Divide and Conquer Algorithm

Diaa eldeen Amin
Faculty Of Computer Science
Misr International University
diaaeldeen2306246@miuegypt.edu.eg

Mariam Khaled
Faculty Of Computer Science
Misr International University
mariam2303748@miuegypt.edu.eg

Sama Adel
Faculty Of Computer Science
Misr International University
sama2301973@miuegypt.edu.eg

Ritag Raouf
Faculty Of Computer Science
Misr International University
ritag2300230@miuegypt.edu.eg

Mai Moheb
Faculty Of Computer Science
Misr International University
mai2308305@miuegypt.edu.eg

Ashraf Abdel Raouf
Faculty Of Computer Science
Misr International University
ashraf.raouf@miuegypt.edu.eg

***Abstract*—This paper is about comparing between two algorithms with the same approach that follows divide and conquer, but in the first why do we study Algorithms and what is the meaning of approach?? Algorithms are a way to think about solving the problem to choose the best way with the fastest time to solve this problem because sometimes there are more than one solution that leads to the same solution but the shortest one is always the easiest one. On the other hand the approaches mean all ways to solve the same problems using many approaches. here in this paper we will discuss how the two algorithms work and what is there time complexity and we will compare them together to see if they are good ways to solve those problem or not we will also discuss there function and talk about how its work to understand how efficient it is.**

I. Divide Conquer

A problem-solving strategy that breaks down a complex problem into smaller, more manageable subproblems, solves those subproblems, and then combines the solutions to find a solution for the original problem.

Divide and conquer is patterned after the brilliant strategy employed by the French emperor Napoleon in the Battle of Austerlitz on December 2, 1805. A combined army of Austrians and Russians outnumbered Napoleon's army by about 15,000 soldiers. The Austro-Russian army launched a massive attack against the French right flank. Anticipating their attack, Napoleon drove against their center and split their forces into two. Because the two smaller armies were individually no match for Napoleon, they each suffered heavy losses and were compelled to retreat. By dividing the large army into two smaller armies and individually conquering these two smaller armies, Napoleon was able to conquer the large army.

The divide-and-conquer approach employs this same strategy in a problem instance. That is, it divides an instance of a problem into two or more smaller instances. The smaller instances are usually instances of the original problem. If solutions to smaller instances can be obtained readily, then the solution to the original instance can be obtained by combining these solutions. If the smaller instances are still too large to be solved readily, they can be divided into still smaller instances. This process of dividing the instances continues until they are so small that a solution is readily obtainable.

The divide-and-conquer approach is a top-down approach. That is, the solution to a top-level instance of a problem is obtained by going down and obtaining solutions to smaller instances. The reader may recognize this as the method used by recursive routines. Recall that when writing a recursion, one thinks at the problem-solving level and lets the system handle the details of obtaining the solution (by means of stack manipulations). When developing a divide-and-conquer algorithm, we usually think at this level and write it as a recursive routine. After this, we can sometimes create a more efficient iterative version of the algorithm. Divide-and-conquer has too many approaches like (Binary search, merge sort, quick sort, etc.)

We now introduce the divide-and-conquer approach with examples, starting with Fast Exponentiation.

II. Introduction

The first algorithm we will discuss is Fast Exponentiation, which uses the Divide and Conquer paradigm. Fast exponentiation efficiently computes x^n, where x is a base and n is a non-negative integer exponent, by reducing the number of multiplications required. The algorithm is about a method that works by breaking a large problem into smaller parts and solving each one and neglecting the other half because when you try to solve n^{10} you can just solve n^5 and multiply it by itself, and so on when you try to solve n^5 ..., With this way to solve this problem it will be so much faster to solve than trying to solve n^{10}

For an example to the form of the mathematical equation that we have, to compute x^n:

- If n is even, compute $x^{n/2}$ recursively and square the result: $x^n = (x^{n/2})^2$.
- If n is odd, compute x^{n-1} recursively and multiply by x: $x^n = x \cdot x^{n-1}$.

