Sorting

Kriengsak Treeprapin

Agenda

- Sorting Terminology
- Performance Criteria
- Standard Sorting Algorithm
- Linear Time Sorting Algorithm
- External Sorting
- Misc

Sorting

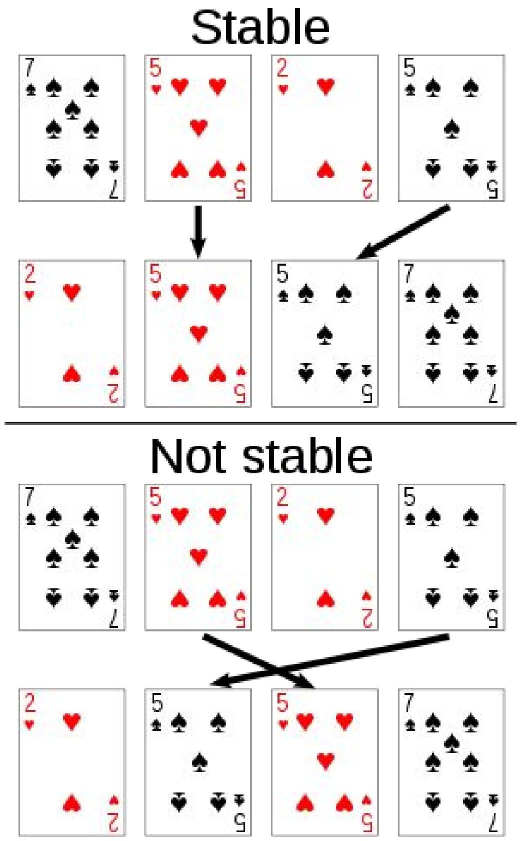
- เป็นกระบวนการสำหรับจัดเรียง (ordering) จัดกลุ่ม (categorizing) ข้อมูลเพื่อ
    - ค้นหาข้อมูลที่ระบุอย่างมีประสิทธิภาพ
    - รวมข้อมูลลำดับ (sequences) อย่างมีประสิทธิภาพ -> merge sort process
    - จัดลำดับการทำงานของกลุ่มข้อมูล

Sorting Terminology

- Inplace sorting
  - An in-place sorting algorithm uses constant extra space even for producing the output (modifies the given array only).
  - EX) Insertion Sort, Selection Sorts.
- Internal or External sorting
  - External Sorting is used for massive amount of data. Merge Sort and its variations are typically used for external sorting.
  - When all data is placed in-memory, then sorting is called internal sorting.

Sorting Terminology

- Stability
  - A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.
  - stability means that equivalent elements retain their relative positions, after sorting.



Performance Criteria

There are several criteria to be used in evaluating a sorting algorithm:

- Running time.
    - Typically, an elementary sorting algorithm requires $O(N^2)$ steps to sort N randomly arranged items.
    - More sophisticated sorting algorithms require O(N log N) steps on average.
    - Some sorting algorithms are more sensitive to the nature of the input than others. Quicksort, for example, requires O(N log N) time in the average case, but requires $O(N^2)$ time in the worst case.

Performance Criteria

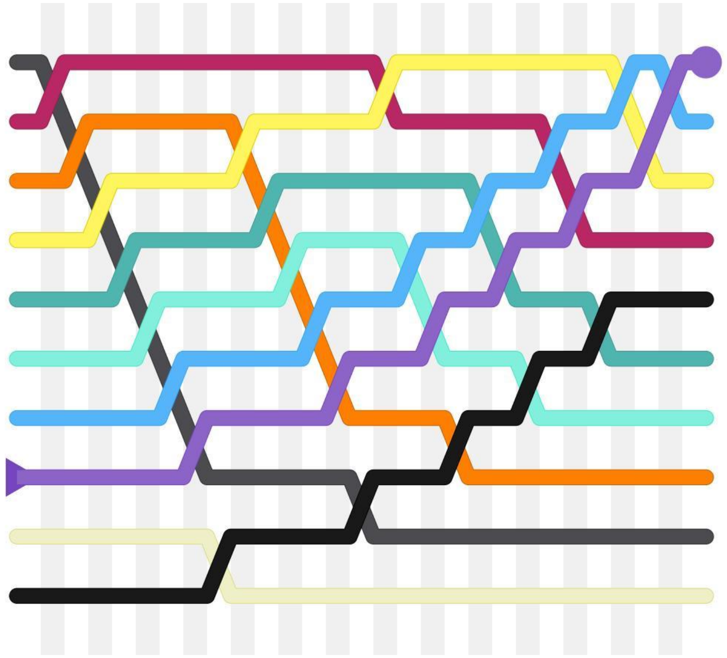
There are several criteria to be used in evaluating a sorting algorithm:

- **Memory requirements.**
  - In place sorting algorithms are the most memory efficient, since they require practically no additional memory.

- **Stability.**
  - The ability of a sorting algorithm to preserve the relative order of equal keys in a file.

Standard Sorting Algorithms

Bubble Sort



Bubble Sort

Works by repeatedly swapping the adjacent elements if they are in wrong order.

EX) First Pass:

[7, 1, 5, 3, 9]	->	[1, 7, 5, 3, 9]	swap!
[1, 7, 5, 3, 9]	->	[1, 5, 7, 3, 9]	swap!
[1, 5, 7, 3, 9]	->	[1, 5, 3, 7, 9]	swap!
[1, 5, 3, 7, 9]	->	[1, 5, 3, 7, 9]	

Bubble Sort

Works by repeatedly swapping the adjacent elements if they are in wrong order.

EX) Second Pass:

[1, 5, 3, 7, 9]	->	[1, 5, 3, 7, 9]	
[1, 5, 3, 7, 9]	->	[1, 3, 5, 7, 9]	swap!
[1, 3, 5, 7, 9]	->	[1, 3, 5, 7, 9]	
[1, 3, 5, 7, 9]	->	[1, 3, 5, 7, 9]	

The algorithm needs one whole pass without any swap to know it is sorted.

Bubble Sort

Works by repeatedly swapping the adjacent elements if they are in wrong order.

EX) Third Pass:

[1, 3, 5, 7, 9] -> [1, 3, 5, 7, 9]

[1, 3, 5, 7, 9] -> [1, 3, 5, 7, 9]

[1, 3, 5, 7, 9] -> [1, 3, 5, 7, 9]

[1, 3, 5, 7, 9] -> [1, 3, 5, 7, 9]

Try this : https://visualgo.net/en/sorting

Performance of Bubble Sort

Worst and Average Case Time Complexity : $O(N^2)$

Best Case Time Complexity : O(N)

Auxiliary Space : O(1)

Stable : yes

Selection Sort

Selection Sort

- Sorts an array by repeatedly finding the minimum element and putting it at the beginning of array.

a.	Select the lowest element in the remaining array	**Selection**
b.	Bring it to the starting position	**Swapping**
c.	Change the counter for unsorted array by one	**Shiftset**

Try this : https://visualgo.net/en/sorting

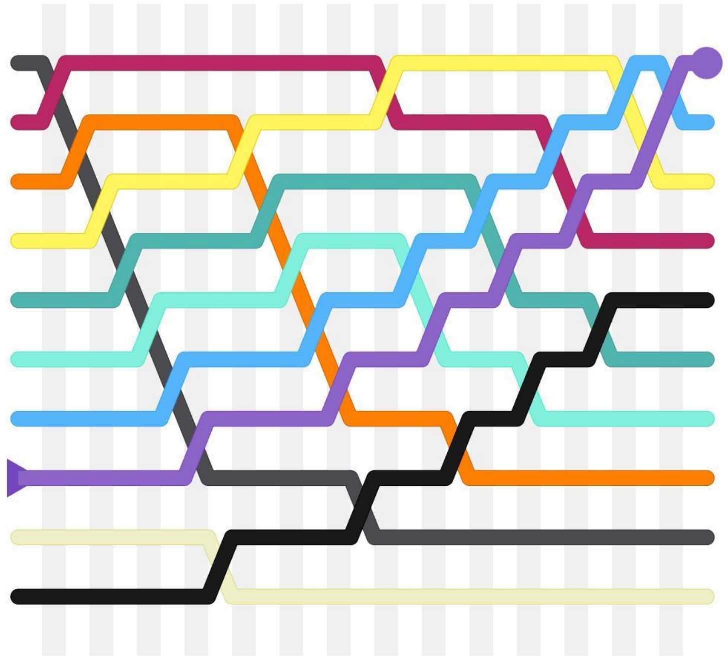
Performance of Selection Sort

Time Complexity : $O(N^2)$

Auxiliary Space : $O(1)$

Stable : no

Insertion Sort



Insertion Sort

At each iteration

- Removes one element from the input data
- Finds the location it belongs within the sorted list and insert it.
- Repeats until no input elements remain.

