ChatGPT

# Week 3 Algorithm Study Session: Brute Force, Recursion, Divide & Conquer, Backtracking

Before diving into new topics, let's briefly reflect on **Week 2**. Last session, we focused on fundamental **data structures** (like arrays, linked lists, stacks, queues, sets, maps, etc.) and **string manipulation** techniques. We compared structures such as arrays vs. linked lists – noting that arrays offer O(1) random access but expensive insertions/deletions, whereas linked lists allow fast insert/delete but only sequential access [1]. We also practiced common string algorithms, e.g. checking if a string is a **palindrome** (reads the same forwards and backwards) [2] or if two strings are **anagrams** of each other (contain the same characters in a different order) [3]. These foundations in data handling and manipulation set the stage for tackling algorithmic problems. Building on that, in **Week 3** we shift focus to core algorithm design paradigms – brute force, recursion, divide-and-conquer, and backtracking. Mastering these techniques will arm you with multiple approaches to solve coding interview problems effectively.

## Brute Force (완전탐색)

**Brute force** (또는 완전탐색) is the most straightforward problem-solving approach: simply try all possibilities until a solution is found. A brute force algorithm **systematically explores every potential candidate solution** to check if it satisfies the problem's requirements [4]. In other words, it does not use any shortcuts or heuristics – it exhaustively generates and tests all outcomes until the answer is discovered. This approach **guarantees** finding a solution if one exists (or determining none exists), because it doesn't miss any possibilities. Brute force is thus a valid baseline or a last resort: as Ken Thompson famously put it, "When in doubt, use brute force." However, the **cost** of this guarantee is very high, since the number of candidate solutions often grows explosively with input size (a phenomenon known as combinatorial explosion). For example, a brute-force solution to the 8-Queens puzzle would try all $64 \choose 8$ arrangements of queens on a chessboard [5]; a brute-force traveling salesman solution would enumerate all $N!$ permutations of $N$ cities. Such approaches become **impractical for large N** because their time complexity is often exponential or factorial [6]. In short, brute force trades efficiency for simplicity and certainty.

Despite its inefficiency, brute force has some **use cases** and merits: - It's easy to implement and reason about – useful for an initial solution or as a benchmark to compare optimized algorithms [7]. - It works well when the problem size is small enough to be explored in reasonable time [8]. In those cases, the simplicity can be an advantage. - It can be used in critical applications where correctness is paramount and any heuristic or complex optimization might introduce risk; brute force will always find the correct answer eventually [9]. - It can serve as a **baseline solution** during interviews: starting with a brute force solution shows understanding of the problem, and then you can discuss how to optimize it.

**Time Complexity Analysis:** Brute force algorithms often have the worst-case time complexity among all methods. A brute force search might be $O(n)$ (for a trivial linear scan), but many brute force solutions are polynomial of high degree or even **exponential** in time. For instance, brute forcing all subsets of an $n$-element set is $O(2^n)$; checking all permutations is $O(n!)$ [10] $. Because of this, brute force is typically feasible only for small input sizes or when combined with pruning techniques (which leads into

backtracking, discussed later). As a rule of thumb, if a brute force solution has to try $10^7$ or more possibilities, it may be too slow in a coding test environment.

**Example – Two-Sum (Brute Force):** As a simple illustration, consider the classic "two-sum" problem: given an array of integers, determine if any two distinct elements sum up to a target value. A brute force solution would check every possible pair of numbers in the array (a double loop over indices). This is $O(n^2)$ time, which is fine for moderate $n$ but scales poorly for large $n$. Below is the brute force two-sum check in C++14, Java 17, and Python 3.10:

**C++14**: brute-force two-sum (check all pairs) – O(n<sup>2</sup>)

```
bool hasPairSum(const vector<int>& arr, int target) {
  int n = arr.size();
  for (int i = 0; i < n; ++i) {
    for (int j = i + 1; j < n; ++j) {
      if (arr[i] + arr[j] == target) {
        return true;  // found a pair
      }
    }
  }
  return false;
}
```

**Java 17**: brute-force two-sum – O(n<sup>2</sup>)

```
static boolean hasPairSum(int[] arr, int target) {
  int n = arr.length;
  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
      if (arr[i] + arr[j] == target) {
        return true;  // found a pair
      }
    }
  }
  return false;
}
```

**Python 3.10**: brute-force two-sum – O(n<sup>2</sup>)

```
def has_pair_sum(arr, target):
    n = len(arr)
    for i in range(n):
        for j in range(i+1, n):
            if arr[i] + arr[j] == target:
                return True   # found a pair
    return False
```

In each version, we simply loop through every unique pair of indices and check the sum. The upside: the code is straightforward. The downside: its nested loop makes it quadratic time. For large arrays, an optimized approach (using a hash set to achieve average O(n) time) would be preferred in practice. But writing out the brute force solution is often a helpful first step in interviews – it demonstrates a correct approach, and then you can discuss how to optimize it.

