Time Complexity: O(nlogn), since there will be one build heap, 2n - 2 delete mins, and n - 2 inserts, on a priority queue that never has more than n elements. Refer to the priority Queues chapter for details.

### 4.4.4 Problem

The following algorithm gives the optimal solution?

**Algorithm :** Merge the files in pairs. That means after the first step, the algorithm produces the n/2 intermediate files. For the next step, we need to consider these intermediate files and merge them in pairs and keep going.

**Note :** Sometimes this algorithm is called 2-way merging. Instead of two files at a time, if we merge K files at a time then we call is K -way merging.

- Solution: This algorithm will not produce the optimal solution and consider the previous example for a counter example. As per the above algorithm, we need to merge the first pair of files (10 and 5 size files), the second pair of files (100 and 50) and the third pair of files (20 and 15). As a result we gel the following list of files. {15, 150, 35}
- Similarly, merge the output in pairs and this step produces [below, the third element does not have a pair element, so keep it the same
- Finally, 165,35}, {185}
- The total cost of merging = Cost of all merging operations = 15 + 150 + 35 + 165 + 185 = 550. This is much more than 395 (of the previous problem). So, the given algorithm is not giving the best (optimal) solution.

**Review Questions**

Q. 1 Describe classifications of algorithm in detail. **(Refer sections 4.1.1, 4.1.1(a), 4.1.1(b), 4.1.1(c))**

Q. 2 Explain properties of greedy algorithms. **(Refer section 4.2.2)**

Q. 3 Write short note on advantages and disadvantages of greedy method. **(Refer section 4.2.3)**

Q. 4 Explain Huffman coding algorithm with example. **(Refer section 4.4.3)**

UNIT III

CHAPTER 5

# Divide and Conquer Algorithms

**Syllabus**

Introduction, What is Divide and Conquer Strategy? Divide and Conquer Visualization, Understanding Divide and Conquer, Advantages of Divide and Conquer, Disadvantages of Divide and Conquer, Master Theorem, Divide and Conquer Applications

**Syllabus Topic : Introduction**

## 5.1 Introduction

In the Greedy chapter, we have seen that for many problems the Greedy strategy failed to provide optimalsolutions. From those problems, there are some that can be easily solved by using the Divide and Conquer(D & C) technique. Divide and Conquer is a popular technique for algorithm design.

**Syllabus Topic : What is Divide and Conquer Strategy?**

### 5.1.1 Divide and Conquer Strategy

- This paradigm, **divide-and-conquer**, breaks a problem into subproblems that are similar to the original problem, recursively solves the subproblems, and finally combines the solutions to the subproblems 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.
- A divide-and-conquer algorithm can be described using three parts :
  1. Divide the problem into a number of subproblems that are smaller instances of the same problem.

2. Conquer the subproblems 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.

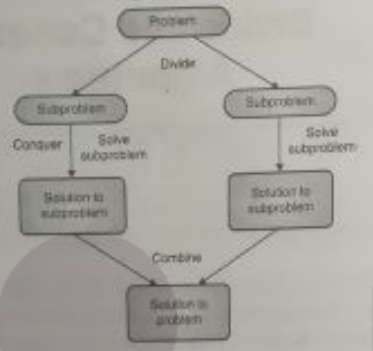


Fig. 5.1.1

- If we expand out two more recursive steps, it looks like this.