Try this : https://visualgo.net/en/sorting

Insertion Sort

- Ex)

rnd 1: [9, 7, 6, 16, 5]

rnd 2: [9, 7, 6, 16, 5] -> [7, 9, 6, 16, 5]

rnd 3: [7, 9, 6, 16, 5] -> [6, 7, 9, 16, 5]

rnd 4: [6, 7, 9, 16, 5]

rnd 5: [6, 7, 9, 16, 5] -> [5, 6, 7, 9, 16]

Performance of Insertion Sort

Time Complexity : $O(N^2)$

Auxiliary Space : $O(1)$

Stable : yes

Shell Sort

Shell Sort

An extension of insertion sort, which gains speed by allowing exchanges of non-adjacent elements.

Definition:

> An *h*-sorted file is one with the property that taking every *h*th element (starting anywhere) yields a sorted file.

Shell Sort

Algorithms:

- Choose an initial large step size, h_K, and use insertion sort to produce an h_K-sorted file.
- Choose a smaller step size, h_{K-1}, and use insertion sort to produce an h_{K-1}-sorted file, using the h_K-sorted file as input.
- Continue this process until done. The last stage uses insertion sort, with a step size $h_1 = 1$, to produce a sorted file.

Shell Sort

EX) gap sequence is N/2, N/4, … , $N/2^k$, …, 1

Input Data [9, 7, 6, 16, 13, 4, 32, 1, 5]

first round h_1 = round(9/2) = 4

```
[9, 7, 6, 16, 13, 4, 32, 1, 5]    ->    [9, 7, 6, 16, 13, 4, 32, 1, 5]
[9, 7, 6, 16, 13, 4, 32, 1, 5]    ->    [5, 4, 6, 16, 13, 7, 32, 1, 5]
[9, 4, 6, 16, 13, 7, 32, 1, 5]    ->    [5, 4, 6, 16, 13, 7, 32, 1, 5]
[9, 4, 6, 16, 13, 7, 32, 1, 5]    ->    [5, 4, 6, 1, 13, 7, 32, 16, 5]
[9, 4, 6, 1, 13, 7, 32, 16, 5]    ->    [5, 4, 6, 1, 9, 7, 32, 16, 13]
```

Shell Sort

EX) gap sequence is N/2, N/4, … , $N/2^k$, …, 1

second round h_2 = round(9/4) = 2

[5, 4, 6, 1, 9, 7, 32, 16, 13] -> [5, 4, 6, 1, 9, 7, 32, 16, 13]

[5, 4, 6, 1, 9, 7, 32, 16, 13] -> [5, 1, 6, 4, 9, 7, 32, 16, 13]

[5, 1, 6, 4, 9, 7, 32, 16, 13] -> [5, 1, 6, 4, 9, 7, 32, 16, 13]

[5, 1, 6, 4, 9, 7, 32, 16, 13] -> [5, 1, 6, 4, 9, 7, 32, 16, 13]

[5, 1, 6, 4, 9, 7, 32, 16, 13] -> [5, 1, 6, 4, 9, 7, 32, 16, 13]

[5, 1, 6, 4, 9, 7, 32, 16, 13] -> [5, 1, 6, 4, 9, 7, 32, 16, 13]

[5, 1, 6, 4, 9, 7, 32, 16, 13] -> [5, 1, 6, 4, 9, 7, 13, 16, 32]

Shell Sort

EX) gap sequence is N/2, N/4, … , $N/2^k$, …, 1

third round h_3 = round(9/8) = 1 => Insertion sort

[5, 1, 6, 4, 9, 7, 32, 16, 13] -> [1, 4, 5, 6, 7, 9, 13, 16, 32]

Performance of Shell Sort

Time Complexity : $O(N^2)$ ~ $O(N \log^2 N)$ depend on gap sequence

Worst and Average Time Complexity : $O(N^{1.5})$

Best Time Complexity : $O(N \log N)$ in Sorted Data

Auxiliary Space : $O(1)$

Stable : no

Gap sequence in Shell Sort

The performance of Shellsort depends on gap sequence. However, the question of deciding which gap sequence to use is difficult.

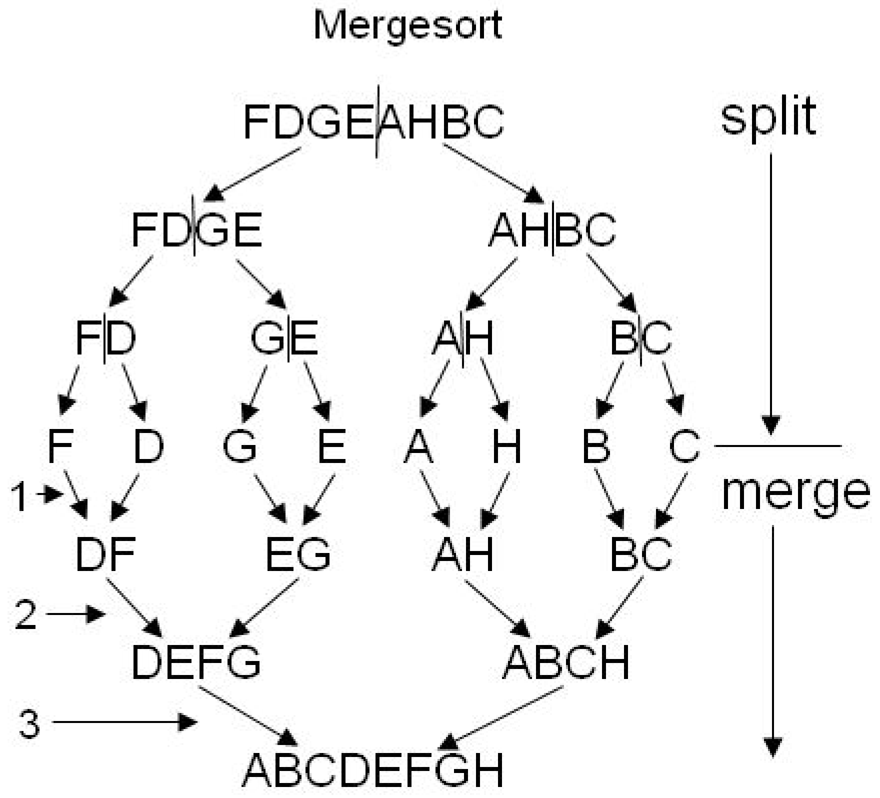
gap sequence 1 :	round($N/2^k$)	->	$\theta(N^2)$
gap sequence 2 :	$2^k - 1$	->	$\theta(N^{1.5})$
gap sequence 3 :	Successive numbers of the form 2^p3^q	->	$\theta(N\log^2 N)$
gap sequence 4 :	$4^{k+1} - 6*2^{(k+1)/2} + 1$	->	$\theta(N^{4/3})$

Merge Sort

Merge Sort

A Divide and Conquer algorithm. It divides (split) input array in two halves, calls itself for the two halves and then merges the two sorted halves.

