18MCA34C

DESIGN AND ANALYSIS OF ALGORITHM

UNIT III

Divide and conquer

FACULTY

Dr. K. ARTHI MCA, M.Phil., Ph.D.,

Assistant Professor,

Postgraduate Department of Computer Applications,

Government Arts College (Autonomous),

Coimbatore-641018.

Divide and conquer - general method

Both merge sort and quicksort employ a common algorithmic paradigm based on recursion. This paradigm, **divide-and-conquer**, breaks a problem into subproblems that are similar to the original problem, recursively solves the sub problems, and finally combines the solutions to the sub problems to solve the original problem. Because divide-and-conquer solves subproblems recursively, each subproblem must be smaller than the original problem, and there must be a base case for subproblems. You should think of a divide-and-conquer algorithm as having three parts:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the sub problems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. **Combine** the solutions to the subproblems into the solution for the original problem.

Binary search

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

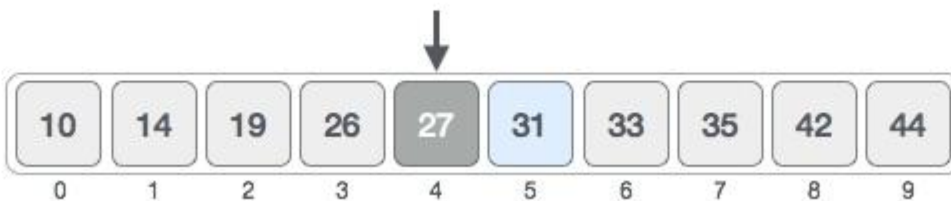
How Binary Search Works?

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

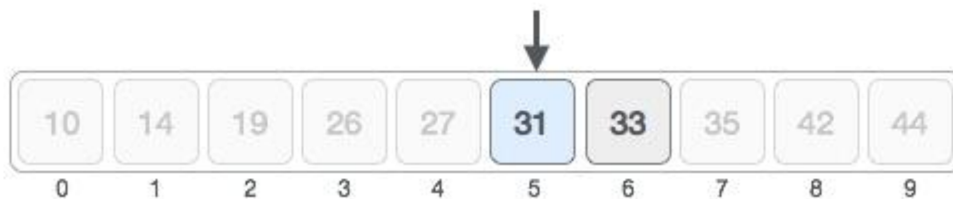
We change our low to mid + 1 and find the new mid value again.

low = mid + 1
mid = low + (high - low) / 2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Pseudocode

The pseudocode of binary search algorithms should look like this −

```
Procedure binary_search
   A ← sorted array
n ← size of array
x ← value to be searched

   Set lowerBound = 1
   Set upperBound = n

while x not found
ifupperBound<lowerBound
         EXIT: x does not exists.

setmidPoint = lowerBound + ( upperBound - lowerBound ) / 2

if A[midPoint] < x
setlowerBound = midPoint + 1

if A[midPoint] > x
setupperBound = midPoint - 1

if A[midPoint] = x
         EXIT: x found at location midPoint
end while

end procedure
```

merge sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following −

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

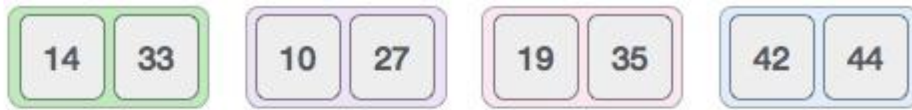
This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target

list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this −

Now we should learn some programming aspects of merge sorting.

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted order.

Pseudocode

We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions − divide & merge.

Merge sort works with recursion and we shall see our implementation in the same way.

```
proceduremergesort(var a as array )
if( n ==1)return a

var l1 as array = a[0]... a[n/2]
var l2 as array = a[n/2+1]... a[n]

  l1 =mergesort( l1 )
  l2 =mergesort( l2 )

return merge( l1, l2 )
end procedure

procedure merge(var a as array,var b as array )

var c as array
while( a and b have elements )
if( a[0]> b[0])
add b[0] to the endof c
remove b[0]from b
else
add a[0] to the endof c
remove a[0]from a
endif
endwhile

while( a has elements )
add a[0] to the endof c
remove a[0]from a
endwhile

while( b has elements )
```

add b[0] to the endof c
remove b[0]from b
endwhile

return c

end procedure

greedy algorithm

An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.

Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

Counting Coins

This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of □ 1, 2, 5 and 10 and we are asked to count □ 18 then the greedy prœcedure will be −

- **1** − Select one □ 10 coin, the remaining count is 8
- **2** − Then select one □ 5 coin, the remaining count is 3
- **3** − Then select one □ 2 coin, the remaining count is 1
- **4** − And finally, the selection of one □ 1 coins solves the problem

Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.

For the currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example, the greedy approach will use 10 + 1 + 1 + 1 + 1 + 1, total 6 coins. Whereas the same problem could be solved by using only 3 coins (7 + 7 + 1)

Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

ExamplesMost networking algorithms use the greedy approach. Here is a list of few of them –

- Travelling Salesman Problem
- Prim's Minimal Spanning Tree Algorithm
- Kruskal's Minimal Spanning Tree Algorithm
- Dijkstra's Minimal Spanning Tree Algorithm
- Graph - Map Coloring
- Graph - Vertex Cover
- Knapsack Problem
- Job Scheduling Problem

container loading problem

The greedy **algorithm** constructs the **loading** plan of a single **container** layer by layer from the bottom up. At the initial stage, the list of available surfaces contains only the initial surface of size L x W with its initial position at height 0.

knapsack problem

The **Knapsack** Problem is a famous Dynamic Programming Problem that falls in the optimization category. It derives its name from a scenario where, given a set of items with specific weights and assigned values, the goal is to maximize the value in a **knapsack** while remaining within the weight constraint.**Example** of a one-dimensional (constraint) **knapsack problem**: which boxes should be chosen to maximize the amount of money while still keeping the overall weight under or equal to 15 kg? A multiple constrained **problem** could consider both the weight and volume of the boxes.

THANK YOU

This content is taken from the text books and reference books prescribed in the syllabus.