In summary, **brute force** is about exhaustive search. It's invaluable to **understand** because many optimized algorithms build on brute force by pruning away impossible candidates or using smarter exploration. Whenever you approach a new problem, it's wise to think: "What is the brute force solution?" – this gives a correctness baseline and sometimes leads to insights for optimization. Just be mindful that pure brute force generally doesn't scale, so it should be used selectively (either for small inputs, as a stepping stone to better methods, or in combination with pruning techniques).

## Recursion (재귀)

**Recursion** (재귀) is an algorithmic technique where a function **calls itself** as a subroutine, directly or indirectly, to solve a problem. In programming terms, a recursive function is one that invokes itself within its own definition [11]. The recursive approach is based on the idea of solving a problem by **breaking it down into smaller subproblems of the same type**, solving those subproblems, and then combining the results. Every recursive algorithm has two key components: - **Base Case:** One or more stopping conditions that are simple to solve directly. When the base case is reached, the function returns a result without making further recursive calls. This prevents infinite recursion. - **Recursive Case:** The part of the function where it calls itself with a smaller or simpler input, moving toward the base case. Each recursive call solves a subproblem that is a fraction of the original problem.

A classic example is computing factorials. The factorial of n (denoted n!) is defined recursively as:
- Base case: 0! = 1 (and by convention 1! = 1)
- Recursive case: for n > 1, n! = n * (n–1)!

Using this definition, a recursive implementation for factorial follows directly: if n <= 1, return 1; otherwise return n * factorial(n-1). Let's see this in code with C++, Java, and Python:

**C++14**: recursive factorial

```
long long factorial(int n) {
  if (n <= 1)  // base case: 0! = 1, 1! = 1
    return 1;
  return 1LL * n * factorial(n - 1);  // recursive case
}
```

**Java 17**: recursive factorial

```
static long factorial(int n) {
  if (n <= 1) { // base case
    return 1;
  }
```

```
    return n * factorial(n - 1);  // recursive case
}
```

**Python 3.10**: recursive factorial

```
def factorial(n):
    if n <= 1:   # base case
        return 1
    return n * factorial(n - 1)   # recursive case
```

All three versions follow the same logic: they reduce the problem (factorial of n) to a smaller problem (factorial of n-1) until reaching n = 1 or 0, then unwind the recursion. For example, `factorial(5)` will call `factorial(4)`, which calls `factorial(3)`, and so on down to `factorial(1)` which returns 1. Then the results multiply back up: `1 -> 1*2 -> 2*3 -> 6*4 -> 24*5 -> 120`. This **divide-and-conquer by self-calling** is the essence of recursion.

**How Recursion Works (Stack Frames):** When a recursive function calls itself, the current function call is paused and a new instance of the function begins. The runtime uses a call stack to manage these function calls. Each call has its own set of parameters and local variables, independent of the others. Consider `factorial(3)` as an example: we call `factorial(3)` (which awaits the result of `factorial(2)`), which calls `factorial(2)` (awaits `factorial(1)`), which calls `factorial(1)` (hits base case and returns 1). Only once the base case returns do the prior calls resume and complete. The function calls form a stack – last call in, first out (LIFO) – exactly like stacking books and then removing them from the top [12] [13]. Each recursive call is pushed onto the stack until a base case is reached, then the calls start returning (popping off the stack) one by one. It's crucial that **every recursive chain eventually hits a base case**; otherwise, the calls would continue indefinitely, causing a stack overflow error (runaway recursion) [14] [15].

**Recursion vs. Iteration (Complexity):** A common question is: what's the performance penalty of using recursion instead of an iterative loop? In many cases, a simple recursion (without repeated overlapping subproblems) has the same Big-O time complexity as its iterative counterpart, but uses more memory due to the call stack. For example, summing numbers 1 to N recursively is O(N) time and O(N) space (for stack), whereas an iterative loop is O(N) time and O(1) space [16]. In our factorial example, both recursive and iterative implementations perform N multiplications (O(N) time); the recursive version, however, consumes O(N) stack space (each call waiting on the next) while the iterative uses constant space.

**When to Use Recursion:** Recursion is a natural fit for problems that can be defined in terms of similar subproblems. Many divide-and-conquer algorithms (like Quick Sort, Merge Sort) are recursive. Recursion is also idiomatic for tree and graph traversals (e.g. DFS), because these structures are recursive in nature (a tree node's children are themselves subtrees, etc.). Certain mathematical sequences are easily expressed with recursion (e.g. Fibonacci, though naive recursion for Fibonacci is inefficient). Recursion can lead to elegant solutions: for instance, a recursive algorithm for **Tower of Hanoi** or generating **combinations/permutations** is often clearer than the iterative approach.

However, one must be cautious with recursion on constraints: deep recursion can crash due to stack overflow if the recursion depth is too large (e.g. recursing 10,000 levels deep might exceed the stack). Tail-recursion optimization (TCO) can mitigate this in languages that support it (unfortunately C++ and Java do not optimize tail calls, but Python internally optimizes some tail calls). In performance-critical code,

an iterative solution might be preferable to avoid function call overhead. Always ensure your recursive function has a correct base case and that each recursive step makes progress toward it (to prevent infinite recursion).

