1) + \log(n-2) + \ldots + \log 1$$

$$= \sum_{k=1}^{n} \log(k)$$

$$\ln n! = \sum_{k=1}^{n} \ln k$$

Notes:

$$\int u'v = uv - \int uv'$$

(for large n) $\approx \int_1^n \ln u \, du = \int_1^n \underbrace{1}_{u'} \cdot \underbrace{\ln u}_{v} \, du = \underbrace{u}_{u} \underbrace{\ln u}_{v} - \int_1^n u \cdot \frac{1}{u} \, du$

$$= \left[u \ln u - \int_1^n 1 \, du \right]_1^n = \left[u \ln u - u \right]_1^n = n \ln n - n - (1 \cdot \ln 1 - 1)$$

$$= n \ln n - n + 1$$

$\therefore_1 \log n! = n \ln n - n + 1$

$\therefore_2 h \geq \log(n!) = n \log n - n + 1 \Rightarrow \Omega(n \log n)$

$\therefore_3$ **Theorem 8.1** - Any comparison sort requires $\Omega(n \log n)$ comparisons in the worst-case

$\therefore_4$ **Corollary 8.2:**

Heapsort and merge sort are asymptotically optimal comparison sorts

Why?:

Merge sort: $O(n \log n)$
Heap sort: $O(n \log n)$

> This matches the lower-bound $\Omega(n \log n)$

8.2 Counting Sort

- Assumes that each of the n input elements is an integer in the range 0 to k.
- When $k = O(n)$ then the sort runs in $\Theta(n)$ time
- Idea: 1) Determine, for each input element x, the number of elements less than x

  2) This information can be used to place element x directly into its position in the output array

  E.g.: If there are 17 elements less than x, then x belong in output position 18

  Note: This scheme must be modified slightly to handle the situation in which several elements have the same value, since we don't want to put them all in the same position

```
CountingSort(A, B, k){
    for c=0 : k {
        C[c] = 0  // Vector initialization
    }

    for j=1 : length[A]{
        C[A[j]] = C[A[j]]+1;
    }

    for c = 1 : k {
        C[c] = C[c] + C[c-1]
    }

    for j = length[A] : 1 {
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1;
    }
}
```

sorted array

original array

k: maximum value of the elements to be sorted

$\Theta(k)$

$\Theta(n)$

// C[i] now contains the number of elements equal to i

$\Theta(k)$

// C[i] now contains the number of elements less than or equal to i

// We need to start from the end

$\Theta(n)$

Example:

	1	2	3	4	5	6	7	8
A =	2	5	3	0	2	3	0	3

, $k = 5$ ← Maximum element

1st step C =

	0	1	2	3	4	5
	2	0	2	3	0	1

→ (calculate the number of elements equal to A[i], ∀i)

2nd step C =

	0	1	2	3	4	5
	2	2	4	7	7	8

→ (calculate the number of elements less than or equal to A[i], ∀i)

3rd step

A[8]=3, B =

	1	2	3	4	5	6	7	8
							3	

C =

	0	1	2	3	4	5
	2	2	4	6	7	8

4th step:

A[7] = 0

B =

	1	2	3	4	5	6	7	8
		0					3	

C =

	0	1	2	3	4	5
	1	2	4	6	7	8

5th step:

A[6] = 3

B =

	1	2	3	4	5	6	7	8
		0				3	3	

C =

	0	1	2	3	4	5
	1	2	4	5	7	8

6th step:

A[5] = 2

B =

	1	2	3	4	5	6	7	8
		0		2		3	3	

C =

	0	1	2	3	4	5
	1	2	3	5	7	8

7th step:

A[4] = 1

B =

1	2	3	4	5	6	7	8
1	1		2		3	3	

C =

0	1	2	3	4	5
1	2	3	5	7	8

8th step:

A[3] = 3

B =

1	2	3	4	5	6	7	8
1	1		2	3	3	3	

C =

0	1	2	3	4	5
1	2	3	4	7	8

9th step:

A[2] = 5

B =

1	2	3	4	5	6	7	8
1	1		2	3	3	3	5

C =

0	1	2	3	4	5
1	2	3	4	7	7

10th step:

A[1] = 2

B =

1	2	3	4	5	6	7	8
1	1	2	2	3	3	3	5

C =

0	1	2	3	4	5
1	2	2	4	7	7

Question:

How much time does counting sort require?