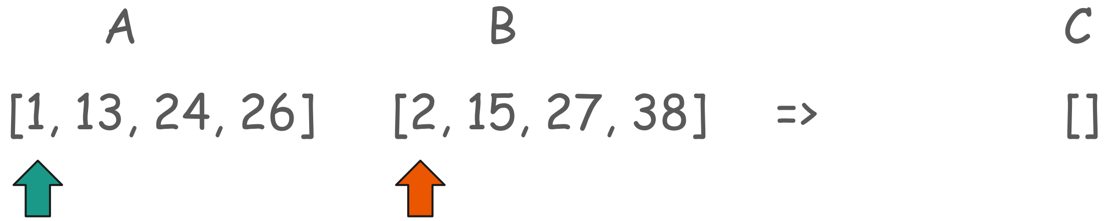
Try this : https://visualgo.net/en/sorting



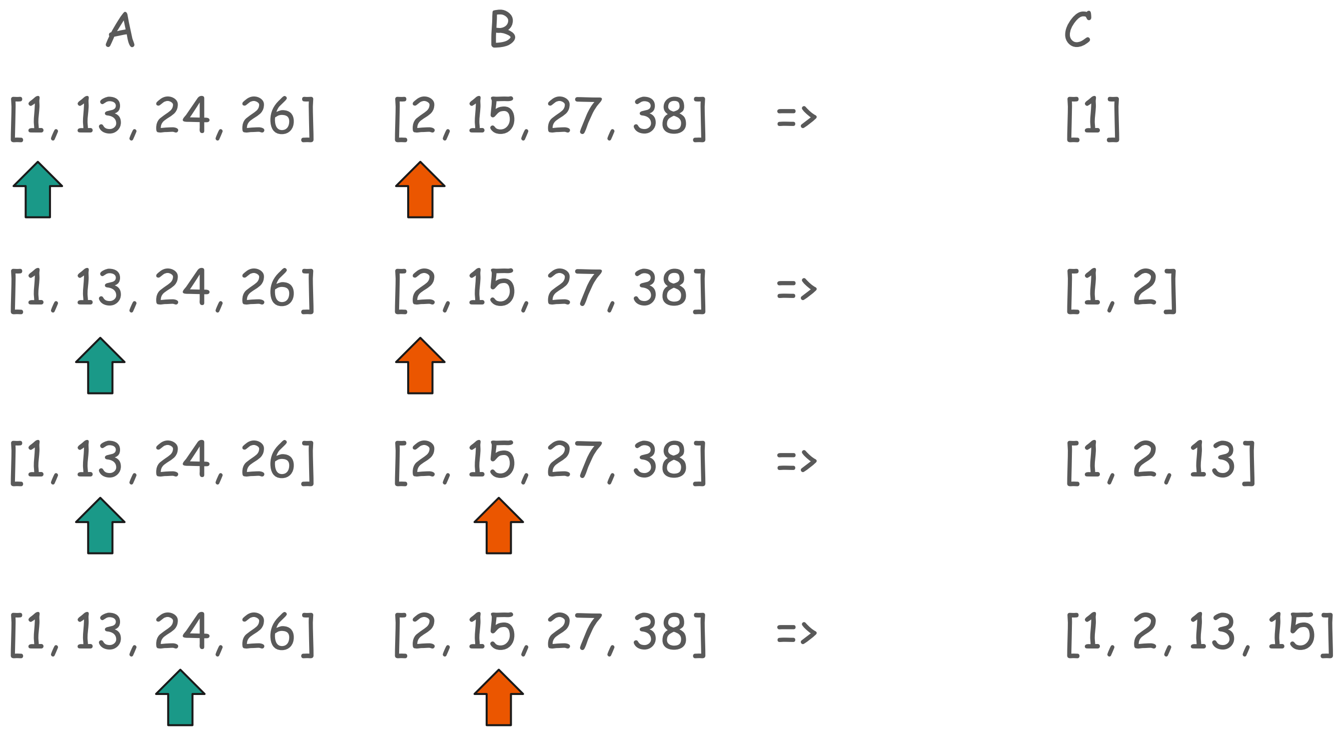
http://www.alwayslearn.com/Computer%20Algorithms

Implementation of Merge Sort

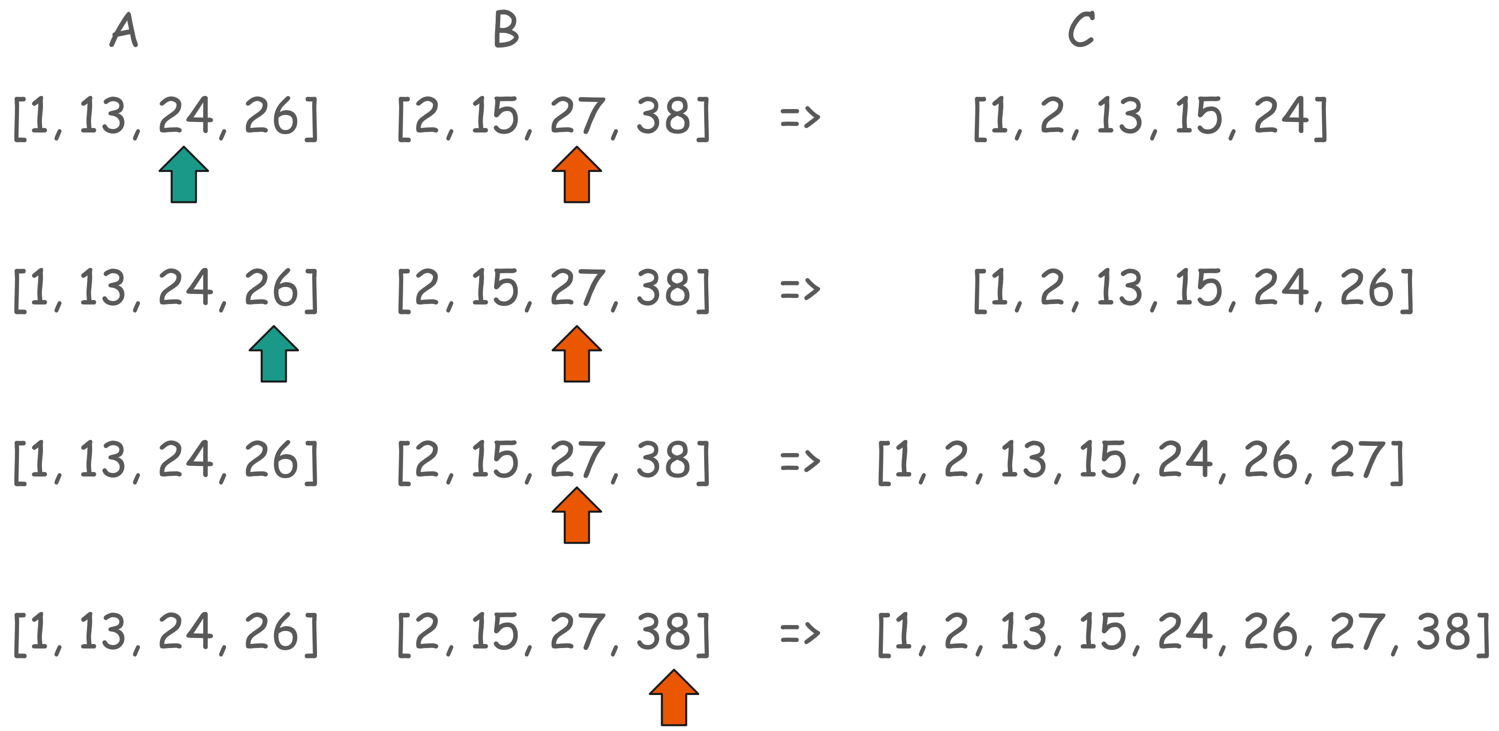
The fundamental operation in this algorithm is merging two sorted lists. Because the list are sorted, this can be done in one pass through the input.



Implementation of Merge Sort



Implementation of Merge Sort



Analysis of Merge Sort

For N = 1, the time to mergesort is constant.

$$T(1) = 1$$

Otherwise, the time to mergesort N numbers is equal to the time to do two recursive mergesorts of size N/2 plus the time to merge.

$$T(N) = 2T(N/2) + N$$

Analysis of Merge Sort

Divide the recurrence relation through by N.

$$T(N)/N = T(N/2)/(N/2) + 1$$

This equation is valid for any N that is a power of 2, thus

$$T(N/2)/(N/2) = T(N/4)/(N/4) + 1$$

and $$T(N/4)/(N/4) = T(N/8)/(N/8) + 1$$

… $$T(2)/2 = T(1)/1 + 1$$

Analysis of Merge Sort

Add up all the equations.

$$T(N)/N = T(1)/1 + \log N$$

Multiplying through by N gives the final answer.

$$T(N) = N + N \log N$$

$$= O(N \log N)$$

Performance of Merge Sort

Time Complexity : O(N log N)

Auxiliary Space : O(N)

Stable : yes

Merge Sort Challenge 1 : 3-way Merge Sort

Merge sort recursively breaks down the arrays to subarrays of size half. Similarly, 3-way Merge sort breaks down the arrays to subarrays of size one third.