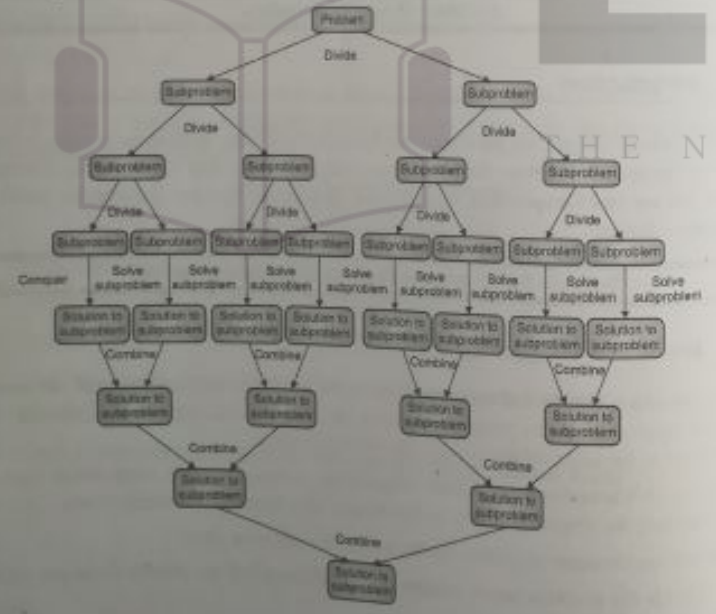


Fig. 5.1.2



- Because divide-and-conquer creates at least two subproblems, a divide-and-conquer algorithm makes multiple recursive calls.
- All the time it is not possible to solve all the problems with the Divide and Conquer technique. As per the definition of D and C, the recursion solves the subproblems which are of the same type.

**Syllabus Topic : Divide and Conquer Visualization**

## 5.2 Divide and Conquer Visualization

- For better understanding, consider the following visualization.
- Assume that n is the size of the original problem. As described above, we can see that the problem is divided into sub problems with each of size n/b (for some constant b).
- We solve the sub problems recursively and combine their solutions to get the solution for the original problem.

```
DivideAndConquer( P ):
if( small ( P) ):
 / / P is very small so that a solution is obvious
return solution ( n)
divide the problem P into k sub problems P1, P2, ... , Pk
return (
Combine (
DivideAndConquer(P1 ),
DivideAndConquer(P2 ),
    ....
DivideAndConquer(Pk )
    )
)
```

**Syllabus Topic : Understanding Divide and Conquer**

### 5.2.1 Understanding Divide and Conquer

- For a clear understanding of D and C, let us consider a story. There was an old man who was a rich farmer and had seven sons.
- He was afraid that when he died, his land and his possessions would be divided among his seven sons, and that they would become enemy of each another.

- So he gathered them together and showed them seven sticks that he had tied together and told them that anyone who could break the bundle would inherit everything. They all tried, but no one could break the bundle.
- Then the old man untied the bundle and broke the sticks one by one. The brothers decided that they should stay together and work together and succeed together. The moral for problem solvers is different. If we can't solve the problem, divide it into parts, and solve one part at a time.

**Syllabus Topic : Advantages of Divide and Conquer**

### 5.2.2 Advantages of Divide and Conquer

- The first, and probably most recognizable benefit of the divide and conquer paradigmis the fact that it allows us to solve difficult and often impossible looking problems such as the Tower of Hanoi, which is a mathematical game or puzzle. Being given a difficult problem can often be discouraging if there is no idea how to go about solving it.
- The divide and conquer method, it reduces the degree of difficulty since it divides the problem into sub problems that are easily solvable and usually runs faster than other algorithms would.
- It often plays a part in finding other efficient algorithms, and in fact it was the central role in finding the quick sort and merge sort algorithms.
- It also uses memory caches effectively. The reason for this is the fact that when the sub problems become simple enough, they can be solved within a cache, without having to access the slower main memory, which saves time and makes the algorithm more efficient.
- And in some cases, it can even produce more precise outcomes in computations with rounded arithmetic than iterative methods would.
- Packaged with all of these advantages, however, are some weaknesses in the process.

**Syllabus Topic : Disadvantages of Divide and Conquer**

### 5.2.3 Disadvantages of Divide and Conquer

- One of the most common issues with this sort of algorithm is the fact that the recursion is slow, which in some cases outweighs any advantages of this divide and conquer process.
- Another concern with it is the fact that sometimes it can become more complicated than a basic iterative approach, especially in cases with a large n.
- In other words, if someone wanted to add a large amount of numbers together, if they just create a simple loop to add them together, it would turn out to be a much simpler approach



than it would be to divide the numbers up into two groups, add these groups recursively, and then add the sums of the two groups together.

- Another downfall is that sometimes once the problem is broken down into sub problems, the same sub problem can occur many times.
- In cases like these, it can often be easier to identify and save the solution to the repeated subproblem, which is commonly referred to as memorization.
- And the last recognizable implementation issue is that these algorithms can be carried out by a non-recursive program that will store the different sub problems in things called explicit stacks, which gives more freedom in deciding just which order the sub problems should be solved.
- These implementation issues do not make this process a bad decision when it comes to solving difficult problems, but rather this paradigm is the basis of many frequently used algorithms

**Syllabus Topic : Master Theorem**

## 5.3 Master Theorem

- We assume a divide and conquer algorithm in which a problem with input size n is always divided into a subproblems, each with input size n/b. Here a and b are integer constants with $a \geq 1$ and $b > 1$.
- We assume n is a power of b, say $n = b^k$.
- Otherwise at some stage we will not be able to divide the sub-problem size exactly by b.
- However, the Master Theorem still holds if n is not a power of b, and the subproblem input sizes are [n/b] or [n/b]

  Note $k = \log_b(n)$
- The recurrence for the running time is :

  $$T(n) = a\,T(n/b) + f(n),\ T(1) = d$$
- Here f(n) represents the divide and combine time (i.e., the non-recursive time). f(n) may involve $\theta$, e.g., $f(n) = \theta(n^2)$.

