gradeup

Data Structure Part-4

Prep Smart. Stay Safe. go Gradeup

gradeup.co

Data Structure Part- 4

Content :

ANALYSIS OF SIMPLE SORTING ALGORITHMS

One of the basic problem of computer science is ordering a list of items. There's a surplus of solutions to this problem, known as **sorting algorithms.** Few sorting algorithms are simple, such as the bubble sort. Such as the quick sort, are extremely complicated, but produce lightning-fast results.

The simple sorting algorithms can be divided into two classes by the difficulty of their algorithms. Algorithmic complexity is generally written in a form known as Big-O notation, where 'O' represents the complexity of the algorithm and a value n represents the size of the set the algorithm is run against.

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

For example, O(n) means that an algorithm has a linear complexity. In other words, it takes ten times longer to operate on a set of 100 items than it does on a set of 10 times (10 ∗ 10 = 100). If the complexity was O(n^2) (quadratic complexity), then it would take 100 times longer to operate on a set of 100 items than it does on a set of 10 items.

The two classes of sorting algorithms are O(n^2), which includes the bubble, insertion, selection, and shell sorts; and O(n log n) which includes the heap, merge, and quick sorts.

In addition to algorithmic complexity, the speed of the various sorts can be compared with empirical data. Since the speed of a sort can vary greatly depending on what data set it sorts, accurate empirical results require several runs of the sort be made and the results averaged together.

Here, we are going to look at three simple sorting techniques : Bubble Sort, Selection Sort, and Insertion Sort.

BUBBLE SORT

It is also known as exchange sort, is a simple sorting algorithm. It works by frequently Because it only uses contrast to operate on elements, it is a comparison sort. stepping through the list to be typed, comparing two items at a time and swapping them if they are in the wrong order. The pass through list is repeated until no swaps are needed, which means the list is sorted. The algorithm gets its name from the way lower elements "bubble" to the top (i.e. the beginning) to the list via the swaps. Because it only uses contrast to operate on elements, it is a **comparison sort**. This is the easiest comparison sort to implement.

Compare every element (except the last one) with its neighbour to the right

- If they are out of order, exchange them

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

- This puts the highest element at the very end
- The last element is now in the correct and final place

Compare every element (except the last two) with its neighbor to the right

- If they are out of sequence, exchange them
- This set the second highest element next to last.
- The last two elements are now in their correct and final places.

Compare every element (except the last three) with its neighbor to the right. Continue as above until you have no unsized elements on the left.

It Each time the outer loop is executed, the inner loop is executed.

is generally considered to the most inefficient sorting algorithm in common usage. Below best-case conditions (the list is already sorted), the bubble sort can approach a constant O(n) level of complexity. General-case is an extremely bad $O(n^2)$.

Bubble Sort (A)

```
1. For i ← 1 to length [A]
2.  For j ← length [A] downto i + 1
3.  If A[j] < A[j - 1]
4.  Exchange (A[j], A[j - 1]
5.
```

The outer loop is performed n - 1 times. All time the outer loop is carry out, the inner loop is performed. Inner loop executes n - 1 times at first, linearly leaving to just once. On average, inner loop performs above n/2 times for each execution of the outer loop. In the inner loop, the balancing is always done (constant time), the swap might be done (also constant time). Thus result is $n*n/2*k$, that is $O(n^2)$.

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

Thus the total number of iteration is

$$B(n) = \sum i = 1^n (n - i) = \frac{1}{2} n(n - 1)$$

Question: Illustrate the operation of Bubble sort on the array A = (5, 2, 1, 4, 3, 7, 6).

SELECTION SORT

we repeatedly search the next highest(or lowest) element in the array and move it to its final position in the condition array. Suppose We begin by selecting **the highest element and moving it to the highest index position.** **that we wish to kind** **we repeatedly find the next largest (or smallest) element in the array and move it to its final** the array in increasing order, i.e. the lowest element at the beginning of the array and the highest element at the end. We begin by selecting the highest element and moving it to the highest index position. We can do this by exchanging the element at the highest index and the largest position. We then reduce the effective size of the array by the one element and repeat the process on the smaller sub array. The process stops when the successful size of the array becomes 1 (an array of 1 element is already sorted). Thus, the selection sort works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled. The selection sort has a complexity of $O(n^2)$.

Selection Sort (A)

```
1. n ← length [A]
2. for j ← 1 to n - 1
3. Smallest ← j
4. For i ← j + 1 to n
5. If A[i] < A [Smallest}
6. Then smallest ← i
```

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

7. Exchange (A[j], A [smallest])

Selection sort is very easy to analyse since none of the loops depends on the data in the array. Choosing the decreasing elements requires scanning all n elements (this takes n – 1 comparisons) and then swapping it into the first position. Searching the next lowest element requires scanning the remaining n – 1 elements and so on, for a total of $(n + 1) + (n - 2) + \ldots + 2 + 1 + O(n^2)$.