Time Complexity : $O(N \log_3 N)$!???

Although by splitting array into 3 sub-arrays, we are decreasing fewer number of passes, we are actually increasing the cost of each pass by doing more number of comparisons.

Merge Sort Challenge 2 : In-place merge

Use Insertion Sort when merge array?

Time Complexity : $O(N^2)$!???

Use Quick Sort concept when merge array

Time Complexity : O(N log N) with Auxiliary Space : O(1)

Merge Sort Challenge 3 : Count Inversion

How far (or close) the array is from being sorted.

If array is already sorted then inversion count is 0.

If array is sorted in reverse order that inversion count is the maximum.

https://programming.in.th/task/rev2_problem.php?pid=1164

Quick Sort

Quick Sort

QuickSort is a Divide and Conquer algorithm.

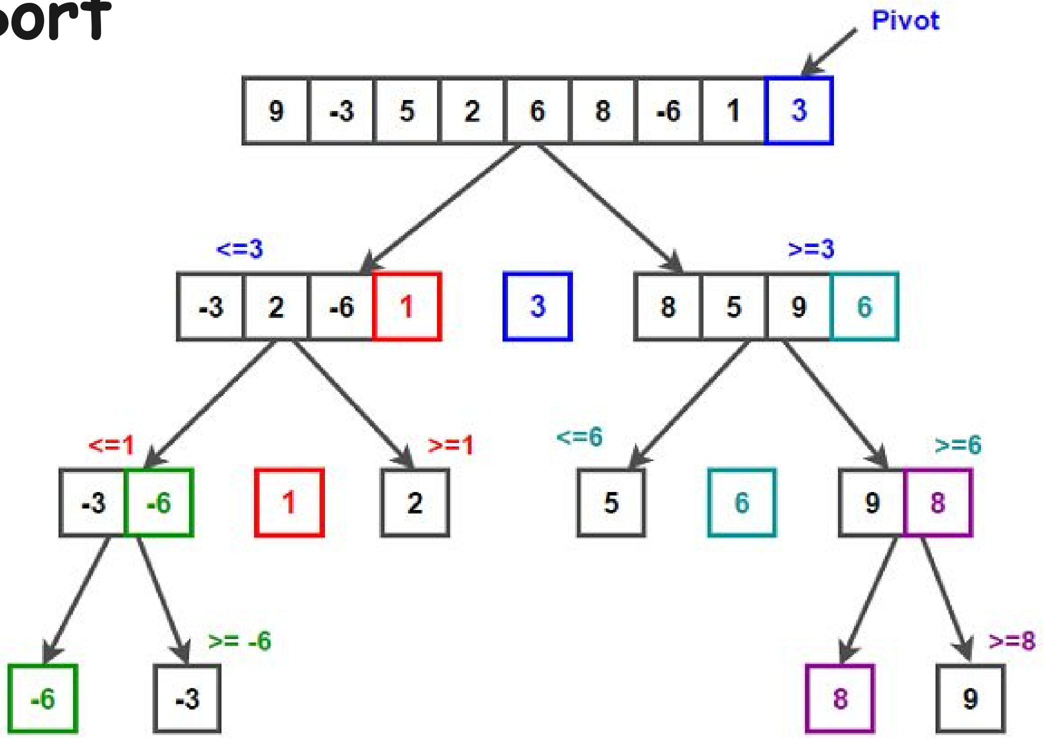
It picks an element as pivot and partitions the given array around the picked pivot.

Quick Sort

Approach :

- Partition the subarray a[left],a[left+1],...,a[right] into two parts, such that
  - element a[i] is in its final place in the array for some i in the interval [left,right].
  - none of the elements in a[left],a[left+1],...,a[i-1] are greater than a[i].
  - none of the elements in a[i+1],a[i+2],...,a[right] are less than a[i].
- Recursively partition the two subarrays, a[left],a[left+1],...,a[i-1] and a[i+1],a[i+2],...,a[right], until the entire array is sorted.

Quick Sort



Try this : https://visualgo.net/en/sorting

http://www.techiedelight.com/wp-content

Quick Sort

How to partition the subarray a[left],a[left+1],...,a[right] :

- Choose a[right] to be the element that will go into its final position.
- Scan from the left end of the subarray until an element greater than a[right] is found.
- Scan from the right end of the subarray until an element less than a[right] is found.

Quick Sort

How to partition the subarray a[left],a[left+1],...,a[right] :

- Exchange the two elements which stopped the scans.
- Continue the scans in this way. When the scan pointers cross we will have two new subarrays, one with elements less than a[right] and the other with elements greater than a[right].

Quick Sort

How to select pivot :

- Always pick first element as pivot.
- Always pick last element as pivot.
- Pick a random element as pivot.
- Pick median as pivot.

Poor partition in some cases!

Best choice! but hard to find median

Analysis of Quick Sort : Worst-Case

The pivot is the smallest element, all the time. The recurrence is

$$T(N) = T(N-1) + cN$$

$$T(N-1) = T(N-2) + cN$$

$$T(N-2) = T(N-3) + cN$$

$$\dots$$

$$T(2) = T(1) + cN$$

Adding up all these equations yields =>

$$T(N) = T(1) + c\Sigma_{i=2} i \quad = \quad \Theta(N^2)$$

Analysis of Quick Sort : Best-Case

The pivot is in the middle, all the time. The recurrence is

$$T(N) = 2T(N/2) + cN$$

$$T(N)/N = T(N/2)/(N/2) + c$$ Divide both side by N

$$T(N/2)/(N/2) = T(N/4)/(N/4) + c$$

…

$$T(2)/2 = T(1) + c$$

Adding up all these equations yields =>

$$T(N)/N = T(1)/1 + c \log N = \Theta(N \log N)$$

Analysis of Quick Sort : Average-case

Need to consider all possible permutation of array and calculate time taken by every permutation.

from

$T(N) = T(i) + T(N - i - 1) + cN$: i is the number of element

The average value of $T(i)$ is $(1/N)\Sigma_{j=0} T(j)$

Therefore

$$T(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} T(j)\right] + cN$$

Analysis of Quick Sort : Average-case

Then multiply by *N* it becomes

$$NT(N) = 2\left[\sum_{j=0}^{N-1} T(j)\right] + cN^2$$

Then telescope one more equation

$$(N-1)T(N-1) = 2\left[\sum_{j=0}^{N-2} T(j)\right] + c(N-1)^2$$

and subtract these two equations, we obtain

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN - c$$

Analysis of Quick Sort : Average-case

Rearrange equation and divide by $N(N+1)$

$$T(N)/(N+1) = T(N-1)/N + 2c/(N+1)$$

Telescope it

$$T(N-1)/N = T(N-2)/(N-1) + 2c/N$$

$$T(N-2)/(N-1) = T(N-3)/(N-2) + 2c/(N-1)$$

$$\ldots$$

$$T(2)/3 = T(1)/2 + 2c/3$$

Analysis of Quick Sort : Average-case

Adding all equations

$$\frac{T(N)}{N+1} = \frac{T(1)}{2} + 2c \sum_{i=3}^{N+1} \frac{1}{i}$$

The sum is about $\log_e(N+1) + \gamma - 3/2$, where $\gamma \approx 0.577$

$$T(N)/(N+1) = O(\log N)$$

And so

$$T(N) = O(N \log N)$$

Performance of Quick Sort

Best Case and Average Case Time Complexity : O(N log N)

Worst Case Time Complexity : $O(N^2)$

Auxiliary Space : O(log N)

Stable : no

Improvement of Quick Sort

Median of sample.

    Best choice of pivot.

    How to compute them in O(N) ?

Insertion sort for small files. Cutoff

    Decrease overhead of quicksort.

Optimize parameters.

    Median-of-3 random elements. => median of first, last and middle element

Quick Sort Challenge 1 : 3-way Quick Sort

It is used when arrays with large number of duplicate sort keys arise frequently.

An array arr[l..r] is divided in 3 parts:

a) arr[l..i] elements less than pivot.

b) arr[i+1..j-1] elements equal to pivot.

c) arr[j..r] elements greater than pivot.

Dutch National Flag (DNF) Algorithm

Quick Sort Challenge 2 : 2 pivot Quick Sort

Also called as Dual-Pivot Quicksort algorithm.