**Tracing a Recursive Call – Example:** To solidify understanding, let's trace a simple recursive function – computing the sum of an array – in Python:

```
def recursive_sum(arr):
    # Base case: if array is empty, sum is 0
    if len(arr) == 0:
        return 0
    # Recursive case: sum = first element + sum of the rest
    return arr[0] + recursive_sum(arr[1:])

print(recursive_sum([3, 1, 4, 1]))   # Example call
```

When `recursive_sum([3,1,4,1])` is called, it will add 3 to `recursive_sum([1,4,1])`. That will add 1 to `recursive_sum([4,1])`. That adds 4 to `recursive_sum([1])`. That adds 1 to `recursive_sum([])`. Finally `recursive_sum([])` returns 0 (base case). The call stack unwinds and each addition completes: `0 + 1 -> 1; 1 + 4 -> 5; 5 + 1 -> 6; 6 + 3 -> 9`. The result 9 is returned. Each call had to wait for the result of the next call – a clear demonstration of the stack in action. Recursion can be mentally tricky at first, but tracing calls in this manner (or using debugger/print statements) helps greatly in understanding the flow.

## Divide and Conquer (분할 정복)

**Divide and Conquer** (분할 정복) is an algorithm design paradigm that involves breaking a problem into smaller subproblems, solving those subproblems (often recursively), and then **combining** their solutions to solve the original problem [17]. In essence: - **Divide:** Split the problem into two or more smaller **independent** subproblems of the same type. - **Conquer:** Solve each subproblem recursively (the recursion bottoms out when the subproblem becomes simple enough to solve directly, e.g. size 1 or 0). - **Combine:** Merge the results of the subproblems to form the final solution of the original problem.

This approach is powerful because solving subproblems that are half the size (for example) can drastically reduce complexity. Many efficient algorithms use divide-and-conquer with recursion, along with the mathematical tool of the **Master Theorem** to solve their recurrence relations and determine run-time complexity.

Famous examples of divide-and-conquer algorithms include **Merge Sort**, **Quick Sort**, **Binary Search**, **Exponentiation by Squaring**, **Strassen's Matrix Multiplication**, and more [18]. We will explore a few key examples: Merge Sort, fast exponentiation, and matrix exponentiation.

### Merge Sort (병합 정렬)

Merge sort is a classic divide-and-conquer sorting algorithm. It sorts an array by dividing it into two halves, sorting each half, and then merging the sorted halves. The algorithm can be summarized in two operations: 1. **Divide:** Split the array (or list) into two roughly equal halves (left and right). 2. **Conquer:** Recursively sort the left half, and recursively sort the right half. 3. **Combine:** Merge the two sorted halves into one sorted array.

The recursion bottoms out when the sub-array has 0 or 1 element, which is inherently sorted (base case). The key work is done in the **merge** step, which takes two sorted lists and **efficiently combines** them into a single sorted list.

Merge Sort recursively divides the array, then merges sorted subarrays. The diagram above shows how a list of unsorted numbers is split down to single elements and merged back in sorted order.

In the figure, the original array is divided repeatedly until we have single-element arrays (which are sorted by definition). Then the merge process begins: adjacent single elements are merged into sorted pairs, those pairs are merged into sorted quadruples, and so on until the whole array is sorted. This binary splitting yields a recursion tree of $\log_2(n)$ levels (for n elements), and at each level the merging of all subarrays costs O(n) total. As a result, **Merge Sort runs in O(n log n) time in the worst and average case**, and requires O(n) additional space for merging [19]. It's a very efficient sorting algorithm, and importantly, it's a **stable sort** (maintains the relative order of equal elements) and guarantees O(n log n) worst-case (unlike Quick Sort's worst-case $O(n^2)$, though Quick Sort is usually faster in practice on average).

**Time complexity analysis:** If T(n) is the time to sort n elements with Merge Sort, the algorithm divides into two subproblems of size n/2 and does O(n) work to merge results. This gives the recurrence:
$T(n) = T(n/2) + T(n/2) + O(n) = 2T(n/2) + O(n)$.
By the Master Theorem or expansion, T(n) resolves to **O(n log n)**. Space complexity is O(n) due to the temporary arrays used in merging (though there are in-place variants with more complex logic). Merge Sort's predictable O(n log n) performance makes it a favorite in situations where worst-case guarantees are important (e.g., sorting linked lists, or external sorting on disk).

**Merge Sort Implementation:** Let's implement Merge Sort in C++14, Java 17, and Python 3.10. We'll write a recursive function that sorts a portion of the array defined by indices (for C++/Java), and a helper function to merge two sorted subarrays.