Question: Sort the following array using selection sort: A [ ] = (5, 2, 1, 4, 3)

It yields a 60% performance improvement over the bubble sort, but the insertion sort is over twice as fast as the bubble sort and is just as easy to implement as the selection sort. In short, there really isn't any reason to use the selection sort – use the insertion sort instead.

INSERTION SORT

It inserts every item into its proper area in the final list. The easy iapplication of this requires two list form- the source list and the list into which sorted items are inserted. Most application use an in-place sort that works by moving the current item pass the already sorted items and repeatedly swapping it with the preceding item until it is in place.

Like the bubble sort, the insertion sort has a complexity of $O(n^2)$. It has the same difficulty, the insertion sort is a little over twice as efficient as the bubble sort. It has various advantages.

- It is simple to implement
- It is efficient on small data sets
- It is efficient on data sets which are already substantially sorted: it runs in $O(n + d)$ times where d is the number of inversions.

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

- It is more efficient in operation than most other simple O(n^2) algorithms such as selection sort or bubble sort : the average time is $n^2/4$ and it is linear in the best base.
- It is secure (does not change the relative order of elements with equal keys)
- It is setup (only requires a constant amount O(1) of extra memory space)
- It is an online algorithm, in that it can sort a list as it receives it.

Each succession of an insertion sort removes an element from the input data, inserting it at the correct position in the already sorted list, until no elements are left in the input. The alternative of element to remove from the input is random and can be made using almost any choice algorithm. Sorting is typically done in-place. The resulting array after k succession contains the first k entries of the input array and is sorted. In each step, the first remaining entry of the input is removed, inserted into the result at the right position, thus extending the result.

Insertion - Sort (A)

```
1. For j ← 2 to length [A]
2.    do key ← A [j]
3.      ►Insert A[j] into the sorted sequence A [1 … j – 1]
4.         i ← j - 1
5.     While i > 0 and A [j] > key
6.         Do A[i - 1] ← A [i]
7.         i ← i - 1
8.         A[i + 1] ← key
```

Question: Illustrate the operation of INSERTION SORT on the array A = (2, 13, 5, 18, 14).

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

SHELL SORT

- It is a sorting algorithm which requires O(n^2) comparisons and exchanges in the worst case.
- It is a generalization of insertion sort, with two important observations:
  a. Insertion sort is efficient if the input is “almost sorted”
  b. Insertion sort is inefficient, on average, because it moves values just one position at a time
- It upgrade insertion sort by comparing elements separated by a gap of several positions.
- This leads an element take “longer steps” toward its await position. Many passes over the data are taken with lower gap sizes.
- The last step of this is a plain insertion sort, but by then, the arrangement of data is guaranteed to be sorted.

COMPARISON OF VARIOUS SORTING ALGORITHMS

It is a fundamental task that is performed by most computers. It is used commonly in a large variety of important applications. All spread sheet programs contain few sort of sorting code. Database applications used by schools, banks, and other instructions all contain sorting code. Because of the value of sorting in these applications, dozens of sorting algorithms have been developed over the decades with varying complexity. Slow sorting process such as bubble sort, insertion sort, and selection sort have a theoretical complexity of O(N^2). These algorithms are slow for sorting big arrays, the algorithm is easy, so they are not useless. If an application only needs to sort small arrays, the nit is satisfactory to use one of the simple slow sorting algorithms as opposed to a faster, but more complicated sorting algorithm. For these applications, the increase in coding time and chance of coding mistake in using the faster sorting algorithm is not worth the speedup in execution time. Of course, if an

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

application needs a faster sorting algorithm, there are certainly many ones available, including quicksort, merge sort, and heap sort. These algorithms have a pure problem of O(N log N). They are much faster then the O(N^2) algorithms and can sort large arrays in a reasonable amount of time, the cost of these increasing sorting procedure is that the algorithm is much more compound and is harder to correctly code. But the result of the more compound algorithm is an structured sorting method capable of being used to sort very large arrays.

Besides to different complexity, sorting algorithms also fall into two basic categories – balancing based and non-balancing based. A balancing based algorithm sequence a sorting array by weighing the value of one element against the value of other elements. Algorithms such as quicksort, merge sort, bubble sort, and insertion sort are balancing based. Instead, a non-comparison based algorithm category an array without consideration of pairwise data elements. Radix type is a non-balancing based algorithm that treats the sorting elements as numbers represented in a base-M number system, and then works with individual digits of M.

DESCIPTION OF SORTING ALGORITHMS

INSERTION SORT

Insertion sort is good only for sorting small arrays (usually less than 100 times). Actually, the smaller the array, the faster insertion sort is balance to any other sorting algorithm. However, being an O(n^2) algorithm, it becomes slow and quick when the size of the array increases. It was used in the tests with arrangement of size 100.