(Please refer to page 8)

- Total time: $\Theta(k) + \Theta(n) + \Theta(k) + \Theta(n)$

  $= \Theta(k+n)$

- What does this mean?
  - Notice that if $k = O(n)$ (read as: k is order n)
    then $\Theta(k+n) = \Theta(n+n) = \Theta(n)$

- Counting sort beats the $\Omega(n \lg n)$ time of comparison sort because no comparisons between input elements occur in the code. Instead, counting sort uses the actual values of the elements to index into an array.

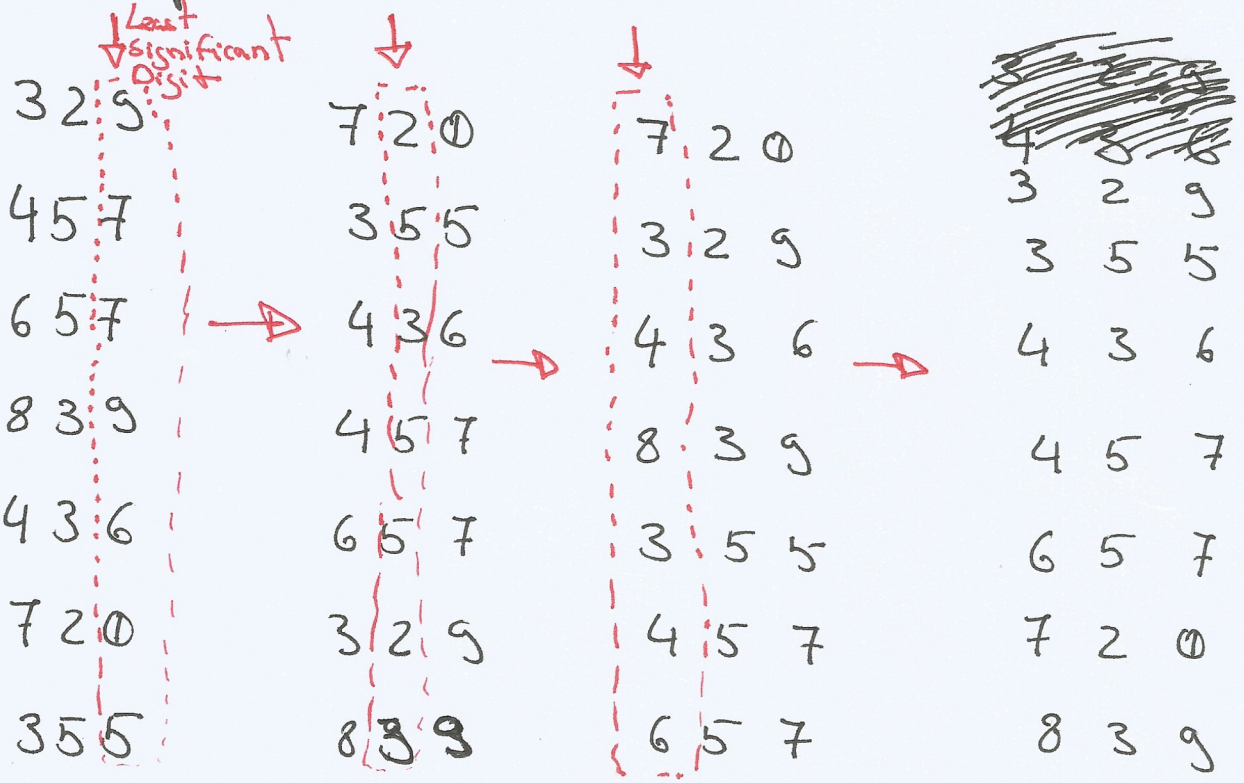
- Counting sort is also stable:

  Numbers with the same value appear in the output array in the same order as they do in the input array

  Now you ask "why is this important?" ↳ Stability is important when satellite data need to be carried around with the element being sorted.

8.3 – Radix Sort

- Sorts d-digit algorithms
- Solves the problem counterintuitively by sorting on the least significant digit first
- The numbers are sorted again on the second-least significant digit.
- The process continues until all d digits have been sorted.
- Example:



- Notice that the problem of sorting n number of d-digits requires a sorting algorithm that is stable, since we have (d-1) satellite data that needs to be moved around with the sorted element