SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF INFORMATION TECHNOLOGY

UNIT - II

Design and Analysis of Algorithm – SCSA1403

Mathematical Foundations **9 Hrs.**

Solving Recurrence Equations - Substitution Method - Recursion Tree Method - Master Method - Best Case - Worst Case - Average Case Analysis - Sorting in Linear Time - Lower bounds for Sorting - Counting Sort - Radix Sort - Bucket Sort.

Recurrence Equations

The recurrence equation is an equation that defines a sequence recursively .It is normally in the form

$T(n) = T(n-1) + n$ for $n>0$ (Recurrence relation)

$T(0) = 0$ (Initial condition)

The general solution to the recursive function specifies some formula.

Solving Recurrence Equations

The recurrence relation can be solved by following methods

- Substitution method
- Master's method

1.Substitution Method

There are two types of substitution

- Forward substitution
- Backward substitution

Forward Substitution method

This method makes use of an initial condition in the initial term and value for the next term is generated. This process is continued until some formula is guessed. Thus in this kind of method, we use recurrence equations to generate few terms.

For Example

Consider a recurrence relation $T(n) = T(n-1) + n$ with initial condition $T(0) = 0$

Let $T(n) = T(n-1) + n$

If n = 1 then

$T(1) = T(0) + 1 = 0+1 = 1$ ------- (1)

If n = 2 then

$T(2) = T(1) + 2 = 1+2 = 3$ ------- (2)

If n = 3 then

$T(3) = T(2) + 3 = 3 + 3 = 6$ -------- (3)

By observing above equation , we can says that it is sum of n natural number

$T(n) = \frac{n(n+1)}{n} = n^2/2 + \frac{n}{2}$

So we can written as

$T(n) = O(n^2)$

Backward Substitution Method

In this method backward values are substituted recursively in order to derive some formula.

For Example

Consider , a recurrence relation T(n) = T(n-1) + n with initial condition T(0) = 0 ------ (1)

Solution:

In Eqa(1) , to calculate T(n) , we need to know the value of T(n-1)

T(n-1) = T(n-1-1) + (n-1) = T(n-2)+(n-1)

Now Equ(1) becomes T(n) = T(n-2)+(n-1) + n ---------- (2)

T(n-2) = T(n-2-1) + (n-2) = T(n-3) + (n-2)

Now Eqa(2) becomes T(n) = T(n-3)+(n-2)+(n-1)+n ------ ---(3)

In the k^{th} terms

T(n) = T(n-k)+(n-k+1)+(n-k+2)+-----+n ----------(4)

If k = n in equ(4) then

T(n) = T(0)+1+2+3+ ------ +n

T(n) = 0+1+2+3+-----+n by substituting initial value T(0) = 0

T(n) = $\frac{n(n+1)}{n} = n^2/2 + \frac{n}{2}$

So T(n) in terms of big oh notation as

T(n) = $O(n^2)$

Example : 2

T(n) = T(n-1) + 1 with initial condition with T(0) = 0 . Find big oh notation.

Solution:

T(n) = T(n-1) + 1 --------- (1)

T(n-1) = T(n-2)+1

Now eqa(1) becomes T(n) = (T(n-2)+1)+1 = T(n-2)+2 -------- (2)

T(n-2) = T(n-3) + 1

Now eqa(2) becomes T(n) = (T(n-3)+1)+2 = T(n-3)+3 ------ (3)

So

T(n) = T(n-k)+k ---------- (4)

If k = n then eqa(4) becomes

T(n) = T(0) + n = 0 + n = n

T(n) = O(n)

Example 3:

T(n) = $2T(n/2) + n$. T(1) = 1 as initial condition
Solution:

T(n) = $2T(n/2) + n$. ---------- (1)

$T(n/2) = 2T(n/4) + n/2$

Now Eqa (1) becomes

T(n) = $2[2T(n/4) + n/2]$+n = $4T(n/4)$+n+n = $4T(n/4)$+2n ----- (2)

$T(n/4) = 2T(n/8) + n/4$

Now eqa(2) becomes

T(n) = $4[2T(n/8) + n/4]$+2n = 8T(n/8)+n+2n = 8T(n/8)+3n ------ (3)

Equ(3) can be written as

T(n) = 2^3T($n/2^3$)+3n

In general

T(n) = 2^kT($n/2^k$) + kn ---------- (4)