**C++14**: Merge Sort (recursive)

```
void merge(vector<int>& arr, int l, int m, int r) {
  // Merge arr[l..m] and arr[m+1..r] into a sorted array
  int n1 = m - l + 1;
  int n2 = r - m;
  vector<int> left(n1), right(n2);
  for (int i = 0; i < n1; ++i) left[i] = arr[l + i];
  for (int j = 0; j < n2; ++j) right[j] = arr[m + 1 + j];
  int i = 0, j = 0, k = l;
  while (i < n1 && j < n2) {
    if (left[i] <= right[j])
      arr[k++] = left[i++];
    else
      arr[k++] = right[j++];
  }
  // Copy any remaining elements
  while (i < n1) arr[k++] = left[i++];
  while (j < n2) arr[k++] = right[j++];
}
```

```cpp
void mergeSort(vector<int>& arr, int l, int r) {
    if (l >= r) return;           // base case: 0 or 1 element
    int mid = l + (r - l) / 2;
    mergeSort(arr, l, mid);       // sort left half
    mergeSort(arr, mid + 1, r);   // sort right half
    merge(arr, l, mid, r);        // merge the two halves
}
```

In this C++ implementation, `mergeSort(arr, 0, n-1)` will sort the entire array. We split the range [l, r] in half, recursively sort each half, then call `merge` to combine them. The merge function uses temporary arrays for left and right parts, then merges them in O(n) time.

**Java 17**: Merge Sort (recursive)

```java
static void merge(int[] arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int[] left = new int[n1];
    int[] right = new int[n2];
    for (int i = 0; i < n1; i++) left[i] = arr[l + i];
    for (int j = 0; j < n2; j++) right[j] = arr[m + 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (left[i] <= right[j]) {
            arr[k++] = left[i++];
        } else {
            arr[k++] = right[j++];
        }
    }
    while (i < n1) arr[k++] = left[i++];
    while (j < n2) arr[k++] = right[j++];
}

static void mergeSort(int[] arr, int l, int r) {
    if (l >= r) return;           // base case
    int m = l + (r - l) / 2;
    mergeSort(arr, l, m);         // sort left
    mergeSort(arr, m + 1, r);     // sort right
    merge(arr, l, m, r);          // merge results
}
```

Java's version is similar, with `mergeSort(array, 0, n-1)` as the entry point. Again, we copy into temporary `left` and `right` arrays for merging. (In an interview, you could also merge in-place with careful index arithmetic, but using temp arrays is easier to get correct.)

**Python 3.10**: Merge Sort (recursive)

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr  # base case: already sorted
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    # merge left and right
    merged = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i]); i += 1
        else:
            merged.append(right[j]); j += 1
    # append any remaining elements
    merged.extend(left[i:]); merged.extend(right[j:])
    return merged
```

In Python, we return a new sorted list rather than sorting in place (for clarity). We find the midpoint, recursively sort the slices `arr[:mid]` and `arr[mid:]`, then merge them. Python's list slicing and extending make the merge step concise. The overall complexity remains O(n log n).

**Walkthrough:** Suppose we sort the array `[38, 27, 43, 3, 9, 82, 10]` with merge sort: - We split into `[38, 27, 43, 3]` and `[9, 82, 10]`. - Recursively sort `[38, 27, 43, 3]` -> split into `[38, 27]` and `[43, 3]` ... - Eventually, we reach single-element arrays: `[38]`, `[27]` which then merge into `[27, 38]`; `[43]`, `[3]` merge into `[3, 43]`. - Next, `[27, 38]` and `[3, 43]` merge into `[3, 27, 38, 43]`. - Meanwhile, sort `[9, 82, 10]` -> split to `[9, 82]` and `[10]` -> `[9, 82]` splits to `[9]` and `[82]` which merge into `[9, 82]`. Then merge `[9, 82]` with `[10]` to get `[9, 10, 82]`. - Finally, merge the two halves `[3, 27, 38, 43]` and `[9, 10, 82]` into `[3, 9, 10, 27, 38, 43, 82]`.

The algorithm ensures that at each merge step, both sublists are already sorted, so the merge can be done in linear time by a linear scan through both lists. This sorted merging is the reason for the O(n log n) complexity. Merge sort's deterministic performance and simplicity in implementation (especially recursively) make it a great example of divide-and-conquer in action.

## Fast Exponentiation (Exponentiation by Squaring)

A classic application of divide-and-conquer is **fast exponentiation**, also known as **binary exponentiation** or exponentiation by squaring. The goal is to compute $a^n$ (a number raised to the power n) much faster than the naive approach of multiplying `a` by itself n times. The trick is to use the binary representation of the exponent to reduce the number of multiplications to O(log n) [20].

The insight is based on these mathematical identities: - $a^{2k} = (a^k)^2$. (To compute a power with an even exponent, you can square the result of half the exponent.) - $a^{2k+1} = a \times (a^k)^2$. (If the exponent is odd, factor out one `a` and then square the rest.)

Using these, we can recursively define the power: - Base case: $a^0 = 1$ (anything to the power 0 is 1). - If n is even: $a^n = (a^{n/2})^2$. (Compute half power and square it.) - If n is odd: $a^n = a \times (a^{(n-1)/2})^2$. (Reduce to an even exponent case by factoring out one `a`.)

This recursive approach will roughly halve the exponent at each step, achieving **O(log n)** multiplications instead of O(n). For example, to compute $a^{13}$, we note 13 in binary is 1101 (8 + 4 + 0 + 1):
$a^{13} = a^{8} \cdot a^{4} \cdot a^{1}$.
And $a^{8} = ((a^4)^2)$, $a^4 = ((a^2)^2)$, etc. In total, instead of 12 multiplications (for 13 multiplies), we can do it in about $\lceil \log_2 13 \rceil = 4$ squaring steps (plus a few extra multiplies for odd factors). The saving becomes dramatic for large n.

Let's implement fast exponentiation in our three languages:

**C++14**: binary exponentiation (recursive)