- Little bit faster than the original single pivot quicksort. But still, the worst case will remain O(N^2)
- Take two pivots. The left pivot must be less than or equal to the right pivot

Quick Sort Challenge 2 : 2 pivot Quick Sort

The algorithm follows following steps:

1. Partitioning the array into three parts:
   a. First part, all elements will be less than the left pivot.
   b. Second part all elements will be greater or equal to the left pivot and also will be less than or equal to the right pivot.
   c. Third part all elements will be greater than the right pivot.
2. Then, shift the two pivots to their appropriate positions, and after that begin quicksorting these three parts recursively.

Heap Sort

Heap Sort

A comparison based sorting technique based on Binary Heap data structure.

Binary Heap

A Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes.

Heap Sort

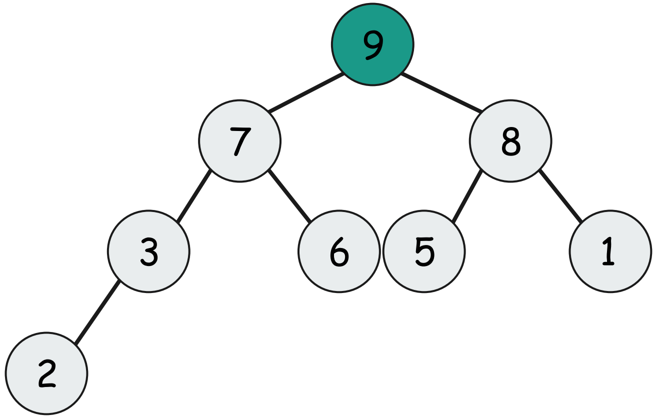
Algorithm :

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap
3. Reducing the size of heap by 1. and heapify the root of tree.
4. Repeat above steps while size of heap is greater than 1.

Heap Sort

Example :

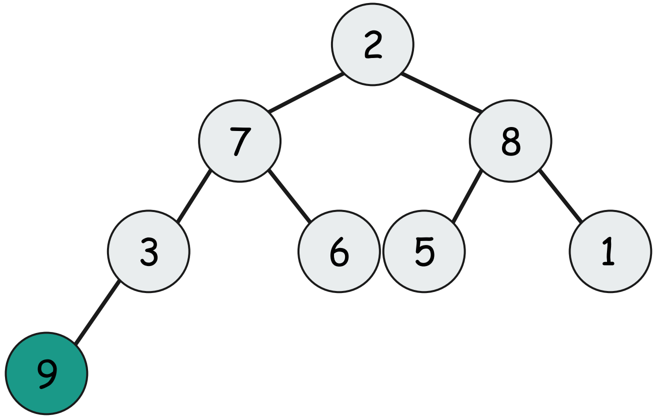
[9, 7, 5, 2, 6, 8, 1, 3] == make heap ==> [9, 7, 8, 3, 6, 5, 1, 2]



Heap Sort

Example :

[9, 7, 8, 3, 6, 5, 1, 2] == push heap ==> [2, 7, 8, 3, 6, 5, 1, 9]

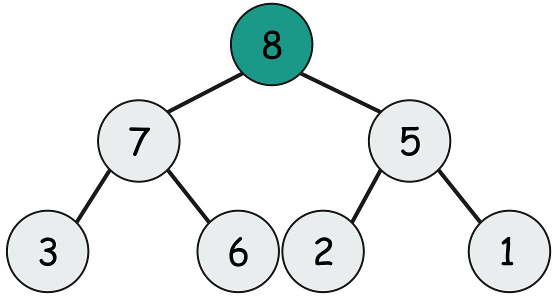


Heap Sort

Example :

remove heap size 1 and heapify

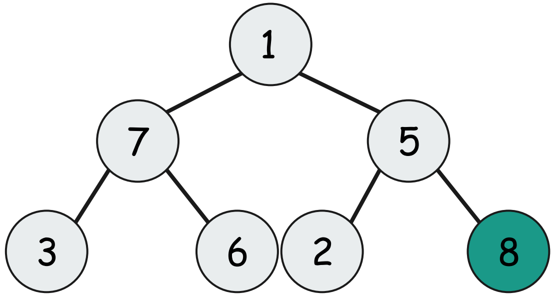
[2, 7, 8, 3, 6, 5, 1, 9] == heapify ==> [8, 7, 5, 3, 6, 2, 1, 9]



Heap Sort

Example :

[8, 7, 5, 3, 6, 2, 1, 9] == push heap ==> [1, 7, 5, 3, 6, 2, 8, 9]

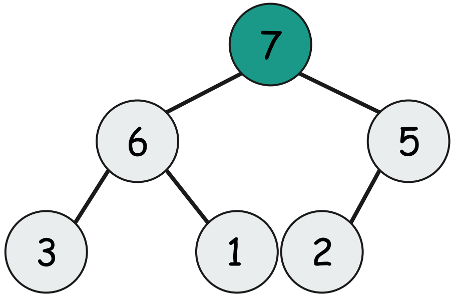


Heap Sort

Example :

remove heap size 1 and heapify

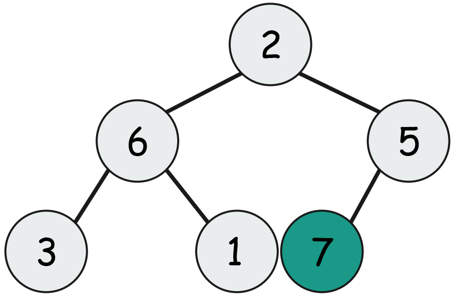
[1, 7, 5, 3, 6, 2, 8, 9] == heapify ==> [7, 6, 5, 3, 1, 2, 8, 9]



Heap Sort

Example :

[7, 6, 5, 3, 1, 2, 8, 9] == push heap ==> [2, 6, 5, 3, 1, 7, 8, 9]

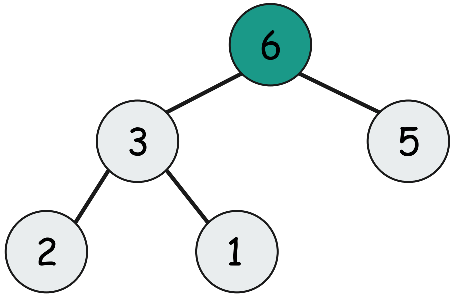


Heap Sort

Example :

remove heap size 1 and heapify

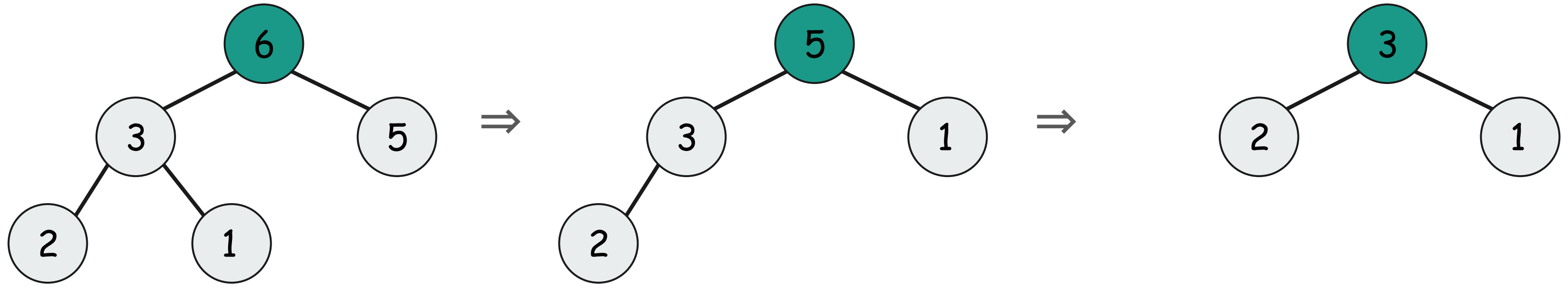
[2, 6, 5, 3, 1, 7, 8, 9] == heapify ==> [6, 3, 5, 2, 1, 7, 8, 9]



Heap Sort

Example :