Assume $2^k = n$

Now Equ(4) can be written as

T(n) = n.T(n/n)+logn.n

=n.T(1) + n.logn

T(n) = n + n.logn

i.e T(n) = O(n.logn)

Example 4:

T(n) = T(n/3) + C and initial condition T(1) = 1

Solution :

T(n) = T(n/3) + C ---------- (1)

T(n/3) = T(n/9)+C

Now Equ(1) becomes

$T(n) = [T(n/9)+C] + C = T(n/9) + 2C$ ----- (2)

$T(n/9) = T(n/27) + C$

Now Equ(2) becomes

$T(n) = [T(n/27)+C] + 2C$

$T(n) = T(n/27) + 3C$

In General

$T(n) = T(n/3^k) + kC$

Put $3^k = n$ then

$T(n) = T(n/n)+\log_3 n.C$

$= T(1) + \log_3 n.C$

$T(n) = C. \log_3 n + 1$

Tree Method

In this method, we buit a recurrence tree in which each node represents the cost of a single sub problemin the form of recursive function invocations.Then we sum up the cost at each levelto determine the overall cost.Thus the recursion tree helps us to make a good guess of time complexity. The pattern is typically a arithmetic or geometric series.

For example consider the recurrence relation

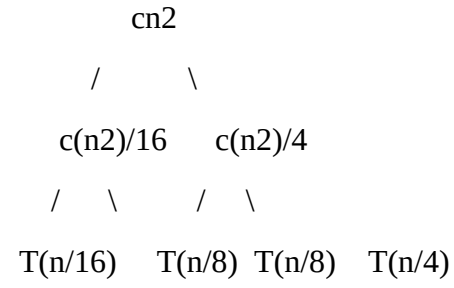
$T(n) = T(n/4) + T(n/2) + cn2$

cn2

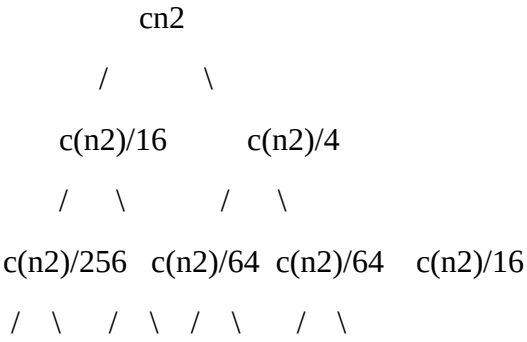
/ \

T(n/4) T(n/2)

If we further break down the expression T(n/4) and T(n/2), we get following recursion tree.



Breaking down further gives us following



To know the value of T(n), we need to calculate sum of tree nodes level by level. If we sum the above tree level by level, we get the following series

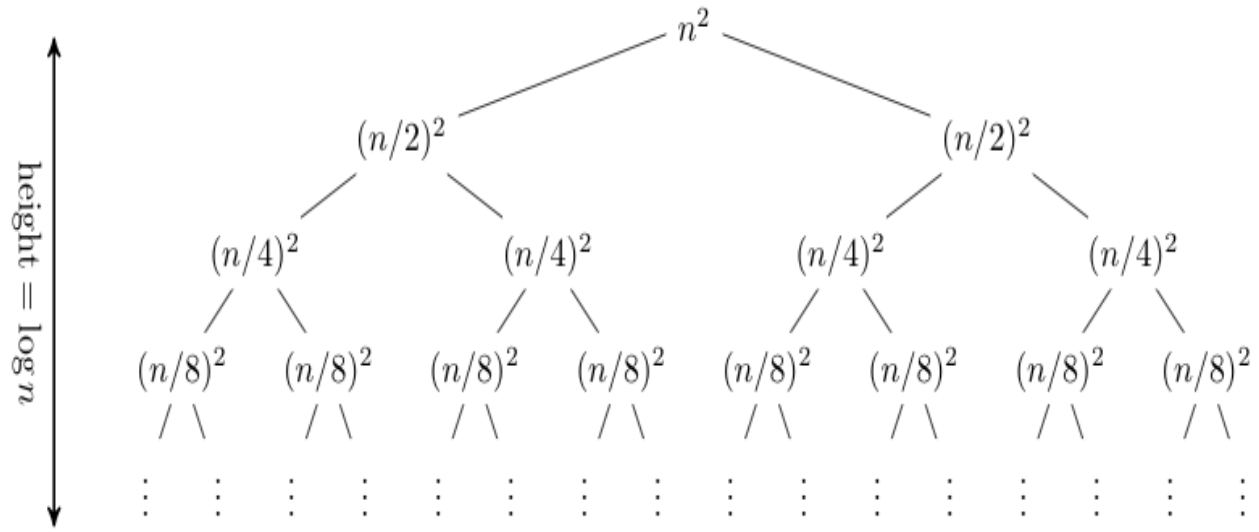
T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ...