```
long long fastPower(long long x, long long n) {
  if (n == 0)
    return 1;  // base case
  long long half = fastPower(x, n / 2);
  if (n % 2 == 0) {
    return half * half;
  } else {
    return half * half * x;
  }
}
```

**Java 17**: binary exponentiation

```
static long fastPower(long x, long n) {
  if (n == 0)
    return 1;  // base case
  long half = fastPower(x, n / 2);
  if (n % 2 == 0) {
    return half * half;
  } else {
    return half * half * x;
  }
}
```

**Python 3.10**: binary exponentiation

```
def fast_power(x, n):
    if n == 0:
        return 1   # base case
    half = fast_power(x, n // 2)
    if n % 2 == 0:
        return half * half
```

```
    else:
        return half * half * x
```

Each of these functions uses the divide-and-conquer approach: computing `x^(n/2)` and then squaring it, which is the “conquer” step, and an extra multiply by `x` if needed for odd n. The recursion depth is O(log n) since we halve n each time. Thus, the number of multiplications is on the order of $\log_2(n)$. For instance, to compute $2^{32}$, the naive method does 31 multiplications, whereas binary exponentiation does only 6 multiplications (since 32 in binary is 100000, which leads to a series of squarings). In general, **binary exponentiation runs in O(log n) time** [21], a significant improvement over O(n). This technique is extremely useful in many domains – especially in modular arithmetic for cryptography (large exponent mod M) or in algorithms that need to repeatedly square values (like matrix exponentiation below).

To illustrate quickly: say we want to compute $3^{13}$: - n=13 (odd): compute $3^{(13-1)/2} = 3^6$, then square it and multiply by 3. - To get $3^6$: - n=6 (even): compute $3^3$, then square it. - To get $3^3$: - n=3 (odd): compute $3^{(3-1)/2} = 3^1$, square it, multiply by 3. - $3^1$ is base case = 3. - Square of $3^1$ is 9, times 3 gives 27, so $3^3 = 27$. - Now square $3^3$: $27^2 = 729$, so $3^6 = 729$. - Square $3^6$: $729^2 = 531441$. Multiply by 3: $531441 * 3 = 1,594,323$ which is $3^{13}`. We did a handful of multiplications instead of 12.

Most languages have built-in power functions (e.g. `pow` in Python or `Math.pow` in Java) which likely use exponentiation by squaring internally. But it’s important to know how to implement it yourself. In interviews, binary exponentiation might come up when discussing optimizing a power calculation or as part of another algorithm (like binary search, it also follows a halving strategy).

### Matrix Exponentiation (행렬 거듭제곱)

Divide-and-conquer exponentiation isn’t limited to numbers; it works for any associative operation – notably, **matrix multiplication**. **Matrix exponentiation** applies the same squaring technique to compute $M^n$, where M is a matrix [22]. This is particularly useful for solving linear recurrences quickly (like Fibonacci numbers, dynamic programming state transitions, etc.). The idea is: to compute the nth power of a matrix, we can square the matrix $\log_2(n)$ times instead of multiplying it n times.

For example, Fibonacci numbers can be computed via matrix exponentiation. The Fibonacci recurrence can be encoded in a $2\times2$ matrix:

$$M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix},$$

such that:

$$M^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix},$$

where $F_k$ is the k-th Fibonacci number (with $F_0 = 0, F_1 = 1$). Thus, if we can compute $M^n$ quickly, we can get $F_{n+1}$ from the (0,0) element (or $F_n$ from (0,1)). Using divide-and-conquer, we can compute $M^n$ in O(log n) matrix multiplications rather than n multiplications. Each matrix multiplication for 2x2 matrices is a constant number of scalar multiplications (in general, for d×d matrices it’s O(d^3) naïvely). The principle is identical to scalar fast power: - If n is even, $M^n = (M^{n/2}) \times (M^{n/2})$. - If n is odd, $M^n = M \times (M^{(n-1)/2}) \times (M^{(n-1)/2})$.

Let's implement matrix exponentiation for 2x2 matrices as an example, focusing on the Fibonacci application. We will write a function to multiply two 2x2 matrices and a function to exponentiate a 2x2 matrix by an integer exponent using binary exponentiation logic:

**Python 3.10**: Matrix exponentiation (2x2 matrices)