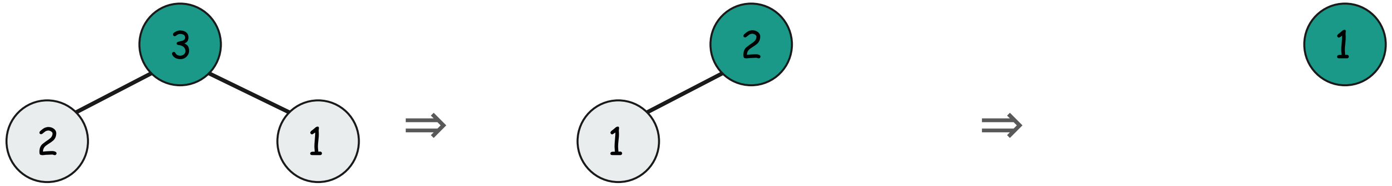
[6, 3, 5, 2, 1, 7, 8, 9] ⇒ [5, 3, 1, 2, 6, 7, 8, 9] ⇒ [3, 2, 1, 5, 6, 7, 8, 9]



Heap Sort

Example :

[3, 2, 1, 5, 6, 7, 8, 9] ⇒ [2, 1, 3, 5, 6, 7, 8, 9] ⇒ [1, 2, 3, 5, 6, 7, 8, 9]



Analysis of Heap Sort

Time Complexity for make heap : $O(N)$

Time Complexity for heapify : $O(\log N)$

Thus overall time complexity is $O(N) + O(N \log N) \Rightarrow O(N \log N)$

Performance of Heap Sort

Best Case Worst Case and Average Case Time Complexity : O(N log N)

Auxiliary Space : O(1)

Stable : no

Time Complexities of all Sorting algorithm

	Best	Average	Worst
Selection	$\Omega(N \log N)$	$\Theta(N^2)$	$O(N^2)$
Bubble	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$
Insertion	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$
Shell	$\Omega(N \log N)$	$\Theta(N (\log N)^2)$	$O(N (\log N)^2)$
Merge	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$
Quick	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N^2)$
Heap	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$

Lower bound for comparison based sorting algorithms

Although we have O(N log N) algorithms for sorting, it is not clear that this is as good as we can do.

prove the following result:

Any sorting algorithm that uses only comparisons requires $\lceil \log(N!) \rceil$ comparisons in the worst case and $\log(N!)$ comparisons on average.

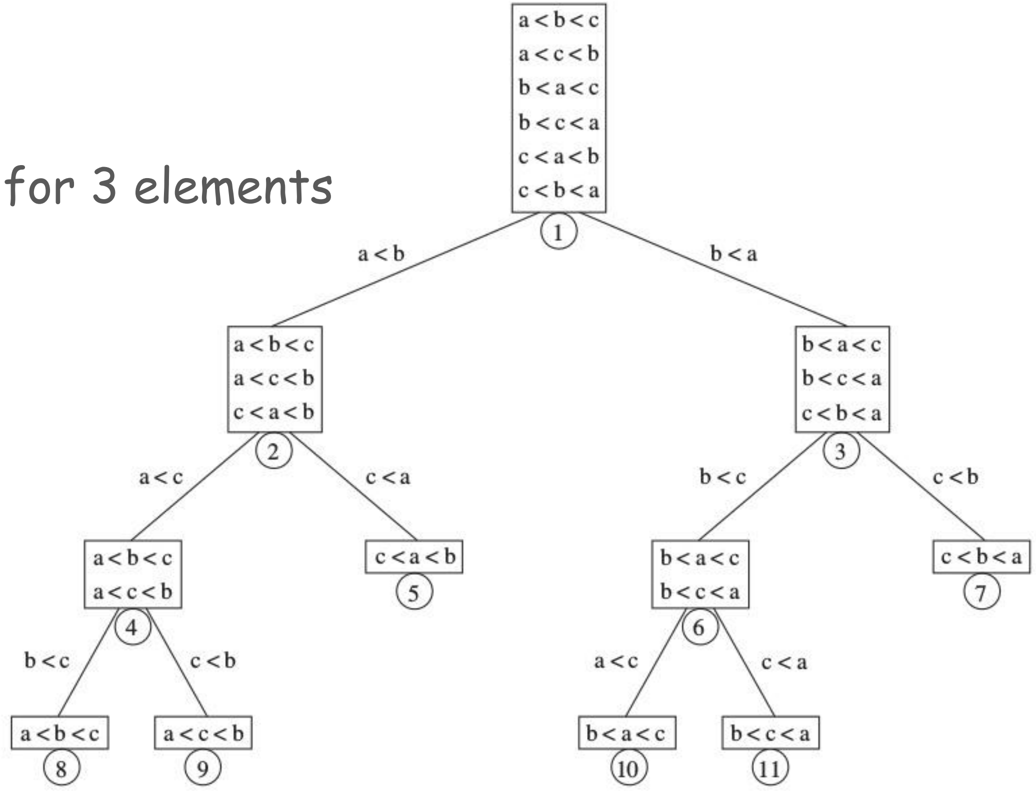
Lower bound for comparison based sorting algorithms

A sorting algorithm is comparison based if it uses comparison operators to find the order between two numbers.

Comparison sorts can be viewed abstractly in terms of decision trees.

Lower bound for comparison based sorting algorithms

Decision tree for 3 elements



Lower bound for comparison based sorting algorithms

1) Each of the n! permutations on n elements must appear as one of the leaves of the decision tree for the sorting algorithm to sort properly.

2) Let x be the maximum number of comparisons in a sorting algorithm. The maximum height of the decision tree would be x. A tree with maximum height x has at most 2^x leaves.

Lower bound for comparison based sorting algorithms

Therefore,

$$N! <= 2^x$$

Taking log on both sides.

$$\log_2(N!) <= x$$

Since $\log_2(N!) = \Theta(N \log N)$

$$x = \Omega(N \log_2 N)$$

Linear Time Sort

Linear Time Sort

Although any general sorting algorithm that uses only comparisons requires O(N log N) time in the worst case.

It is still possible to sort in linear time in **some special cases.**

Counting Sort

Counting Sort

Sorting technique based on keys between a specific range.

It works by counting the number of objects having distinct key values (kind of hashing).

Try this : https://visualgo.net/en/sorting

Counting Sort

Algorithm:

1. Loops over the Data and computing a histogram of the number of times each key occurs within the input collection.
2. performs a prefix sum computation (a second loop, over the range of possible keys) to determine, for each key.
3. loops over the items again, moving each item into its sorted position in the output array.

Counting Sort

Example :

Input : [6, 4, 7, 2, 3, 6, 2]

Index :	0	1	2	3	4	5	6	7

Counting Sort

Example : 1) Loop Input Array and computing histogram of number

Input : [6, 4, 7, 2, 3, 6, 2]

	0	0	0	0	0	0	1	0
Index :	0	1	2	3	4	5	6	7

Counting Sort

Example : 1) Loop Input Array and computing histogram of number

Input : [6, 4, 7, 2, 3, 6, 2]

	0	0	0	0	1	0	1	0
Index :	0	1	2	3	4	5	6	7

Counting Sort

Example : 1) Loop Input Array and computing histogram of number

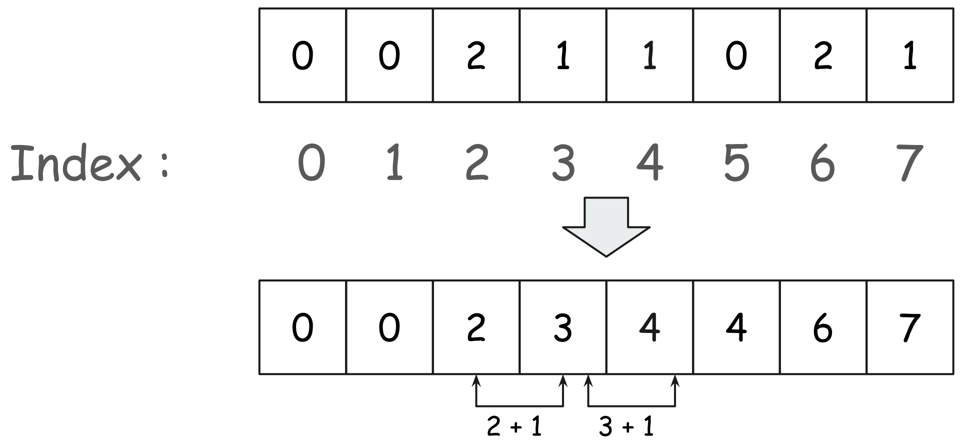
Input : [6, 4, 7, 2, 3, 6, 2]