The above series is geometrical progression with ratio 5/16. To get an upper bound, we can sum the infinite series. We get the sum as (n2)/(1 - 5/16) which is O(n2)

Example :

$T(n) = 2T(n/2) + n^2$.

The recursion tree for this recurrence has the following form:

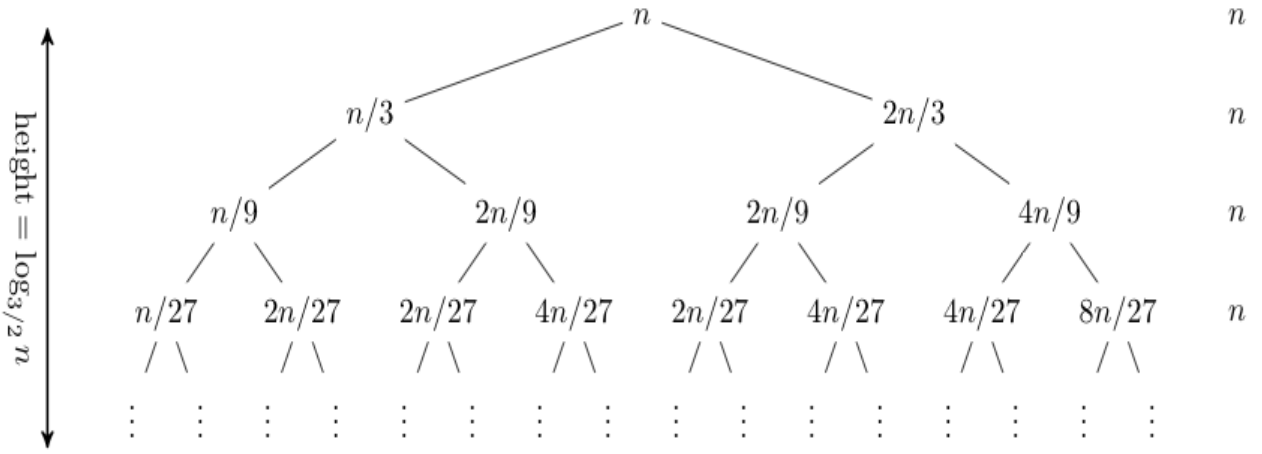


Time complexity of above tree is $O(n^2)$

Let's consider another example,

T(n) = T(n/3) + T(2n/3) + n.

Expanding out the first few levels, the recurrence tree is:



Time complexity of above tree is O(nlogn)

Master's Method:

We can solve the recurrence relation using a formula denoted by master's method.

T(n) = aT(n/b) + F(n) where n ≥ d and d is a constant

Then the master theorem can be stated for efficiency analysis as:

If F(n) is $\Theta(n^d)$ where d ≥ 0

- Case 1 : T(n) = $\Theta(n^d)$ if a< b^d

- Case 2: $T(n) = \Theta(n^d \log n)$ if $a = b^d$
- Case 3 : $T(n) = \Theta(n^{\log_b a})$ if $a > b^d$

EXAMPLE.1 : $T(n) = 4T(n/2) + n$

$A=4, b = 2, F(n) = n = n^1$ i.e $d = 1$

Compare a and b^d , i.e 4 and $2^1 = 4>2$ which satisfied case 3 :

Now $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 4}) = \Theta(n^2)$

Example 2 : $T(n) = T(n/2)+\frac{1}{2}n^2+n$

$A = 1, b = 2, d= 2$

Compare a and b^d , i.e 1 and $2^2 = 1<4$ which satisfied case 1:

$T(n) = \Theta(n^d) = \Theta(n^2)$

Example 3 : $T(n) = 2T(n/4) + \sqrt{n} + 4^2$

$A = 2, b = 4, d = ½$

Compare a and b^d , i.e 2 and $4^{1/2} = 2 = 2$ which satisfied case 2:

$T(n) = \Theta(n^{1/2}\log n) = \Theta(\sqrt{n}\log n)$

Example 4 : $T(n) = 3T(n/2) + \frac{3}{4}n+1$

$A = 3 , b = 2, d = 1$

Compare a and b^d , i.e 3 and $2 = 3 > 2$ which satisfied case 3:

$T(n) = \Theta(n^{\log_b a}) = = \Theta(n^{\log_2 3})$

Another Variation of Master's Method:

$T(n) = aT(n/b) + f(n)$ where $n \geq d$

- Case 1 : if $f(n)$ is $O(n^{\log_b a})$ and $f(n) < n^{\log_b a}$ then

  $T(n) = \Theta(n^{\log_b a})$

- Case 2 : if $f(n) = \Theta(n^{\log_b a}\log n)$ and $f(n) = n^{\log_b a}$ then

  $T(n) = \Theta(n^{\log_b a}\log n)$

- Case 3 : if $f(n) = \Omega(n^{\log_b a})$ and $f(n) > n^{\log_b a}$ then

  $T(n) = \Theta(f(n))$

Steps:

(i) Get the values of a,b and f(n)
(ii) Determine the value $n^{\log_b a}$
(iii) Compare f(n) and $n^{\log_b a}$

Example : 1

T(n) = 2T(n/2)+n

A = 2, b = 2, f(n) = n

Determine $n^{\log_b a} = n^{\log_2 2} = n^1 = n$

Compare $n^{\log_2 2}$ and f(n) i.e n = n which is case 2:

T(n) = $\boldsymbol{\Theta}(n^{\log_b a}\log n) = \boldsymbol{\Theta}(n^1 \log n) = \boldsymbol{\Theta}(n\log n)$

Example : 2:

T(n) = 9T(n/3) + n

A = 9 , b =3,f(n) = n

Determine $n^{\log_b a} = n^{\log_3 9} = n^2$ and

F(n) = n

Now f(n) < $n^{\log_b a}$ which is case 1:

T(n) = $\boldsymbol{\Theta}(n^{\log_b a}) = \boldsymbol{\Theta}(n^{\log_3 9}) = \boldsymbol{\Theta}(n^2)$

Example : 3:

T(n) = 3T(n/4) + nlogn

A = 3, b = 4, f(n) = nlogn

Determine $n^{\log_b a} = n^{\log_4 3}$

f(n)> $n^{\log_4 3}$ which is case 3:

T(n) = $\boldsymbol{\Theta}(f(n)) = \boldsymbol{\Theta}(n\log n)$

Example 4:

T(n) = 3T(n/2) + n^2

A = 3, b = 2, f(n) = n^2

Determine $n^{\log_b a} = n^{\log_2 3}$

$n^2 > n^{\log_2 3}$ case 3:

$T(n) = \Theta(f(n)) = \Theta(n^2)$

Example 5:

$T(n) = 4T(n/2) + n^2$

$A = 4, b = 2, f(n) = n^2$

Determine $n^{\log_b a} = n^{\log_2 4} = n^2$

$F(n) = n^2$ case 2:

$T(n) = \Theta(n^{\log_b a}\log n) = \Theta(n^{\log_2 4}\log n) = \Theta(n^2\log n)$

Example 6:

$T(n) = 4T(n/2) + n/\log n$

$A = 4 , b = 2 , f(n) = n/\log n$

Determine $n^{\log_b a} = n^{\log_2 4} = n^2$

$F(n) < n^2$ case 1 :

$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 4}) = \Theta(n^2)$

Example 7 :

$T(n) = 6T(n/3) + n^2\log n$

$A = 6 , b = 3 , f(n) = n^2\log n$

Determine $n^{\log_b a} = n^{\log_3 6} = n^2$

$F(n) > n^{\log_b a}$ case 3:

$T(n) = \Theta(f(n)) = \Theta(n^2\log n)$

Example 8 : (Need to be solved)

$T(n) = 4T(n/2) + cn$ case 1:

$T(n) = \Theta(n^2)$

Example 9 : (Need to be solved)

$T(n) = 7T(n/3) + n^2$

$T(n) = \Theta(n^2)$ case 3:

Example 10 : (Need to be solved)

$T(n) = 4T(n/2) + \log n$

$T(n) = \Theta(n\log n)$ case 2.

Example 11 : (Need to be solved)

$T(n) = 16T(n/4) + n$

$T(n) = \Theta(n^2)$ case 1

Example 12 : (Need to be solved)

$T(n) = 2T(n/2) + n\log n$

$T(n) = \Theta(\log n)$ case 3.