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

SHELL SORT

It is a rather curious algorithm, quite different from other fast sorting algorithms. It's actually so different that it even isn't an O(n log n) algorithm like the others, but instead it's something between $O(n \log^2 n)$ and $O(n^{1.5})$ depending on implementation details. It's an in-place non-recursive algorithm and it compares very well to the other algorithms, shell sort is a very good alternative to consider.

HEAP SORT

It is the other (and by far the most popular) in-place non-recursive sorting algorithm used in this test. It is not quick possible in all (nor in most) cases, but it's the de-facto sorting algorithm when one wants to make sure that the sorting will not take longer than O(n log n). It is generally value because it is trust worthy: There aren't any "pathological" cases which would cause it to be unacceptably slow. Other than, sorting in-place and in a non-recursive way also makes sure that it will not take extra memory, which is often a nice feature.

gradeup

MERGE SORT

The good quality of merge sort is that it's truly O(n log n) algorithm (like heap sort) and that it's stable (i.e., it doesn't change the order of equal items like e.g., heap sort often does). Its problem is that it contains a second array with the same size as the array to be sorted, thus doubling the memory requirements.

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

QUICK SORT

It is the most popular sorting algorithm. Its asset is that it sorts in-place (even though it's recursive algorithm) an that it usually is very fast. Causes for its speed is that its inner loop is short and can be optimized very well.

The main problem with quick sort is that it's not trustworthy : Its worse - case scenario is $O(n^2)$ (in the worst case it's as slow, if not a little slower than insertion sort) and the pathological cases tends to appear too unexpectedly and without warning, even if an optimized version of quicksort is used.

MERGE SORT

INTRODUCTION

Conceptually, it works as follows:

1. **Divide.** Divide the unclassified list into two sub lists of about half the size.
2. **Conquer.** Sort each of the two subs lists recursively until we have list sizes of length 1, in which case the list itself is returned.
3. **Combine.** Combine the two-sorted sub lists back into one sorted list.

We note that the recursion "bottom out". When the sequence to be sorted has length 1, in which case there is no work to be done, and since every sequence of length 1 is already in sorted order. The working of the combine sort algorithm is the merging of two sorted sequences in the "combine" step. To execute the merging, we use an additional method **MERGE (A, p, q, r),** where A is an array and p, q and r are indices numbering elements of the array such that $p \leq q < r$. the methods suppose that the sub arrays A[p..q] and A[q + 1.. r] are in sorted order. It combines them to form a single sorted sub array that replaces the current sub array A[p..r].

MERGE (A, p, q, r)

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

```
1.  $n_1 \leftarrow q - p + 1$
2.  $n_2 \leftarrow r - q$
3.  create arrays $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$
4.  for $i \leftarrow 1$ to $n_1$
5.          do $L[i] \leftarrow A[p + i - 1]$
6.  For $j \leftarrow 1$ to $n_2$
7.          do $R[j] \leftarrow A[q + j]$
8.  $L[n_1 + 1] \leftarrow \infty$
9.  $R[n_2 + 1] \leftarrow \infty$
10. $i \leftarrow 1$
11. $j \leftarrow 1$
12. for $k \leftarrow p$ to $r$
13.         do if $L[i] \leq R[j]$
14.                 then $A[k] \leftarrow L[i]$
15.                      $i \leftarrow i + 1$
16.                 else $A[k] \leftarrow R[j]$
17.                      $j \leftarrow j + 1$
```

It is easy to imagine a MERGE method that takes time θ(n), where n = r - p + 1 is the number of elements being merged.

We can now use the MERGE method as a subroutine in the merge sort algorithm. The procedure MERGE-SORT (A, p, r) sorts the elements in the sub array A[p..r]. If p $\leq$ r, the sub array has to most one element and is therefore already sorted. If not, the divide step simply computes an index q that

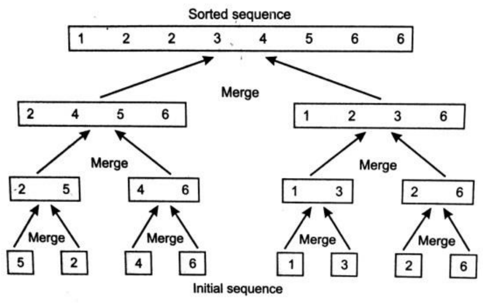
partitions A[p..r] into two sub arrays: A[p..r], containing $\lceil n/2 \rceil$ elements, and A [q + 1..r], containing $\lfloor n/2 \rfloor$ elements.