```
def matrix_multiply(A, B):
    # Multiply two 2x2 matrices A and B
    return [
        [ A[0][0]*B[0][0] + A[0][1]*B[1][0],
          A[0][0]*B[0][1] + A[0][1]*B[1][1] ],
        [ A[1][0]*B[0][0] + A[1][1]*B[1][0],
          A[1][0]*B[0][1] + A[1][1]*B[1][1] ]
    ]

def matrix_power(M, n):
    # Compute M^n for 2x2 matrix M using binary exponentiation
    result = [[1, 0], [0, 1]]      # 2x2 Identity matrix
    while n > 0:
        if n % 2 == 1:
            result = matrix_multiply(result, M)
        M = matrix_multiply(M, M)
        n //= 2
    return result

# Example: Compute 10th Fibonacci number using matrix exponentiation
fib_matrix = [[1, 1], [1, 0]]
M_power = matrix_power(fib_matrix, 9)   # M^9 (since F_10 will be at index (0,0))
print("F10 =", M_power[0][0])   # Should print 55, since F10 (if F0=0) is 55
```

In this code, `matrix_power` uses an iterative binary exponentiation (it could be done recursively as well). We repeatedly square the matrix `M` (modifying it each loop) and multiply it into `result` whenever we encounter a 1 bit in the exponent (when n is odd). This is the standard bit-halving method to exponentiate in log n steps. The identity matrix acts like 1 in matrix multiplication.

For the Fibonacci example, we set `fib_matrix` as the transformation matrix. If we want $F_{10}$ (with the convention $F_0=0, F_1=1$), we compute $M^9$ and take element (0,0). Indeed, `M_power[0][0]` will be 55, the 10th Fibonacci number.

**Complexity:** Matrix exponentiation by squaring yields O(log n) matrix multiplications. Each matrix multiplication for a 2x2 is O(1) (constant number of operations), so Fibonacci can be obtained in O(log n) time this way [22]. For a general d×d matrix, each multiplication is O(d^3) (or better with advanced algorithms), so it would be O(d^3 log n). This is still a huge win for large n compared to O(n). In contexts like competitive programming, matrix exponentiation is commonly used to solve recurrences (e.g. Fibonacci, tribonacci, etc.) under modulo arithmetic quickly, or to advance the state of a DP transition matrix to the nth step in log n time.

It's important to note that matrix exponentiation is only applicable when the problem can be modeled by linear transformations. But when it is applicable, it's a powerful tool. We saw how it solves Fibonacci in

log n. Another example: computing $F_{n}$ mod m for a huge n (say n = 10^9) directly with DP or recursion is infeasible, but with matrix exponentiation it becomes trivial. Many interview problems involving recurrences can be cracked with this technique if n is large and a direct DP would be too slow.

### Summary of Divide & Conquer

Divide-and-conquer algorithms often have a recursive structure and their efficiency comes from cutting down the input size rapidly. The paradigmatic runtime form is T(n) = a T(n/b) + f(n), which via the Master Theorem gives various complexities. In merge sort and exponentiation, we saw `a=2, b=2, f(n)=O(n)` leading to O(n log n) and `a=1, b=2, f(n)=O(1)` leading to O(log n). Not all divide-and-conquer is about halving (for example, Quick Sort partitions arrays but expected O(n log n) time as well). The key takeaway is: think about how you can break a problem down. If you can reduce a problem significantly (say to half size) and do linear or sublinear extra work, you usually get a big speedup over a naive approach. We will next explore backtracking, which in some sense is brute force enhanced with divide-and-conquer (exploring one possibility at a time and backtracking) and pruning.

## Backtracking (백트래킹)

**Backtracking** (백트래킹) is an algorithmic technique for **systematically searching** for a solution among all possibilities, but with the ability to **backtrack** when a path cannot possibly lead to a valid solution. It builds solutions one step at a time and abandons ("backtracks") a path as soon as it determines that this path cannot lead to a valid solution. In essence, backtracking is a depth-first search (DFS) on a state-space tree of choices, where we **prune** branches that are infeasible, rather than blindly exploring all possibilities like brute force [23].

In more concrete terms, backtracking incrementally **constructs a solution** (e.g., building an array, string, or configuration one element or decision at a time). At each step, it checks if the partial solution is still valid or promising. If it is, the algorithm recurses to add the next component. If it isn't (i.e., it violates a constraint or cannot possibly lead to a complete valid solution), the algorithm **undoes** the last step (backtracks) and tries a different option in that step. This "try – if fail – backtrack – try next" process continues until solutions are found or all options are exhausted [24].