Worst Case - Average Case Analysis - Linear Search

Let us consider the following implementation of Linear Search.

```
// Linearly search x in arr[].  If x is present then return the index,
// otherwise return -1
int search(int arr[], int n, int x)
{
   int i;
   for (i=0; i<n; i++)
   {
     if (arr[i] == x)
       return i;
   }
   return -1;
}
```

Worst Case Analysis (Usually Done)

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array or the search element is present at n^{th} location. For these cases, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be O(n).

Average Case Analysis (Sometimes done)

Average case complexity gives information about the behaviour of an algorithm on a random input. Let us understand some terminologies that are required for computing average case time complexity.

Let the algorithm is for linear search and P be a probability of getting successful search.N is the total number of elements in the list.

The first match of the element will occur at i^{th} location. Hence the probability of occurring first match is P/n for every i^{th} element.The probability of getting unsuccessful search is (1-P).

Now, we can find average case time complexity Θ (n) as-

Θ (n) =probability of successful search+ probability of unsuccessful search

$$\Theta(n) = \left[1.P/n+2.P/n+...+i.P/n+...n.P/n\right] +n.\ (1-P)$$ //There may be n elements at which chances of not getting element are possible. Hence n. (1-P)

$$=P/n\ [1+2+...+i...n] +n\ (1-P)$$

$$=P/n.\ (n\ (n+1))/2+n\ (1-P)$$

$$\Theta(n) =P\ (n+1)/2+n\ (1-P)$$

Thus we can obtain the general formula for computing average case time complexity.

Suppose if P=0 that means there is no successful search i.e. we have scanned the entire list of n elements and still we do not found the desired element in the list then in such a situation ,

$$\Theta(n) =O\ (n+1)\ /\ 2+n\ (1-0)$$

$$\Theta(n) = n$$

Thus the average case running time complexity is n.Suppose if P=1 i.e. we get a successful search then

$$\Theta(n) = 1(n+1)/2 + n\ (1-1)$$

$$\Theta(n) = (n+1)\ /\ 2$$

That means the algorithm scans about half of the elements from the list. Thus computing average case time complexity is difficult than computing worst case and best case time complexities.

Best Case Analysis (omega)

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Ω (1).

Time complexity for linear search

Best Case	Worst Case	Average Case
Ω(1)	O(n)	Θ (n)

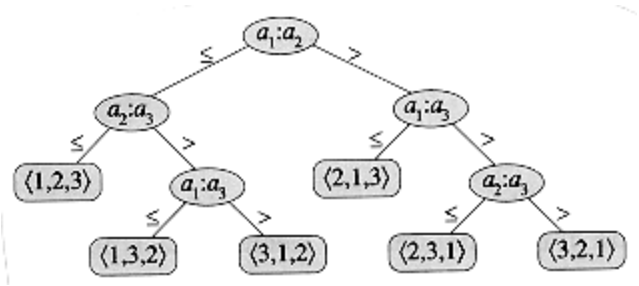
Sorting In Linear Time

Most of the sorting algorithms can sort n numbers in O(n lg n) time. Merge sort and heapsort achieve this upper bound in the worst case; quicksort achieves it on average. Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in Ω(n lg n) time. All those algorithms possess an interesting property say the sorted order they determine is based only on comparisons between the input elements. Therefore such sorting algorithms can be called as comparison sorts.

The following section proves that any comparison sort must make Ω(n lg n) comparisons in the worst case to sort a sequence of n elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor. Further three sorting algorithms which includes--counting sort, radix sort, and bucket sort--that run in linear time. Needless to say, these algorithms use operations other than comparisons to determine the sorted order. Consequently, the Ω(n lg n) lower bound does not apply to them.

Lower bounds for sorting:

In a comparison sort, comparisons between elements made in order to gain the order information about an input sequence ⟨a1, a2, . . . ,an) That is, given two elements ai and aj, One of the tests might be performed ai < aj, ai ≤ aj, ai = aj, ai ≥ aj, or ai > aj to determine their relative order. We may not inspect the values of the elements or gain order information about them in any other way. We assume without loss of generality that all of the input elements are distinct. Given this assumption, comparisons of the form ai = aj are useless, so we can assume that no comparisons of this form are made. We also note that the comparisons ai ≤ aj, ai ≥ aj, ai > aj, and ai < aj are all equivalent in that they yield identical information about the relative order of ai and aj. We therefore assume that all comparisons have the form ai ≤ aj.



The decision tree for insertion sort operating on three elements. There are 3! = 6 possible permutations of the input elements, so the decision tree must have at least 6 leaves.