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

```
MERGE-SORT (A, p, r)

1. if p < r
2.       then q ← $\lfloor (p+r)/2 \rfloor$
3.             MERGE-SORT (A, p, q)
4.             MERGE-SORT (A, q + 1, r)
5.             MERGE (A, p, q, r)
```

To sort the entire order A = (A[1], A[2], ….. A[n]), we call MERGE-SORT (A, 1, length [A]), where once again length [A] = n. If we look at the operation of the procedure bottom-up when n is a power of two, the algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length n / 2 are merged to form the final sorted order of length n. Figure shows this method in the array (5, 2, 4, 6, 1, 3, 2, 6).



ANALYSIS OF MERGE SORT

Even if the pseudo code for MERGE-SORT works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that the original problem size is a power of two. Each divide step then yield two subsequence of size exactly n/2. This assumption does not affect the order to growth of the solution to the recurrence.

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

It is one element takes constant time. When we have n > 1 elements, we break down the running time by following ways:

- **Divide.** The divide step just computes the middle of the sub array, which takes constant time. Thus, D(n) = θ(1).
- **Conquer.** We recursively solve two sub problems, each of size n/2, which contributes 2T (n/2) to the running time.
- **Combine.** We have already noted that the MERGE procedure on an n-element sub array takes time θ(n), so C(n) = θ(n).

When we add the functions D(n) and C(n) for the merge sort analysis, we are adding a function that is θ(n) and a function that is θ(1). This sum is a linear function of n, that is θ(n). Adding it to the 2T (n/2) term from the "conquer" step gives the recurrence for the worst-case running time T(n) of merge sort:

$$T(n) = \begin{cases} \theta(1) & if\ n=1 \\ 2T\left(\frac{n}{2}\right)+\theta(n) & if\ n>1 \end{cases}$$

By Master Theorem, we shall show that T(n) is θ(n 1 g n), where1g n stand for $\log_2$ n. For large enough inputs, merge sort, with its θ(n 1 g n) running time, outperforms insertions sort, whose running time is $\theta(n^2)$, in the worst case.

Merge sort's merge operation is useful in online sorting, where the list to be sorted is received a piece at a time, instead of all at the beginning. In this application, we sort each new piece that is received using any sorting algorithm, and then merge it into our sorted list so far using the merge operation. However, this approach can be expensive in time and space if the received pieces are small compared to the sorted list - a better approach in this case is to store the list in a self-balancing binary search tree and add elements to it as they are received.

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

Although heap sort has the same time bounds as merge sort, it requires only $\theta(1)$ auxiliary space instead of merge sort's $\theta(n)$, and is consequently often faster in practical implementations. Quick sort, however, is considered by many to be the faster general-purpose sort algorithm although there are proofs to show that merge sort is indeed the faster sorting algorithm out of the three. Its average-case complexity is O(n log n) even with perfect input and given optimal pivots quick sort is not as fast as merge sort and it is quadratic in the worst case. On the plus side, merge sort is a stable sort, parallelizes better, and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for sorting a linked list: in this situation it is relatively easy to implement a merge sort in such a way that it requires only $\theta(1)$ extra space, and the slow random-access performance of a linked list makes some other algorithms (such as Quick sort) perform poorly, and others (such as heap sort) completely impossible.

As of Perl 5.8, merge sort is its default-sorting algorithm (it was Quick sort in previous versions of Perl). In Java, the Array sort ( ) methods use merge sort or a turned Quick sort depending on the data types and for implementation efficiency switch to insertion sort when fewer than seven array elements are being sorted.

Merge sort incorporates two main ideas to get better its runtime:

1. A small list will take fewer step to sort than a large list.
2. Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. For example, you only have to traverse each list once if they're already sorted (see the merge function below for an example implementation).

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

QUICK SORT

INTRODUCTION

The fundamental of quick sort algorithm was invented by C. A. R. Hoare in 1960 and formally establish quick sort in 1962. It is used on the principle of divide - and - conquer. It is an algorithm of choice in many situations because it is not difficult to implement. It is a good "general - purpose" sort and it consumes relatively fewer resources during execution.

It works by portioning a given array A[p..r] into two non-empty sub arrays A[p..q] and A[q+ ….r] such that every key in A[p…q] is less than or equal to key in A[q+..r]. The exact position of the partition depends on the given array and index q is computed as a part of the partitioning procedure.

QUICK SORT (A, p, r)

```
1.  If p < r then
2. q → PARTITION (A, p, r)
3. QUICK SORT (A, p, q - 1)
4. QUICK SORT (A, q + 1, r)
```

Note that to sort entire array, the initial call is Quick Sort (A, 1, length [A])

As a first step, Quick Sort chooses as pivot one of the items in the array to be sorted. Then array is partitioned on either side of the pivot. Elements that are less than or equal or pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

Example: Use Quick Sort algorithm to sort

36, 15, 40, 1, 60, 20, 55, 25, 50, 20.