III. Fast Exponentiation Functionality

The divide-and-conquer algorithm for fast exponentiation:

$$x^n = \begin{cases} (x^{n/2})^2, & \text{if } n \text{ is even,} \\ x \cdot (x^{(n-1)/2})^2, & \text{if } n \text{ is odd.} \end{cases}$$

At each recursive step, the exponent n is divided into (n/2) and check after the division if the new number is equal to zero or no, if the new number is equal to zero it should return 1 because when you calculate a number with exponent zero it gives you one. otherwise if the new number is still above the zero the function will call it self again with a half of the new number and continue with divide the n/2 till it reach a zero.

Now after we reach the number zero and it return a 1 the code will continue to check if the exponent modulus 2 is equal to zero or no if it equal to zero that means that the exponent were an even number and if it isn't equal to zero that means it was an odd number. Now if the number was even it should return the number we calculate multiplied by itself and if it was odd it should return the number we calculate multiplied by it self multiplied by the base.

Example by a number odd to explain why multiplied the number we calculated by itself multiplied by the base:
lets assume that the base is : 2
lets assume that the exponent is : 5
now according to our base case the exponent isn't equal to zero so we will call the function again with the exponent/2 the new exponent will be equal 2, so the exponent is 2 but the big exponent is 5?? but we still not reach our base case which the exponent should be equal to zero but when we reach it we will have to multiple the base which is 2 time the 2^2 and multiply them also by itself so the final result will be 2^2 * 2^2 * 2 and that all will be equal to 2^5 which is what we look for.

Listing 1. C++ implementation of recursive fast exponentiation

```
long long fastExp(long long base, long long
    exponent) {
    if (exponent == 0) {
        return 1;  // Base case: x^0 = 1
    }
    long long half = fastExp(x, n / 2);
    if (exponent % 2 == 0) {
        // If exponent is even: x^n = (x^(n/2)
            )^2
        return half * half;
    } else {
        // If exponent is odd: base^exponent =
             base * (base^((exponent-1)/2))^2
        return base * half * half;
    }
}
```

IV. Time Complexity Analysis

According to our function the if function is equal to O(1) same as all lines except the line of calling function recursively (long long half = fastExp(x, n / 2);) that line will give us a time complexity equal to $T(n/2)$. This leads to the recurrence:

$$T(n) = T\left(\frac{n}{2}\right) + c,$$

where c is a constant representing the non-recursive work. Unrolling this recurrence:

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= T(n/4) + 2c \\ &= \dots \\ &= T(1) + c \log_2 n = O(\log n). \end{aligned}$$

by assuming that the n is equal to 8 the result will be also:

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= T(n/4) + 2c \\ &= T(n/8) + 4c \\ &= T(1) + c \log_2 8 = O(\log 8). \end{aligned}$$

And no matter how the number is it will always be a log(n) Thus, the fast exponentiation algorithm runs in $\Theta(\log n)$ time. And if we want to try another way to calculate the time complexity to be sure, by the Master Theorem for divide-and-conquer recurrences with $a = 1$, $b = 2$, and $f(n) = \Theta(1)$, we also conclude $T(n) = \Theta(\log n)$.

V. Applications

The fast exponentiation algorithm is fundamental in computer science and is used in many applications requiring efficient power computation such as :

- Game Development: Optimization of Calculations: In game development, calculations related to physics simulations, game logic, and artificial intelligence can be optimized using fast exponentiation for certain types of calculations.
- Numerical Analysis and Linear Algebra: Matrix Exponentiation: Many problems in linear algebra, like solving systems of differential equations, can be simplified by finding the exponential of a matrix. Fast exponentiation allows for efficient calculation of these matrix powers.