The decision-tree model

Comparison sorts can be viewed abstractly in terms of decision trees. A decision tree represents the comparisons performed by a sorting algorithm when it operates on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored. The above figure shows the decision tree corresponding to the insertion sort algorithm for an input sequence of three elements.

In a decision tree, each internal node is annotated by ai : aj for some i and j in the range $1 \le i, j \le n$, where n is the number of elements in the input sequence. Each leaf is annotated by a permutation $\langle \Pi(1), \Pi(2), \ldots, \Pi(n) \rangle$. 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 $ai \le aj$ is made. The left subtree then dictates subsequent comparisons for $ai \le aj$, and the right subtree dictates subsequent comparisons for $ai > aj$. When we come to a leaf, the sorting algorithm has established the ordering $a\Pi(1) \le a\Pi(2) \le \ldots \le a\Pi(n)$. 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.

A lower bound for the worst case

The length of the longest path from the root of a decision tree to any of its leaves represents the worst-case number of comparisons the sorting algorithm performs. Consequently, the worst-case number of comparisons for a comparison sort corresponds to the height of its decision tree. A lower bound on the heights of decision trees is therefore a lower bound on the running time of any comparison sort algorithm. The following theorem establishes such a **lower bound.**

Theorem

Any decision tree that sorts n elements has height $\Omega(n \lg n)$.

Proof Consider a decision tree of height h that sorts n elements. Since there are n! permutations of n elements, each permutation representing a distinct sorted order, the tree must have at least n! leaves. Since a binary tree of height h has no more than 2h leaves, we have

$n! \leq 2h$,

which, by taking logarithms, implies

$h \geq \lg(n!)$,

since the lg function is monotonically increasing. From Stirling's approximation, we have

$$n! > \left(\frac{n}{e}\right)^n ,$$

$$\begin{aligned} h &\geq \lg\left(\frac{n}{e}\right)^n \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n) . \end{aligned}$$

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. Radix sort iteratively orders all the strings by their n^{th} character – in the first iteration, the strings are ordered by their last character. In the second run, the strings are ordered in respect to their penultimate character. And because the sort is stable, the strings, which have the same penultimate character, are still sorted in accordance to their last characters. After n^{th} run the strings are sorted in respect to all character positions.

Consider the following 9 numbers:

493 812 715 710 195 437 582 340 385

We should start sorting by comparing and ordering the one's digits:

Digit	Sublist
0	710,340
1	
2	812, 582
3	493
4	
5	715, 195, 385
6	
7	437
8	
9	

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sub lists above. Now, we gather the sub lists (in order from the 0 sub list to the 9 sub list) into the main list again:

710, 340 ,812, 582, 493, 715, 195, 385, 437

Note: The order in which we divide and reassemble the list is extremely important, as this is one of the foundations of this algorithm.

Now, the sub lists are created again, this time based on the ten's digit:

Digit	Sub list
0	
1	710,812, 715
2	
3	437
4	340
5	
6	
7	
8	582,,385
9	493, 195

Now the sub lists are gathered in order from 0 to 9:

710, 812, 715, 437, 340, 582,385, 493,195

Finally, the sub lists are created according to the hundred's digit:

Digit	Sub list
0	
1	195
2	
3	340, 385
4	437, 493
5	582
6	
7	710 ,715
8	812
9	

At last, the list is gathered up again:

195, 340, 385,437,493,582,710,715,812

And now we have a fully sorted array! Radix Sort is very simple, and a computer can do it fast. When it is programmed properly, Radix Sort is in fact one of the fastest sorting algorithms for numbers or strings of letters.

Radix-Sort(A, d)

// Each key in A[1..n] is a d-digit integer.

(Digits are // numbered 1 to d from right to left.)

for i = 1 to d do

Use a stable sorting algorithm to sort A on digit i.

Another version of Radix Sort Algorithm

```
Algorithm RadixSort(a,n)
{
m = Max(a,n)
d = Noofdigit(M)
Make all the element are having “d” number of digit
for(i=1;i<=d,i++)
{
        for(r=0; r<= 9; r++)
        count[r] = 0;
        for(j =1;j<=n;j++)
        {
```

```
        p= Extract(a[j],i);
        b[p][count[p]] = a[j];
        count[p]++;
        }
        s =1;
        for(t=0;t<=9; t++)
        {
        for(k=0;k<count[t];k++)
        {
        a[s] = b[t][k];
        s++;
        }
        }
}
print " Sorted list"
}
```

In the above algorithm assume Max(a,n) is a method used to find out the maximum number in the array, Noofdigit(M) is a method used to find out the number of digit in 'M' and Extract(a[j],i) is a method used to extract the digit from a[j] based on i value (i.e if i value is 1 extract first digit , if i value is 2 extract second digit, if i value is 3 extract third digit from right to left) . Count[] is an array which contains the number of elements available in each row and in each iteration. The number of time i 'for' loop is executed is Depending on the value of 'd', i for loop is repeated.