A classic description: "Backtracking is like trying to find your way through a maze. You move forward until you hit a dead end, then you step back (backtrack) and choose a different path." This method is commonly used for **combinatorial search** problems: finding all permutations, combinations, solving puzzles like N-Queens, Sudoku, crosswords, maze-solving, graph coloring, Hamiltonian paths, etc. It's essentially optimized brute force: it still may explore many possibilities, but it **cuts off** large sets of them by recognizing failure early.

**How Backtracking Works:** We can outline a generic backtracking algorithm: 1. **Choose**: Make a choice from a set of options (e.g., pick a number for the next position, choose a path direction, etc.). 2. **Constraint Check**: If the choice violates a constraint (i.e., it makes the partial solution invalid), **prune** this branch – backtrack immediately without exploring further down this path. 3. **Recurse**: If the choice is valid, apply it and move on to the next step (recursive call to make the next choice). 4. **Backtrack**: After exploring the choice (and perhaps finding that ultimately it didn't lead to a solution or after recording a solution), undo the choice (restore state) and try the next alternative option at this step. 5. Continue until all options have been tried at every step.

This approach will explore the state-space tree of possible solution configurations in a depth-first manner [25]. The moment a partial configuration is determined to be invalid, the algorithm backtracks, which can

save enormous time compared to brute force (which would continue exploring that dead path fully). In a sense, backtracking is brute force with early pruning.

Backtracking search tree for the 4-Queens problem. Each level represents placing a queen in a row. Invalid placements (where queens can attack each other) are pruned (red X), and the search backtracks to try alternate columns. A valid solution is found (green check).

In the figure above, we see the state-space tree for the 4-Queens puzzle (placing 4 queens on a 4×4 board so none attack each other). The algorithm places a queen in row 1 in some column and then recursively attempts to place a queen in row 2, and so on. Whenever a partial placement is found to be unsafe (queens can attack), that branch is abandoned (shown as red X dead ends). The backtracking algorithm then tries the next column for the queen in the previous row. Eventually, it finds all solutions (one such solution path is highlighted with a green checkmark). Without backtracking, one would have to check all $4^4 = 256$ possible placements; with backtracking, the algorithm eliminates the majority of those early by noticing conflicts. This is the power of pruning.

**Applications:** Backtracking is ideal for **combinatorial generation** problems (generate all permutations, all combinations, subsets, etc.), constraint satisfaction problems (scheduling, puzzle solving like Sudoku, N-Queens, crosswords), and search problems where the solution is a sequence or configuration that must satisfy certain conditions. Many NP-complete problems (like SAT, graph coloring, Hamiltonian cycle) have backtracking solutions – which in worst-case may still be exponential, but backtracking can handle moderate input sizes and is often the basis for more advanced approaches like branch-and-bound and heuristic search.

**Complexity:** The worst-case time complexity of backtracking is often still **exponential** (since it may have to explore all possibilities in the worst scenario). For instance, the N-Queens problem has $O(N!)$ possibilities in brute force, and backtracking still worst-case explores on the order of $N!$ in the absence of solutions [26] (with some pruning overhead). However, the **average** complexity can be much better with effective pruning. In many practical scenarios, backtracking prunes a huge portion of the search space, making it feasible to solve quite large problems that brute force couldn't. The efficiency of backtracking hinges on the constraints: the tighter the constraints (more pruning), the less of the search tree is explored.

**Backtracking vs. Brute Force:** It's worth noting the distinction: brute force would enumerate all possibilities without exception; backtracking enumerates possibilities too but **skips many by using problem-specific logic** to stop exploring fruitless paths [23]. If no pruning is possible (no constraints to check until a full solution), backtracking is essentially brute force. But when partial validation is possible (as in puzzles where even a partial assignment can violate a rule), backtracking can drastically cut down the work.

**Example – Generating Permutations:** As an illustrative example, let's use backtracking to generate all permutations of a list of numbers. This is a typical backtracking problem because we have to build permutations one element at a time and we can backtrack to swap out choices. We'll demonstrate the approach in C++, Java, and Python.

**C++14**: Backtracking to generate all permutations of a list