Is it a stable sorting algorithm?

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

Solution: Stable Sorting. We say that a sorting algorithm is stable if, when two records have the same key, they stay in their original order.

Let $A[\,] =$

1	2	3	4	5	6	7	8	9	10
36	15	40	1	60	20	55	25	50	20
									x

Here $p = 1$ $r = 10.$

$x = A[10]$ *i.e.,* $x = 20$

$i = p - 1$ *i.e.,* $i = 0.$

$j = 1$ to 9

Now, $j = 1$ and $i = 0$

$A[j] = A[1] = 36$ and $36 \nleq 20$

So $j = 2$ and $i = 0$

$A[2] = 15 \leq 20$ (True)

then $i = 0 + 1 = 1$ and $A[1] \leftrightarrow A[2]$

i.e.,

1	2	3	4	5	6	7	8	9	10
15	36	40	1	60	20	55	25	50	20

Now $j = 3$ and $i = 1$

$A[3] = 40 \nleq 20$

$j = 4$ and $i = 1$

$A[4] = 1 \leq 20$ (True)

then $i = 1 + 1 = 2$ and $A[2] \leftrightarrow A[4]$

i.e.,

1	2	3	4	5	6	7	8	9	10
15	1	40	36	60	20	55	25	50	20

Now $j = 5,$ $i = 2$

$A[5] = 60 \nleq 20$

$j = 6$ and $i = 2$

$A[j] = 20 \leq 20$ (True)

then $i = 1 + 1$ *i.e.,* $i = 2 + 1 = 3$

and $A[3] \leftrightarrow A[6]$

i.e.,

1	2	3	4	5	6	7	8	9	10
15	1	20	36	60	40	55	25	50	20

Now

$j = 7,$ $i = 3$

$A[7] = 55 \nleq 20$

$j = 8$ $i = 3$

$A[8] = 25 \nleq 20$

$j = 9,$ $i = 3$

$A[9] = 50 \nleq 20$

Now $A[i + 1]$ *i.e.,* $A[4] \leftrightarrow A[10]$

i.e.,

1	2	3	4	5	6	7	8	9	10
15	1	20	20	60	40	50	25	50	36

and two sub-arrays are

15	1	20

and

60	40	50	25	50	36

Thus, this is a stable algorithm.

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

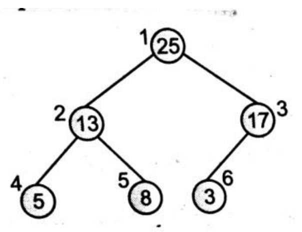
HEAP SORT

HEAP PROPERTY

In a Max-heap, for every node i other then the root, the value of a node is greater than or equal to (at most) to the value of its parent.

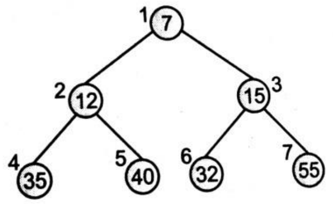
$$A\ [\text{PARENT}\ (i)] \geq A[i]$$

Thus, the largest element in a heap is sorted at the root. Following is an example of MAX-HEAP



And a Min-Heap is organized in the opposite way i.e., the min-heap property is that for every node i other than the root is

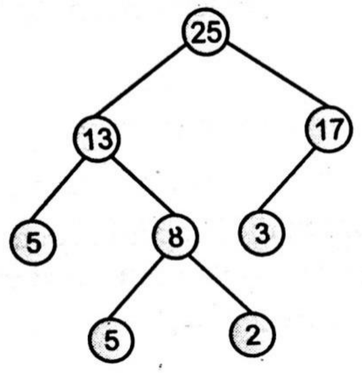
$$A\ [\text{PARENT}\ (i)] \leq A[i]$$



By the definition of heap, all the tree levels are completely filled except possibly for the lowest level which is filled from the left up to a point.

Following is not a heap, because it only has the heap property – it is not a completely binary tree. Recall that to be complete, a binary tree has to fill up all of its levels with the possible exception of the last one, which must be filled in from the left side.

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series



HEIGHT OF A HEAP

HEIGHT OF A NODE

We define the height of a node in a tree to be a number of edges on the longest simple downward path from a node to a leaf.

HEIGHT OF A TREE

The number of edges on a simple downward path from a root to a leaf.

Note that the height of a tree with n node is $\lfloor \lg n \rfloor$ which is θ(lg n). This implies that an n-element heap has height$\lfloor \lg n \rfloor$.

In order to show this let the height of n-element heap be h. From the bounce obtained on highest and lowest number of elements in a heap, we get

$$2^h \leq n \leq 2^{h+1} - 1$$

Where n is the number of elements in a heap.

$$2^h \leq n \leq 2^{h+1}$$

Taking logarithms to the base 2

$$h \leq \lg n \leq h + 1$$

It follows that h = $\lfloor \lg n \rfloor$.