Disadvantages

Still, there are some tradeoffs for Radix Sort that can make it less preferable than other sorts.

The speed of Radix Sort largely depends on the inner basic operations, and if the operations are not efficient enough, Radix Sort can be slower than some other algorithms such as Quick Sort and Merge Sort. These operations include the insert and delete functions of the sub lists and the process of isolating the digit you want.

In the example above, the numbers were all of equal length, but many times, this is not the case. If the numbers are not of the same length, then a test is needed to check for additional digits that need sorting. This can be one of the slowest parts of Radix Sort, and it is one of the hardest to make efficient.

Analysis

Worst case complexity O(d *n)
Average case complexity **Θ**(d* n).

Best Case Complexity **Ω**(d * n)

Let there be d digits in input integers. Radix Sort takes O(d*(n+b)) time where b is the base for representing numbers, for example, for decimal system, b is 10. What is the value of d? If k is the maximum possible value, then d would be $O(\log_b(k))$. So overall time complexity is $O((n+b) * \log_b(k))$. Which looks more than the time complexity of comparison based sorting algorithms for a large k. Let us first limit k. Let $k <= n^c$ where c is a constant. In that case, the complexity becomes $O(n\text{Log}_b(n))$. But it still doesn't beat comparison based sorting algorithms. What if we make value of b larger?. What should be the value of b to make the time complexity linear? If we set b as n, we get the time complexity as O(n). In other words, we can sort an array of integers with range from 1 to n^c if the numbers are represented in base n (or every digit takes $\log_2(n)$ bits).

Bucket Sort

Bucket sort (bin sort) is a stable sorting algorithm based on partitioning the input array into several parts – so called buckets – and using some other sorting algorithm for the actual sorting of these sub problems.

At first algorithm divides the input array into buckets. Each bucket contains some range of input elements (the elements should be uniformly distributed to ensure optimal division among buckets).In the second phase the bucket sort orders each bucket using some other sorting algorithm, or by recursively calling itself – with bucket count equal to the range of values, bucket sort degenerates to counting sort. Finally the algorithm merges all the ordered buckets. Because every bucket contains different range of element values, bucket sort simply copies the elements of each bucket into the output array (concatenates the buckets).

BUCKET SORT (a,n)

n ← length [a]

m = Max(a,n)

nob = 10 // Number of backet

divider = ceil((m+1)/nob);

for i = 1 to n do

{

j = floor(a[i]/divider)

b[j] = a[i]

}

for i = 0 to 9 do

sort b[i] with Insertion sort

concatenate the lists B[0], B[1], . . B[9] together in order.

End Bucket Sort

Example :

a = { 123,67,45,3,69,245,35,90}
n= 8
max = 245
nob = 10 (No of backet)
divider = ceil((m+1)/nob) = ceil((245+1)/nob)
= ceil(246/10) = ceil(24.6) = 25
j = floor(125/25) = 5 , so b[5] = 125
j = floor(67/25) = floor(2.68) = 2 , so b[2] = 67
j = floor(45/25) = floor(1.8) = 1 , so b[1] = 45
j = floor(3/25) = floor(0.12) = 0 , so b[0] = 3
j = floor(69/25) = floor(2.76) = 2 , so b[2] = 69
j = floor(245/25) = floor(9.8) = 9 , so b[9] = 245
j = floor(35/25) = floor(1.4) = 1 , so b[1] = 35
j = floor(90/25) = floor(3.6) = 3, so b[3] = 90

0	3	
1	45	35
2	67	69
3	90	
4		
5	125	
6		
7		
8		

9	245	

In the above array apply insertion sort in each row

0	3	
1	35	45
2	67	69
3	90	
4		
5	125	
6		
7		
8		
9	245	

Now concatenate all the row elements of b array
So sorted list is a = {3,35,45,67,69,125,245}

Complexity

T(n) = [time to insert *n* elements in array *A*] + [time to go through auxiliary array *B*[0 . . *n*-1] *

(Sort by INSERTION_SORT)

$= O(n) + (n-1) * (n)$

$= O(n) + n^2 - n$

$= O(n^2)$

Worse case $= O(n^2)$

Best case : $\Omega(n+k)$

Average case : $\Theta(n + k)$.