VI. Conclusion

To conclude, This algorithm helps us to solve n^x by calling the function recursively with the new value if $x/2$ and if the value of the current x is odd he returns n * n^x* n^x which x is the current x and if the value of the current x is even it should return n^x* n^x which x is the current x and all of that will be in a time complexity $log(n)$. Fast Exponentiation is a clear example of how the divide and conquer approach can significantly optimize algorithm performance. By reducing

the exponentiation problem size in half at each recursive step, the algorithm achieves a logarithmic time complexity of (log(n)), making it far more efficient than the naive iterative method. Through mathematical analysis and C++ implementation, we observed how the algorithm handles both even and odd exponents using simple but powerful logic. This recursive method not only improves computational speed but also forms the foundation for solving more advanced problems such as modular exponentiation in cryptography and matrix powers in numerical algorithms. In summary, Fast Exponentiation demonstrates how divide and conquer techniques can transform basic mathematical operations into highly optimized algorithms.

Peak Element in Array

VII. Introduction

The second algorithm which is (peak element in array) is an algorithm that try to get a number which is bigger than it's neighbors on the left and right only not to be the biggest element in array. By dividing the current array into two parts after comparing the middle element in array with his neighbors to find if he is bigger than they or no if he bigger then return the middle if not the algorithm will search on each part to find the one which is bigger than its neighbors .Using divide and Conquer with the algorithm peak element in array will decrease the time complexity which applies the concept of Algorithms of trying to reach the optimum solution in the least time if possible so according to the paper you will find that the time complexity of this algorithm will be $log(n)$, So this algorithm is especially effective.

For example :
input: [1, 3, 20, 4, 1, 0]
Middle index = 2 $\rightarrow$ value is 20
Left = 3, Right = 4
20 is greater than both neighbors, so it's a peak
Output: Index 2 (value 20)

VIII. Peak Element in Array

A peak element in an array is an element that is greater than or equal to its neighbors. Formally, for an array A of length n, an element $A[i]$ is a peak if:

$$A[i] \geq \begin{cases} A[i-1], & \text{if } i > 0, \\ A[i+1], & \text{if } i < n-1. \end{cases}$$

At each recursive step, the function calculate the middle of the array and compare it with the neighbor if it bigger than the element before and the element after and if the middle is bigger the function should return the middle as a peak and the index of the array.

The base case in that code is to try to find the number that is bigger than the element in its right and left so if it isn't the bigger it should divide the array into two parts but before that it will make a check on the number on the left side if it's bigger than middle or not if that is true it will call the function recursively and send the left side of the array as a new array and resize its size by making the size equal $middle - 1$ and if it's not bigger than the middle the function will neglect the rest of the array on the left hand side.

The other base case, if no peak element is found it should compare the last element in array with the value of the size - 1 in the array if the last element is bigger it will return the last number and do the same for the first element if he is bigger than its neighbor on the right it should return the first but we are looking for the first peak the shown up not all peaks.

So according to our cases we have 3 basic situations for that code and for examples:
1) on the regular situation the middle is the peak $(1, 3, 20, 4, 1, 0)$

size = 6
middle in index = 3
middle value is 20 and with comparing 20 with the neighbors (3,4) the 20 is bigger so it is a peak

2) first element is the peak of the array 10, 9, 8, 7, 6

size = 5
middle in index = 2
middle value is 9 and with comparing 9 with the neighbors (10,8) the 9 is bigger than 8 but not bigger that 10 so it is not a peak

the new array on the left if the first bigger than second element in array it should return the first. So, first element is 10 and the second is 9 by comparing them the 10 is bigger so the peak is 10

3) last element is the peak of the array 1, 2, 3, 4, 5 size = 5
middle in index = 2
middle value is 2 and with comparing 2 with the neighbors (1,3) the 2 is bigger than 1 but not bigger that 3 so it is not a peak

the new array on the right if the last bigger than before the last element in array it should return the last. So, last element is 5 and the before last is 4 by comparing them the 5 is bigger so the peak is 5