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

Clearly, a stack of height h has the minimum number of elements when it has just one node at the lowest level. The levels above the lowest level form a complete binary free of h - 1 and $2^k - 1$ nodes.

Hence, the minimum number of nodes possible in a heap of height h is 2^k.

Clearly, a heap of height h, has the maximum number of elements when its lowest level is completely filled. In this case the heap is a complete binary free of height h and hence has $2^{h+1} - 1$ nodes.

HEAPIFY

MAINTAINING THE HEAP PROPERTY

It is a procedure for manipulating heap data structures. It is a given an array A and index i into the array. The sub tree rooted at the children of A[i] are heap but node A [i] itself has possibly violate the heap property i.e., A[i] < A[2i] or A[i] < A[2i + 1]. The method 'Heapify' manipulates the tree rooted at A[i] so it becomes a heap. It is let the value at A[i] "float down" in a heap so that sub tree rooted at index i becomes a heap.

OUTLINE OF PROCEDURE HEAPIFY

It picks the largest child key and compares it to the parent key. If parent key is largest than heapify quits, otherwise it swaps the parent key with the largest child key. So that the parent now becomes biggest than its children. It is imposed to note that swap may destroy the heap property of the sub tree rooted at the largest child node. If this is the case, heapify calls itself again using largest child node as the new root.

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

ANALYSIS

If we put a value of root that is less than every value in the left and right subtree, then 'Heapify' will be called recursively until leaf is reached. To make recursive calls traverse the longest path to a leaf, choose value that makes 'Heapify' always recurs on the left child. It follows the left branch when left child is greater than or equal to the right child, so putting 0 at the root and 1 at all other nodes, for example, will accomplish this task. With such values 'Heapify' will be called h times, where h is the heap height so its running time will be θ(h) (since each call does θ(1) work), which is θ(lg n). Since we have a case in which Heapify's running time θ(lg n) its worst-case running time is Ώ (lg n).

HEAP SORT ALGORITHM

The heap sort combines the best of both merge sort and insertion sort. Like merge sort, the worst case time of heap sort is O(n log n) and like insertion sort, heap sort sorts in-place. The heap sort algorithm starts by using procedure MAX-BUILD-HEAP to built a heap on the input array A[1..n]. Since the maximum element of the array stored at the root A[1], it can be put into its correct final position by exchanging it with A[n] (the last element in A). if we now discard node n from the heap then the remaining elements can be made into heap. Note that the new element at the root may violate the heap property. All that is needed to restore the heap property.

```
HEAP-SORT(A)

1. BUILD-MAX-HEAP (A)
2. for i ← length [A] down to 2
3.  do exchange A[1] ↔ A[i]
4.   Heap-size [A] ← heap-size [A] - 1
```

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

```
5.    MAX-HEAPIFY (A, 1)
```

ANALYSIS

We have seen that the running time of 'Build-heap' is O(n). The heap-sort algorithm makes a call to 'Build-heap' for creating a (max) heap, which will take O(n) time and each of the (n-1) calls to Max-heapify to fix up the new heap (which is created after exchanging the root and by decreasing the heap size). We know 'MAX-HEAPIFY' takes time O(lg n).

Thus the total running time for the heap-sort is O(n lg n).

HEAP-INSERT

The procedure MAX-HEAP-INSERT implements the INSERT operation. It takes the key of new element as the input into max-heap A. the method first expands the max-heap by adding to the tree a new leaf whose key is -∞. Then it calls HEAP-INCREASE-KEY to set the key of this new nodes to its correct value and maintain the max-heap property.

```
MAX-HEAP-INSERT (A, KEY)

1. Heap-size [A] ← heap-size [A] + 1
2. A [heap-size [A] ] ← -∞
3. HEAP-INCREASE-KEY (A, heap-size [A], key)
```

```
HEAP-INCREASE-KEY (A, i,. key)

1.  if key < A[i]
2.    then error "new key is smaller than current key"
3.  A [i] ← key
4.  While i > 1 and A {PARENT (i) < A[i] ]
5.    Do exchange A[i] ↔ A [PARENT (i)]
```

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

```
6.   i ← PARENT (i)
```

the running time of MAX-HEAP-INSERT on an n-element heap is O(lg n).

HEAP-DELETE

HEAP-DELETE (A, i) is the procedure, which deletes the item in node i from heap A. HEAP-DELETE runs in O(lg n) time for an n-element max-heap.

```
HEAP-DELETE(A, i)

1. A[i] ← A[heap-size [A] ]
2. Heap-size [A] ← heap-size [A] - 1
3. MAX-HEAPIFY (A, i)
```

SORTING IN LINEAR TIME

Merge sort, quick sort and heap sort algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements. We call such sorting algorithms **comparison sorts.** Any comparison sort must make Ω (n lg n) comparisons in the worst case to sort a sequence of n elements.