  We define $E = \log_b(a)$.

  E is called the critical exponent. (It strongly influences the solution) By definition, $b^E = a$.

  Note that $a^k = n^E$

  Why ? $a^k = (b^E)^k = (b^k)^E = n^E$.
- We can write down the total time to solve all sub-problems at a given depth in the recursion tree.

| Depth of recursion | Size of sub-problems | Number of sub-problems | Total (non-recursive) time at this depth is roughly proportional to |
|---|---|---|---|
| 0 | n | 1 | f(n) |
| 1 | n/b | a | af (n/b) |
| 2 | $n/b^2$ | $a^2$ | $a^2 f(n/b^2)$ |
| 3 | $n/b^3$ | $a^3$ | $a^3 f(n/b^3)$ |
| ⋮ | ⋮ | ⋮ | ⋮ |
| k – 2 | $n/b^{k-2}$ | $a^{k-2}$ | $a^{k-2} f(n/b^{k-2})$ |
| k – 1 | $n/b^{k-1} = b$ | $a^{k-1}$ | $a^{k-1} f(n/b^{k-1}) = \theta(n^E)$ |
| k | $n/b^k = 1$ | $a^k = n^E$ | $a^k d = O(n^E)$ |

$T(n)$ = Sum of terms in rightmost column above

$= f(n) + af(n/b) + a^2 f(n/b)^2 + \ldots + a^{k-1} f(n/b^{k-1}) + a^k d$

The critical functions in determining T(n) are :

**(i) f(n) : (the non-recursive time at depth 0)**

**(ii) $n^E$ : (the non-recursive time at depth k, or k – 1).**

Clearly : $T(n) \geq \theta(\max(n^E, f(n)))$.

On the other hand, if the terms in the right hand column of the table either increase as we move down, or decrease as we move down,

then : $T(n) \leq \theta(\max(n^E, f(n)) \cdot \log_b(n))$.

We will see that, if one of $n^E$ and f(n) grows much more rapidly than the other, then $T(n) \leq \theta$ (more rapidly growing function).

(1) $f(n)$ in $O(n^{E-\epsilon})$ for fixed $\epsilon > 0$ implies $T(n) = \theta(n^E)$.

(2) $f(n)$ in $\theta(n^E)$ implies $T(n) = \theta(n^E \log_b(n))$.

(3) $f(n)$ in $\Omega(n^{E+\epsilon})$ for fixed $\epsilon > 0$ implies $T(n) = \theta(f(n))$.

Actually, (3) requires an additional hypothesis, that typically holds.

Note none of these cases may apply. For example, if $f(n) = n^E \log_b(n)$, we are between cases (2) and (3) : neither case holds.



**Syllabus Topic : Divide and Conquer Applications**

## 5.4 Divide and Conquer Applications

Following are some problems, which are solved using divide and conquer approach.

1. Finding the maximum and minimum of a sequence of numbers
2. Strassen's matrix multiplication
3. Merge sort
4. Binary search

Let us consider a simple problem that can be solved by divide and conquer technique.

**Example 5.4.1**

Statement : The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.

**Solution :**

To find the maximum and minimum numbers in a given array numbers[] of size n, the following algorithm can be used. First we are representing the naive method and then we will present divide and conquer approach.

**Naïve Method**

Naïve method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

**Algorithm : Max-Min-Element (numbers[])**

```
max := numbers[1]
min := numbers[1]
for i = 2 to n do
if numbers[i] > max then
max := numbers[i]
if numbers[i] < min then
min := numbers[i]
return (max, min)
```

**Analysis**

- The number of comparison in Naive method is 2n - 2.
- The number of comparisons can be reduced using the divide and conquer approach. Following is the technique.