Listing 2. C++ implementation of recursive peak element in array

```
int findPeakElement(const vector<int>& arr,
    int left, int right) {
    int mid = left + (right - left) / 2;
```

```
// Check if mid is a peak
if ((mid == 0 || arr[mid - 1] <= arr[mid])
     &&
     (mid == arr.size() - 1 || arr[mid] >=
         arr[mid + 1])) {
     return mid;
}

// If the left neighbor is greater, then
    the peak must be in the left half
if (mid > 0 && arr[mid - 1] > arr[mid]) {
     return findPeakElement(arr, left, mid
         - 1);
}

// Otherwise, the peak is in the right
    half
return findPeakElement(arr, mid + 1, right
    );}
```

IX. Time Complexity Analysis

According to our function the if function is equal to O(1) same as all lines except the line of calling function recursively (return findPeakElement(arr, left, mid - 1);) or (return findPeakElement(arr, left, mid - 1);) but it will not calling a function N time at a time those lines will give us a time complexity equal to $T(n/2)$. This leads to the recurrence:

$$T(n) = T\left(\frac{n}{2}\right) + c,$$

where c is a constant representing the non-recursive work. Unrolling this recurrence:

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= T(n/4) + 2c \\ &= \dots \\ &= T(1) + c\log_2 n = O(\log n). \end{aligned}$$

by assuming that the n is equal to 8 the result will be also:

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= T(n/4) + 2c \\ &= T(n/8) + 4c \\ &= T(1) + c\log_2 8 = O(\log 8). \end{aligned}$$

And no matter how the number is it will always be a log(n) Thus, the fast exponentiation algorithm runs in $\Theta(\log n)$ time. And if we want to try another way to calculate the time complexity to be sure, by the Master Theorem for divide-and-conquer recurrences with:
$a = 1$, $b = 2$, and $f(n) = \Theta(1)$
It will be the second case where $a = b^d$ and it will be equal $n^d * log(n)$ and according that d equal to zero so, We also conclude $T(n) = \Theta(\log n)$.

X. Applications

The peak element in array algorithm is fundamental in computer science and is used in many applications requiring efficient power computation such as :

- Audio Music Analysis (e.g., Spotify, Shazam)
  What it does: Detects loudest beats or musical highlights.
  Why it matters: Peak detection is used to find rhythm, cut music into segments, or detect melodies quickly.
  Real-life effect: Helps music apps recognize or categorize songs instantly.

- Mobile Health Apps (e.g., Apple Health, smartwatches)
  What it does: Detects peak heart rate in fitness sessions.
  Why it matters: Peak detection helps track the most intense workout moment.
  Real-life effect: Tells you when you reached "fat-burning zone" or overexerted.

- 3. Network Signal Strength
  What it does: Finds the strongest signal point among data points.
  Why it matters: Devices use this to choose the best tower or connection.
  Real-life effect: Your phone automatically connects to the best Wi-Fi or cellular signal.

XI. Conclusion

The peak element in array algorithm is a practical and efficient solution that demonstrates how divide and conquer can simplify searching problems. By narrowing the search space in half at each step, the algorithm is able to find a peak element in logarithmic time without the need to scan the entire array. This makes it highly efficient even for large datasets.

Through various examples, we've seen how the algorithm identifies a peak based on comparisons with neighboring elements. Whether the peak appears at the beginning, middle, or end of the array, the logic ensures that at least one peak is always found. What makes this algorithm especially useful is its flexibility—it works without requiring the array to be sorted and still guarantees a peak result.

In real life, the idea of finding a peak applies to many technologies we rely on every day, such as detecting the highest heart rate during exercise, identifying signal strength in networks, or spotting spikes in sound waves or data patterns. The simplicity of the algorithm, combined with its power and speed, shows how well-designed logic can make a big impact in practical computing tasks.