Heap Sort	n log n
Insertion Sort	n^2
Quick Sort	n^2
Merge Sort	n log n
Randomized Quick Sort	n log n expected

There are sorting algorithms that run quicker than O(n lg n) time but they require special belief about the input sequence to be sort. Examples of sorting algorithms that run in limited time are counting sort, radix sort and bucket sort. Counting sort and radix sort suppose that the input consists of integers in a small range. Whereas, bucket sort assumes that a random process that distributes elements uniformly over the interval generates the input. Needless

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

to say, these algorithms use operations other than comparisons to determine the sort order.

STABILITY

We say that a sorting algorithm is secure, when two records have the same key, they stay in their original order. This property will be important extending bucket sort to an algorithm that works well when k is large.

- Bucket sort? Yes. We add items to the lists A[i] in order, and concatenating them preserves that order.
- Heap sort? No. The act of placing objects into a heap (and heap filing them) destroys any initial ordering they might have.
- Merge sort? Maybe. It depends on how we divide lists into two, and on how we merge them. For instance if we divide by choosing every other element to go into each list, it I unlikely to be stable. If we divide by splitting a list at its midpoint, and break ties when merging in favour of the first list, then the algorithm can be stable.
- Quick sort? Again, may be. It depends on how we do the partition step.

COUNTING SORT

It suppose that every of the n input elements is an integer in the range 1 to k, for some integer k. when k = O(n), the sort runs in O(n) time. The basic idea of counting sort is to determine, for each input element x, the number of elements less then x. this information can be used to place element x directly into its position 16. This project must be updated 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. In the code for counting sort, we suppose that the input is an array A[1..n], and thus length[A] = n. We require two other

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

arrays: the array B[1..n] holds the sorted output, and the array C[1..k] provides temporary working storage, where k is the highest number in array A.

COUNTING-SORT (A, B, k)

```
1. for i ← 0 to k
2.         do C[i] ← 0
3. for j → 1 to length [A]
4.         do C[A[j] ] ← C[A[j] ] + 1
5. /∗ C[i] now contains the number of elements equal to i. ∗/
6. for i ← 1 to k
7.     do C[i] ← C[i] + C[I - 1]
8. /∗ C[i] now contains the number of elements less than or equal to i. ∗/
9. for j ← length [A] down to 1
10.    do B[C[A[j] ] ] ← A [j]
11.    C[A[j] ] ← C[A[j] ] - 1
```

An important property of counting sort is that it is stable: numbers with the same value appear in the output array in the same order as they do in the input array. That is, number appears first in the input array appear first in the output array. Naturally, the property of strength is important only when satellite data are carried around with the element being sorted.

Example: Illustrate the operation of COUNTING-SORT on the array

A = (6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2).

Solution:

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

Here k = 6 (highest number in A)

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

For i = 0 to 6

c[i] = 0

i.e.,

	0	1	2	3	4	5	6
c	0	0	0	0	0	0	0

For j ← 1 to 11

	0	1	2	3	4	5	6
c	2	2	2	2	1	0	2

For i = 1 to 6

	0	1	2	3	4	5	6
c	2	4	6	8	9	9	11

j	A[j]	C[A[j]]	B[C[A[j]]] ← A[j]	C[A[j]] ← C[A[j]] – 1
11	2	6	B[6] ←2	C[2] ← 5
10	3	8	B[8] ←3	C[3] ← 7
9	1	4	B[4] ←1	C[1] ← 3
8	6	11	B[11] ←6	C[6] ← 10
7	4	9	B[9] ←4	C[4] ← 8
6	3	7	B[7] ←3	C[3] ← 6
5	1	3	B[3] ←1	C[1] ← 2
4	0	2	B[2] ←0	C[0] ← 1
3	2	5	B[5] ←2	C[2] ← 4
2	0	1	B[1] ←0	C[0] ← 0
1	6	10	B[10] ←6	C[6] ← 9

	1	2	3	4	5	6	7	8	9	10	11
B	0	0	1	1	2	2	3	3	4	6	6

Thus, the final sorted array is B.

RADIX SORT

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

It is a sorting algorithm that is useful when there is a constant ‘d’ such that all the keys are d digit numbers. To execute Radix-Sort, for p = 1 toward ‘d’ sort the number with respect to the pth digit from the right using any linear-time stable sort.

In a essentially computer, which is a succeeding random-access machine, radix sort is sometimes used to sort records of information that are keyed by multiple fields. For example, we might desire to sort dates by three keys: year, month, and day. We could run a sorting algorithm with a balancing function that, given two dates, compares years, and if there is a tie, compares months, and if another tie occurs, compares days, Alternatively, we could sort the information three times with a stable sort: first on day, next on month, and finally on year.