## Divide and Conquer Approach

In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

In this given problem, the number of elements in an array is y−x+1y−x+1, where y is greater than or equal to x.

Max–Min(x, y)Max–Min(x, y) will return the maximum and minimum values of an array numbers[x...y]numbers[x...y].

### Algorithm

```
Max - Min(x, y)
if x – y ≤ 1 then
return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y]))
else
  (max1, min1) = maxmin(x, ⌊((x + y)/2)⌋)
  (max2, min2) = maxmin(⌊((x + y)/2) + 1⌋,y)
return (max(max1, max2), min(min1, min2))
```

### Analysis

Let T(n) be the number of comparisons made by Max – Min(x,y) Max – Min(x,y), where the number of elements n = y – x + 1n = y – x + 1.

If ***T(n)*** represents the numbers, then the recurrence relation can be represented as

$$T(n) = \begin{cases} T\left(\frac{1}{2}\right) + T\left(\frac{1}{2}\right) + 2 & \text{for } n > 2 \\ 1 & \text{for } n = 2 \\ 0 & \text{for } n = 1 \end{cases}$$

Let us assume that n is in the form of power of 2. Hence, n = 2k where k is height of the recursion tree. So,

$$\begin{aligned} T(n) &= 2.T(n^2) + 2 = 2.(2.T(n^4) + 2) + 2.... \\ &= 3n^2 - 2T(n) = 2.T(n^2) + 2 = 2.(2.T(n^4) + 2) + 2..... \\ &= 3n^2 - 2 \end{aligned}$$

Compared to Naive method, in divide and conquer approach, the number of comparisons is less. However, using the asymptotic notation both of the approaches are represented by O(n).

This chapter, we will discuss merge sort and analyze its complexity.



### Example 5.4.2

Statement : The problem of sorting a list of numbers lends itself immediately to a divide-and-conquer strategy: split the list into two halves, recursively sort each half, and then merge the two sorted sub-lists.

**Solution :**

In this algorithm, the numbers are stored in an array numbers[]. Here, p and q represents the start and end index of a sub-array.

### Algorithm

```
Merge-Sort (numbers[], p, r)
if p < r then
q = ⌊(p + q) / 2⌋
Merge-Sort (numbers[], p, q)
  Merge-Sort (numbers[], q + 1, r)
  Merge (numbers[], p, q, r)
Function: Merge (numbers[], p, q, r)
n1 = q – p + 1
n2 = r – q
declare leftnums[1...n1 + 1] and rightnums[1...n2 + 1] temporary arrays
for i = 1 to n1
leftnums[i] = numbers[p + i - 1]
for j = 1 to n2
rightnums[j] = numbers[q + j]
leftnums[n1 + 1] = ∞
rightnums[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
if leftnums[i] ≤ rightnums[j]
numbers[k] = leftnums[i]
  i = i + 1
else
numbers[k] = rightnums[j]
  j = j + 1
```

### Analysis

Let us consider, the running time of Merge-Sort as ***T(n)***. Hence,

$$T(n) = \begin{cases} c & \text{if } n \le 1 \\ 2xT\left(\frac{n}{2}\right) + dxn & \text{otherwise} \end{cases}$$

Where c and b are constants

Therefore, using this recurrence relation,

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + i.d.n$$

As $i = \log n$,

$$T(n) = 2^{\log n} T\left(\frac{n}{2^{\log n}}\right) + \log n.d.n$$

$$= c.n + d.n.\log n$$

Therefore, $T(n) = O(n \log n)$

**Example**

In the following example, we have shown Merge-Sort algorithm step by step. First, every iteration array is divided into two sub-arrays, until the sub-array contains only one element. When these sub-arrays cannot be divided further, then merge operations are performed.

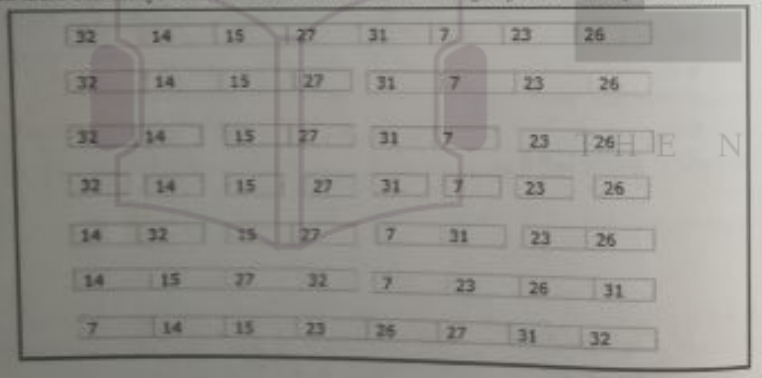


Fig. P. 5.4.2

**Example 5.4.3**

Let we consider an algorithm 'A' which solves problems by dividing them into five sub problems of half the size, recursively solving each sub problem, and then combining the solutions in linear time. What is the complexity of this algorithm?