Therefore, the entire Bucket sort algorithm runs in linear expected time.

Counting Sort

Counting sort is an algorithm for sorting a collection of objects according to keys that are small integers; that is, it is an integer sorting algorithm. It is a linear time sorting algorithm used to sort items when they belong to a fixed and finite set.

The algorithm proceeds by defining an ordering relation between the items from which the set to be sorted is derived (for a set of integers, this relation is trivial).Let the set to be sorted be called A. Then, an auxiliary array with size equal to the number of items in the superset is

defined, say B. For each element in A, say e, the algorithm stores the number of items in A smaller than or equal to e in B(e). If the sorted set is to be stored in an array C, then for each e in A, taken in reverse order, C[B[e]] = e. Counting sort assumes that each of the n input elements is an integer in the range 0 to k. that is n is the number of elements and k is the highest value element.

Counting sort determines for each input element x, the number of elements less than x. And it uses this information to place element x directly into its position in the output array. Consider the input set : 4, 1, 3, 4, 3. Then n=5 and k=4

The algorithm uses three array:

Input Array: A[1..n] store input data where A[j] ∈ {1, 2, 3, …, k}

Output Array: B[1..n] finally store the sorted data

Temporary Array: C[1..k] store data temporarily

Counting Sort Example

Example 1 :

Given List :

A = { 2,5,3,0,2,3,0,3}

Step:1

A

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

C- Highest Element is 5 in the given array

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

B-Output Array

Step:2

C[A[J]]=C[A[J]]+1

C[A[1]]=C[2]=C[2]+1 . In the place of C[2] add 1.

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

Step:3 (Repeat the step C[A[j]]=C[A[j]]+1 until n value)

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

Step:4 C[i]=C[i]+C[i-1]

C

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

Intially C[0]=C[0]
= 2
C[1]=C[0]+C[1]
=2+0 =2
C[2]=C[1]+C[2]
=2+2 =4

Continued till the end of C array.

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

Sorted List: B

B[C[A[j]]] ←A[j]
C[A[j]] ←C[A[j]]-1

J=8 to 1
B[C[A[8]]]= A[8]
B[7]=3

1	2	3	4	5	6	7	8
						3	

B[C[A[7]]]= A[7]
B[2]=0

1	2	3	4	5	6	7	8
	0					3	

Continue still the j value reaches 1.

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

```
Algorithm
        Counting-sort(A,B,K)
        {
            for i←0 to k
                {
                        C[i] ←0
                }
                for j ←1 to length[A]
                {
                        C[A[j]] ←C[A[j]]+1
                }
                        // C[i] contains number of elements equal to i.
                for i ←1 to k
                {
                        C[i]=C[i]+C[i-1]
                }
                // C[i] contains number of elements ≤ i.
                for j ←length[A] downto 1
                {
                        B[C[A[j]]] ←A[j]
                        C[A[j]] ←C[A[j]]-1
                }
        }
```

Analysis of COUNTING-SORT(A,B,k)

```
Counting-sort(A,B,k)
{
for i←0 to k                                    Θ(k)
    {
    C[i] ←0
    }
for j ←1 to length[A]                           Θ(n)
    {
    C[A[j]] ←C[A[j]]+1
    }
// C[i] contains number of elements equal to i.
for i ←1 to k                                   Θ(k)
     {
     C[i]=C[i]+C[i-1]
      }
// C[i] contains number of elements ≤ i.
```

```
for j ←length[A] downto 1                    Θ(n)
    {
     B[C[A[j]]] ←A[j]
     C[A[j]] ←C[A[j]]-1
    }}
```

Complexity

How much time does counting sort requires?

- For loop of lines 1-2 takes time **Θ**(k).
- For loop of lines 3-4 takes time **Θ**(n).
- For loop of lines 6-7 takes time **Θ**(k).
- For loop of lines 9-11 takes time **Θ**(n).

  Thus the overall time is **Θ**(k+n).In practice we usually use counting sort when we have k =O(n), in which the running time is **Θ**(n).

Worst Case Complexity is O(n)

Average Case Complexity is **Θ**(n).

Best Case = Ω(n)