The code for radix sort is straightforward. The following methods supports that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

RADIX-SORT (A, d)

```
1. for i ← I to d
2.        do use a stable sort to sort array A on digit i.
```

Since a linear-time sorting algorithm is used ‘d’ times and d is a constant, the running time of Radix-Sort is linear. When each digit is in the range 1 to k, and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes time θ (n + k). There are d passes, so the total time for radix sort is $\theta(dn + kd)$. When d is constant and k = O(n), radix sort runs in linear time.

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

Example: The first column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. The vertical arrows indicate the digit position sorted on to produce each list from the previous list.

329		720		720		329
457		355		329		355
657		436		436		436
839	=>	457	=>	839	=>	457
436		657		355		657
720		329		457		720
355		839		657		839
		↑		↑		↑

BUCKET SORT

It runs in linear time on the average. Like counting sort, bucket sort is fast because it assumes something about the input. Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes uniformly over the interval U = [0, 1)

To sort n input numbers, Bucket-Sort

1. Partitions U into n non-overlapping intervals, called buckets.
2. Puts each input number into its bucket.
3. Sorts each bucket using a simple algorithm, e.g. Insertion-Sort, and then,
4. Concatenates the sorted lists.

The idea of this is to divide the interval [0, 1) into n equal-sized subintervals, or **buckets,** and then distribute the n input numbers into the buckets. Since the input are uniformly distributed over [0, 1), we don't expect many numbers to fail into each bucket. To produce the output, we simply sort the numbers

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

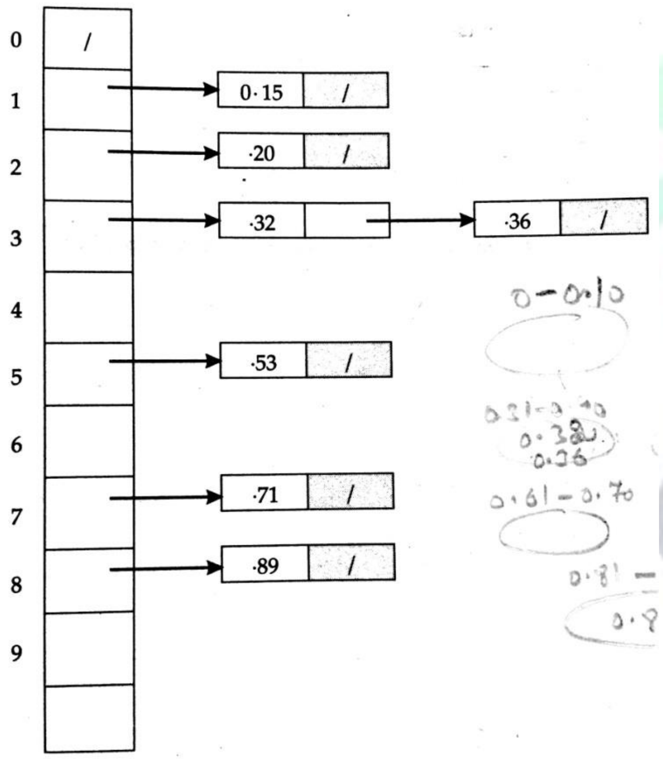
in each bucket and then go through the buckets in order, listing the elements in each.

Example: The Worst-case running time of bucket sort to O(n lg n).

Example: Illustrate the operation of BUCKET-SORT on the array

A = (0.36, 0.15, 0.20, 0.89, 0.53, 0.71, 0.32)

Solution:



Now, Sorted array is

0	1	2	3	4	5	6
0.15	0.20	0.32	0.36	0.53	0.71	0.89

Linear Search :

A simple approach is to do a linear search, i.e

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

- Start from the leftmost element of ar[] and one by one compare x with each element of ar[]
- If x matches with an element, return the index of that element.
- If x doesn't match with any of elements, then return -1.

Time complexity of the algorithm is O(n).

It is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly the faster-searching comparison to Linear search.

Binary Search: It search in a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering whole array. If value of the search key is less than item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to upper half. Repeatedly check until the value is found or the interval is empty.

Idea of binary search is to use information that the array is sorted and reduce the time complexity to **O(Log n).**

Gradeup UGC NET Super Subscription
Access to all Structured Courses & Test Series

gradeup SUPER

Gradeup UGC NET
Super Superscription

Features:

1. 7+ Structured Courses for UGC NET Exam
2. 200+ Mock Tests for UGC NET & MHSET Exams
3. Separate Batches in Hindi & English
4. Mock Tests are available in Hindi & English
5. Available on Mobile & Desktop

Gradeup Super Subscription, Enroll Now