	0	0	2	1	1	0	2	1
Index :	0	1	2	3	4	5	6	7

Counting Sort

Example : 2) Perform prefix sum computation

Input : [6, 4, 7, 2, 3, 6, 2]



Counting Sort

Example : 3) Moving each item into its sorted position

Input : [6, 4, 7, 2, 3, 6, 2]

	0	0	2	3	4	4	6	7
Index :	0	1	2	3	4	5	6	7

Output :							6	

Counting Sort

Example : 3) Moving each item into its sorted position

Input : [6, 4, 7, 2, 3, 6, 2]

	0	0	2	3	4	4	5	7
Index :	0	1	2	3	4	5	6	7

Output :					4		6	

Counting Sort

Example : 3) Moving each item into its sorted position

Input : [6, 4, 7, 2, 3, 6, 2]

	0	0	2	3	3	4	5	7
Index :	0	1	2	3	4	5	6	7
Output :					4		6	7

Counting Sort

Example : 3) Moving each item into its sorted position

Input : [6, 4, 7, 2, 3, 6, 2]

	0	0	2	3	3	4	5	6
Index :	0	1	2	3	4	5	6	7
Output :			2		4		6	7

Counting Sort

Example : 3) Moving each item into its sorted position

Input : [6, 4, 7, 2, 3, 6, 2]

	0	0	1	3	3	4	5	6
Index :	0	1	2	3	4	5	6	7
Output :			2	3	4		6	7

Counting Sort

Example : 3) Moving each item into its sorted position

Input : [6, 4, 7, 2, 3, 6, 2]

	0	0	1	2	3	4	5	6
Index :	0	1	2	3	4	5	6	7

Output :			2	3	4	6	6	7

Counting Sort

Example : 3) Moving each item into its sorted position

Input : [6, 4, 7, 2, 3, 6, 2]

0	0	1	2	3	4	4	6

Index : 0 1 2 3 4 5 6 7

Output :

	2	2	3	4	6	6	7

Performance of Counting Sort

Run one loop for scan input array :	O(N)
Run one loop for prefix sum computation :	O(k)
Run one loop for move input data to output :	O(N)

Time complexity is O(N) + O(k) + O(N) ⇒ O(N + k)

Auxiliary Space : O(N + k)

Bucket Sort

Bucket Sort

Counting sort may not be too practical if the range of elements is too large.

Bucket sort works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually (eg. Insertion sort).

Bucket Sort

Algorithm :

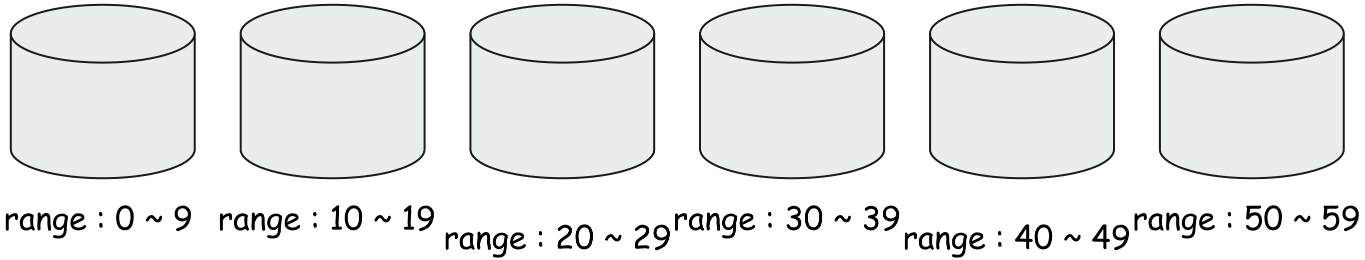
1. Create N empty buckets.
2. Insert data into appropiate bucket
3. Sort individual buckets using insertion sort.
4. Concatenate all sorted buckets.

Bucket Sort

EX) Create N empty buckets.

Input : [12, 45, 33, 54, 10, 41, 42]

bucket :

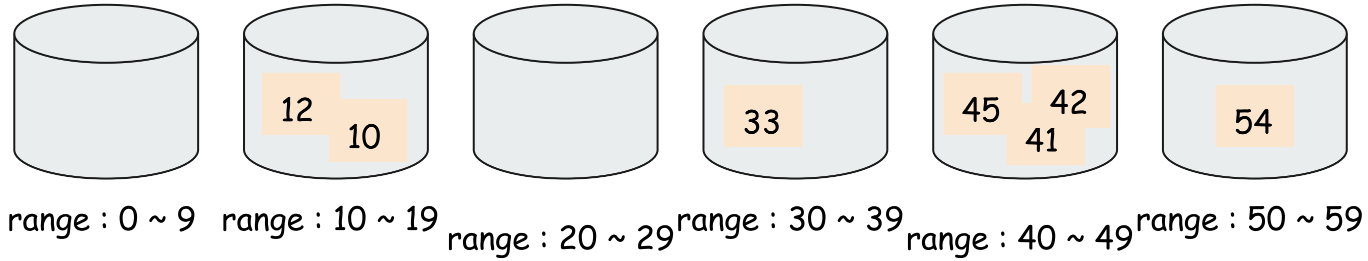


Bucket Sort

EX) Insert data into appropiate bucket.

Input : [12, 45, 33, 54, 10, 41, 42]

bucket :

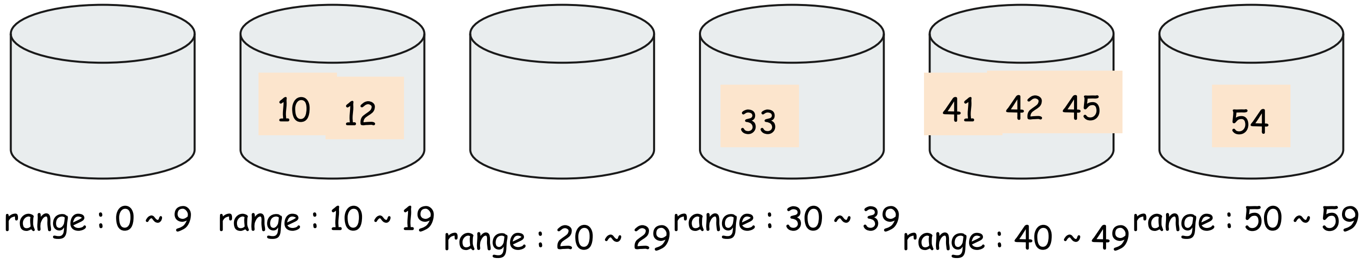


Bucket Sort

EX) Sort individual buckets using insertion sort.

Input : [12, 45, 33, 54, 10, 41, 42]

bucket :

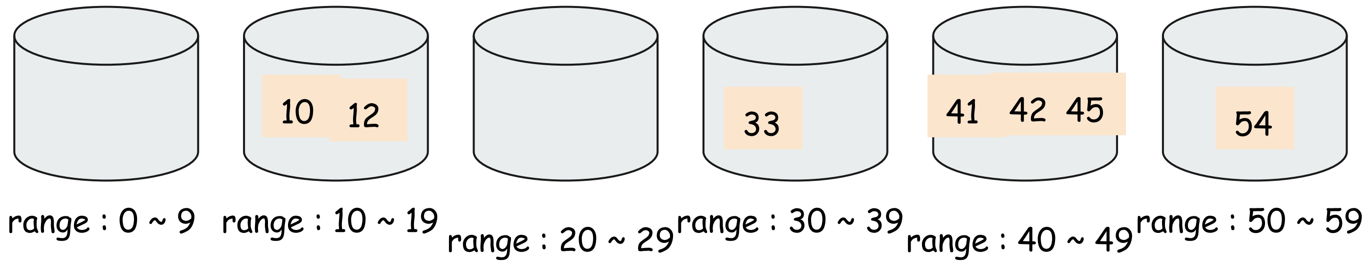


Bucket Sort

EX) Concatenate all sorted buckets.

Input : [12, 45, 33, 54, 10, 41, 42]

bucket :



Output : [10, 12, 33, 41, 42, 45, 54]

Performance of Bucket Sort

Time for put data into buckets :	O(N)
Time for insertion sort in each buckets :	$O(k) \sim O(N^2)$
Time for concatnated buckets :	O(N)

Total Time complexity :

Best Case and Average Case O(N + k)

Worst Case $O(N^2)$

Radix Sort

Radix Sort

The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.