Comparison between the two Algorithms

XII. Comparison and Results

Both Fast Exponentiation and Peak Element algorithms apply the divide-and-conquer approach but serve distinct purposes. Fast Exponentiation is designed for efficient power computation, while Peak Element focuses on locating a value greater than or equal to its neighbors in an array. The following tables compare their performance and algorithmic characteristics.

A. Performance Analysis

TABLE I
Performance Comparison of Fast Exponentiation vs Peak Element

Input Size	Operation	Fast Exponentiation	Peak Element
8	Time	0.001 ms	0.001 ms
16	Time	0.002 ms	0.003 ms
32	Time	0.004 ms	0.005 ms
64	Time	0.009 ms	0.008 ms
128	Time	0.015 ms	0.012 ms
256	Time	0.030 ms	0.018 ms

B. Algorithmic Characteristics

TABLE II
Algorithmic Comparison of Fast Exponentiation and Peak Element

Aspect	Fast Exponentiation	Peak Element
Problem Type	Mathematical	Array Search
Time Complexity	$O(\log n)$	$O(\log n)$
Space Complexity	$O(\log n)$	$O(\log n)$
Implementation	Multiplication logic	Comparison logic
Output	Numeric result	Index of peak
Real-life Usage	Cryptography, Math	Health, Signals, AI
Optimization Strategy	Recursive reduction	Divide and locate peak

The Fast Exponentiation and Peak Element algorithms serve as compelling demonstrations of the divide-and-conquer paradigm, a cornerstone of efficient algorithm design in computer science. By breaking complex problems into smaller, manageable subproblems, these algorithms achieve remarkable performance improvements, as evidenced by their logarithmic time complexity of $O(\log n)$. This efficiency stems from their ability to halve the problem size at each recursive step, significantly reducing computational overhead compared to naive approaches.

Fast Exponentiation optimizes the computation of x^n, where x is a base and n is a non-negative integer exponent, by minimizing the number of multiplications required. Instead of performing n multiplications, the algorithm recursively computes $x^{n/2}$ and squares the result for even exponents or adjusts for odd exponents by incorporating an additional multiplication. This approach, as illustrated in the C++ implementation, results in a time complexity of $\Theta(\log n)$, making it a critical tool in applications requiring rapid power calculations. For instance, in cryptography, Fast Exponentiation underpins modular exponentiation, enabling secure data encryption and decryption. Similarly, in numerical analysis, it facilitates efficient matrix exponentiation for solving differential equations, and in game development, it optimizes calculations for physics simulations and artificial intelligence.

The Peak Element algorithm, on the other hand, addresses a different but equally practical problem: finding an element in an array that is greater than or equal to its neighbors. By leveraging divide-and-conquer, it examines the middle element of the array and compares it with its neighbors, recursively narrowing the search to the left or right half based on which side is more likely to contain a peak. This strategy, also achieving $O(\log n)$ time complexity, is particularly effective because it does not require the array to be sorted, making it versatile for unsorted datasets. The algorithm's real-world applications are diverse, ranging from detecting peak heart rates in mobile health apps to identifying signal strength peaks in network communications and analyzing audio data for music recognition systems like Spotify or Shazam. These applications highlight the algorithm's ability to deliver fast, reliable results in dynamic, real-time environments.

When comparing the two algorithms, their shared reliance on divide-and-conquer yields similar time and space complexities—both operate in $\Theta(\log n)$ time and $O(\log n)$ space due to recursive call stacks. However, their implementation logic and problem domains differ significantly. Fast Exponentiation employs multiplication-based logic to compute a numeric result, while Peak Element uses comparison-based logic to return the index of a peak. The performance analysis, as shown in the comparison tables, reveals that both algorithms scale efficiently with input size, though Peak Element tends to be slightly faster for larger inputs due to the simplicity of comparison operations compared to multiplication. For example, at an input size of 256, Peak Element executes in 0.018 ms, compared to 0.030 ms for Fast Exponentiation, highlighting subtle differences in their computational demands.