**Solution :**

Let us assume that the input size is 'n' and T(n) defines the solution to the given problem.



As per the description, the algorithm divides the problem into 'S' sub problems with each of size (n/2). So we need to solve 5T(n/2) subproblems. After solving these sub problems, the given array (linear time) is scanned to combine these

**Solution :**

The total recurrence algorithm for this problem can be given as:

$$T(n) = 5T(n/2) + O(n).$$

Using the Master theorem (of D &C), we get the complexity as

$$O(n^{\log(5/2)}) = O(n^{2+}) = O(n^2).$$

**Example 5.4.4**

Similar to above Problem, An algorithm B solves problems of size n by recursively solving two sub problems of size n-1 and then combining the solutions in constant time. What is the complexity of this algorithm?

**Solution :**

Let us assume that the input size is n and T(n) defines the solution to the given problem. As per the description of algorithm we divide this problem into 2 sub problems with each of size n - 1. So we have to solve 2T(n - 1) subproblems. After solving these sub problems, the algorithm takes only a constant time to combine these solutions. The total recurrence algorithm for this problem can be given as:

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

Using Master theorem (of Divide and Conquer), we get the complexity as $O(n^{\log 2/2^0}) = O(2^n)$. (Refer to Introduction chapter for more details)

**Example 5.4.5**

Write a recurrence and solve it.

```
def function(n):
if(n > 1):
print((*n))
function(n/2)
function(n/2)
```

**Solution :**

Let us assume that input size is n and T(n) defines the solution to the given problem. As per the given code, after printing the character and dividing the problem into 2 subproblems with each of size (n/2) and solving them. So we need to solve 2T(n/2) subproblems. After solving these subproblems, the algorithm is not doing anything for combining the solutions. The total recurrence algorithm for this problem can be given as:

$$T(n) = 2T(n/2) + O(1)$$

Using Master theorem (of D & C), we get the complexity as $O(n^{\log(2/2)}) = O(n^1) = O(n)$.

**Example 5.4.6**

Stock Pricing Problem

Consider the stock price of Career Monk.com in n consecutive days. That means the input consists of an array with stock prices of the company. We know that the stock price will not be the same on all the days. In the input stock prices there may be dates where the stock is high when we can sell the current holdings, and there may be days when we can buy the stock. Now our problem is to find the day on which we can buy the stock and the day on which we can sell the stock so that we can make maximum profit.

**Solution :**

As given in the problem, let us assume that the input is an array with stock prices [integers]. Let us say the given array is A[1], .... A[n]. From this array we have to find two days [one for buy and one for sell] in such a way that we can make maximum profit. Also, another point to make is that the buy date should be before sell date. One simple approach is to look at all possible buy and sell dates.

```
def calculateProfitWhenBuyingNow(A, index):
buyingPrice = A[index]
maxProfit = 0
sellAt = index
for i in range(index + 1, len(A)):
sellingPrice = A[i]
profit = sellingPrice-buyingPrice
if profit >maxProfit:
maxProfit = profit
sellAt = i
return maxProfit, sellAt
#check all possible buying times
def StockStrategyBruteForce(A):
maxProfit =None
buy = None
sell = None
for index, item in enumerate(A):
profit, sellAt = calculateProfitWhenBuyingNow(A , index)
if (maxProfit is None) or (profit >maxProfit):
maxProfit = profit
buy= index
sell = sellAt
return maxProfit, buy, sell
```

The two nested loops take $n(n + 1)/2$ computations, so this takes time $\Theta(n^2)$.



## Review Questions

Q.1 Write short note on Divide and Conquer strategy. (Refer section 5.1.1)

Q.2 Explain advantages of divide and conquer. (Refer section 5.2.2)

Q.3 Explain disadvantages of divide and conquer. (Refer section 5.2.3)

Q.4 Describe master theorem in detail. (Refer section 5.3)

❑❑❑