Radix sort uses counting sort as a subroutine to sort.

Try this : https://visualgo.net/en/sorting

Least significant digit radix sort

Algorithm :

1. Take the least significant digit (or group of bits) of each key.
2. Group the keys based on that digit, but otherwise keep the original order of keys.
3. Repeat the grouping process with each more significant digit.

Least significant digit radix sort

EX) : Take the least significant digit (or group of bits) of each key.

Input : [12, 140, 33, 254, 101, 341, 242]

Group the keys based on that digit.

Output1 : [140, 101, 341, 12, 242, 33, 254]

Least significant digit radix sort

EX) : Repeat the grouping process with each more significant digit.

Output1 : [140, 101, 341, 12, 242, 33, 254]

Group the keys based on that digit.

Output2 : [101, 12, 33, 140, 341, 242, 254]

Least significant digit radix sort

EX) : Repeat the grouping process with each more significant digit.

Output2 : [101, 012, 033, 140, 341, 242, 254]

Group the keys based on that digit.

Output3 : [12, 33, 101, 140, 242, 254, 341]

Performance of Radix Sort

If there are n integers to sort in radix sort and the integers are represented in base *b* and the maximum length of the digit is *d*, then radix sort will execute O(N + *b*) counting sort for each of digits,

resulting in a O(d(N + b)) runtime for radix sort.

Radix Sort Challenge : Nearly O(N)? Radix Sort

How about the performance ? if we use base 2^{16} in radix sort ?

The number have only 2 digits

Thus the performance becomes $O(2*(N+2^{16}))$

faster than quicksort ?? but the auxiliary memory is O(N)

Time Complexities of all Linear Sorting algorithm

	Best	Average	Worst
Counting	$\Omega(N + k)$	$\Theta(N + k)$	$O(N + k)$
Bucket	$\Omega(N + k)$	$\Theta(N + k)$	$O(N^2)$
Radix	$\Omega(Nk)$	$\Theta(Nk)$	$O(Nk)$

k is number of digits in radix sort

External Sorting

External Sorting

All the algorithms we have examined require that the input fit into main memory.

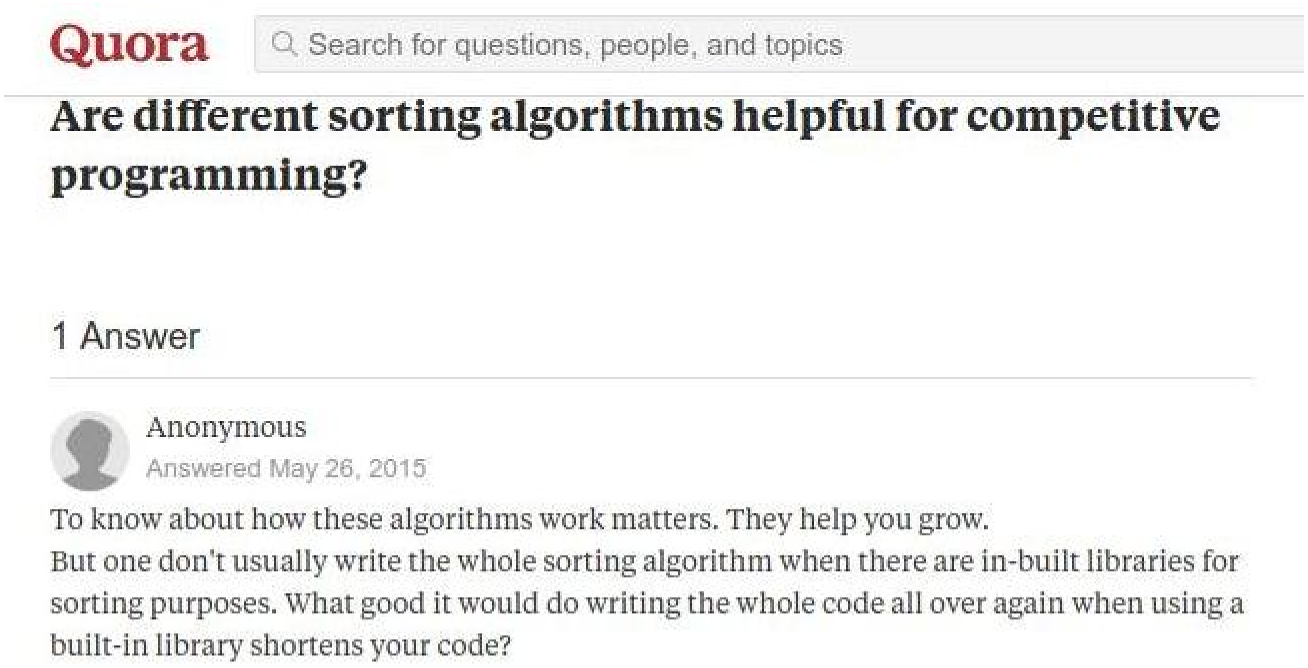
However, there are some applications where the input is much too large to fit into memory.

External Merge Sort

Sorts chunks that each fit in RAM, then merges the sorted chunks together.

1. Divide the file into runs such that the size of a run is small enough to fit into main memory.
2. Sort each run in main memory using merge sort sorting algorithm.
3. Merge the resulting runs together into successively bigger runs, until the file is sorted.

Misc.



Quora

Search for questions, people, and topics

Are different sorting algorithms helpful for competitive programming?

1 Answer

Anonymous

Answered May 26, 2015

To know about how these algorithms work matters. They help you grow.
But one don't usually write the whole sorting algorithm when there are in-built libraries for sorting purposes. What good it would do writing the whole code all over again when using a built-in library shortens your code?

https://www.quora.com/Are-different-sorting-algorithms-helpful-for-competitive-programming

When is linear time sorting algorithm used?

Counting sort: When you are sorting integers with a limited range.

Radix sort: When log(N) is significantly larger than the number of radix digits.

Bucket sort: When you can guarantee that your input is approximately uniformly distributed.

qsort in C VS sort in C++

- qsort() in c use quicksort
- sort() in c++ use hybrid sort called the introspective sort.
    - It starts by using quick sort, but keeps track of whether the sort is resolving in a reasonable number of operations.
    - If the sort is performing poorly, it switches to heap sort instead.
    - Can get quick sort's speed most of the time, and reasonable speed all of the time.

Some sort problems

Check overlap interval among a given set of intervals

Prob. An interval is represented as a combination of start time and end time. Given a set of intervals, check if any two intervals overlap.

```
Input:  arr[] = {{1,3}, {5,7}, {2,4}, {6,8}}
Output: The intervals {1,3} and {2,4} overlap
```

Expected time complexity is O(nLogn) where n is number of intervals.

Some sort problems

Find the point where maximum intervals overlap

Prob. Consider a big party where a log register for guest's entry and exit times is maintained. Find the time at which there are maximum guests in the party. Note that entries in register are not in any order.

```
Input: arrl[] = {1, 2, 9, 5, 5}
       exit[] = {4, 5, 12, 9, 12}
First guest in array arrives at 1 and leaves at 4,
second guest arrives at 2 and leaves at 5, and so on.

Output: There are maximum 3 guests at time 5.
```

Some sort problems

Prob. Given an array of n integers where each value represents number of chocolates in a packet. Each packet can have variable number of chocolates. There are m students, the task is to distribute chocolate packets such that :

1. Each student gets one packet.
2. The difference between the number of chocolates in packet with maximum chocolates and packet with minimum chocolates given to the students is minimum.

Some sort problems

Input : arr[] = {7, 3, 2, 4, 9, 12, 56}
        m = 3
Output: Minimum Difference is 2
We have seven packets of chocolates and
we need to pick three packets for 3 students
If we pick 2, 3 and 4, we get the minimum
difference between maximum and minimum packet
sizes.

Input : arr[] = {3, 4, 1, 9, 56, 7, 9, 12}
        m = 5
Output: Minimum Difference is 6
The set goes like 3,4,7,9,9 and the output
is 9-3 = 6

Exercise

Exercise

1. Determine the running time of Shellsort for sorted input.
2. What is the running time of heapsort for presorted input.
3. Which of the sorting algorithms in this class are stable and which are not ? Why?
4. Suppose arrays A and B are both sorted and both contain N elements. Give an O(log N) algorithm to find the median of $A \cup B$.