```
#include <iostream>
#include <vector>
using namespace std;
```

```cpp
void backtrackPermute(vector<int>& arr, int l, int r) {
  if (l == r) {
    // We found a full permutation, print or store it
    for(int x : arr) cout << x << " ";
    cout << endl;
  } else {
    for (int i = l; i <= r; ++i) {
      swap(arr[l], arr[i]);          // choose: place arr[i] at index l
      backtrackPermute(arr, l+1, r);    // explore: permute remaining
      swap(arr[l], arr[i]);          // backtrack: undo swap
    }
  }
}

int main() {
  vector<int> nums = {1, 2, 3};
  backtrackPermute(nums, 0, nums.size() - 1);
}
```

In this C++ example, `backtrackPermute` generates all permutations of `arr[l..r]` by swapping each possible element into position `l` (the current index to fill), then recursing to permute the rest. When `l == r`, we have a complete permutation (which we output or could collect). The key is the `swap` and then `swap` back – this is the backtracking step that undoes the choice so that other choices can be tried at that position. The time complexity is O(n * n!), because there are n! permutations and printing each permutation of length n takes O(n). But the algorithm efficiently navigates the permutation space without repetition.

**Java 17**: Permutation generation via backtracking

```java
public class  PermuteExample {
  static void backtrackPermute(int[] arr, int l, int r) {
    if (l == r) {
      // output the permutation
      System.out.println(Arrays.toString(arr));
    } else {
      for (int i = l; i <= r; i++) {
        swap(arr, l, i);          // choose element i for position l
        backtrackPermute(arr, l+1, r);    // recurse for next positions
        swap(arr, l, i);          // backtrack (undo choice)
      }
    }
  }
  static void swap(int[] arr, int i, int j) {
    int tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;
  }
  public static void main(String[] args) {
    int[] nums = {1, 2, 3};
    backtrackPermute(nums, 0, nums.length - 1);
```

```
    }
}
```

This Java version does the same: swap the current element with each candidate and backtrack. The use of a helper `swap` makes it straightforward.

**Python 3.10**: Permutations via backtracking

```python
def backtrack_permute(arr, l, r):
    if l == r:
        print(arr)   # or collect a copy of arr as a result
    else:
        for i in range(l, r+1):
            arr[l], arr[i] = arr[i], arr[l]          # choose
            backtrack_permute(arr, l+1, r)           # explore
            arr[l], arr[i] = arr[i], arr[l]          # backtrack (undo)

nums = [1, 2, 3]
backtrack_permute(nums, 0, len(nums)-1)
```

Python's tuple assignment makes swapping easy. We again see the pattern: for each choice at position `l`, place that choice, recurse, then undo.

This example demonstrates the general backtracking pattern: **choose -> explore -> un-choose**. Even though generating all permutations has a high complexity (O(n!)), backtracking is the natural way to do it, and for reasonably small n (say n=6 or 7) it's fine. In interview settings, backtracking solutions are often acceptable when n is small (like n<=10 or 15 for permutations or combinations problems) or when there are additional constraints that prune the search.

**Another Example – N-Queens (sketch):** To see backtracking with constraint checking, think of the N-Queens problem. We place queens row by row. For each row, we try each column and check if placing a queen there would conflict with any previously placed queens (checking columns and diagonals). If conflict, skip (prune) that column choice. If no conflict, place the queen and move to next row. If we successfully place queens in all rows, we found a solution; otherwise, if a row has no valid positions, backtrack to previous row and move the queen there to the next position. This is backtracking in action: partial solutions (queens placed so far) are checked for validity at each step [27] [28], and invalid paths are pruned. The state-space tree for 4-queens we showed earlier is a perfect depiction: backtracking dramatically reduces the possibilities we actually explore, from 256 naive placements to only 16 valid partial placements examined in that example tree.

**When to Use Backtracking:** Use backtracking when you need to search through a space of **combinatorial possibilities** and you can systematically build candidate solutions and test partial progress. If you can detect failure early (via constraints), backtracking will save time. It's especially useful in puzzles and constraint satisfaction (Sudoku solver is a backtracking solution, where you fill cells and backtrack upon conflict), as well as generating combinatorial objects (subsets, permutations, combinations, partitioning problems, etc.). Keep in mind backtracking can still be slow if the search space is huge and constraints are weak (leading to little pruning), so sometimes additional heuristics or

optimization (like ordering choices, forward-checking in constraint solving, etc.) are employed to enhance backtracking.

**Conclusion & Takeaways:** This week's topics – brute force, recursion, divide-and-conquer, and backtracking – form a toolkit of fundamental strategies for algorithm design: - Brute force gives a correctness baseline (try everything) but often needs optimization. - Recursion allows elegant implementation of divide-and-conquer and backtracking algorithms by breaking problems into subproblems. - Divide and Conquer optimizes brute force by splitting problems (we saw how sorting and powering were improved from $O(n^2)$/$O(n)$ to $O(n \log n)$/$O(\log n)$ with this strategy). - Backtracking smartly navigates huge search spaces by pruning and is indispensable for combinatorial search problems.

Each approach comes with typical use cases and trade-offs in terms of time complexity and implementation complexity. As an algorithmic problem-solver, you should be comfortable recognizing which paradigm fits a given problem and how to implement it efficiently. Practice is key: try writing recursive solutions, converting some to iterative if needed; try implementing a backtracking solution for a small puzzle; or analyze the recurrence of a divide-and-conquer algorithm. By mastering these techniques, you will be well-prepared for many coding interview challenges and algorithmic tests, able to start from a brute force solution and refine it using recursion, divide-and-conquer, or backtracking as appropriate. Happy coding and see you next session! [4] [24]

---


[1] [2] [3] study-note-week02.docx
file://file-BpdjM5YCsthCJEDjfKBfTW

[4] [6] [7] [8] [10] Brute Force Approach and its pros and cons - GeeksforGeeks
https://www.geeksforgeeks.org/dsa/brute-force-approach-and-its-pros-and-cons/

[5] [9] [23] Brute-force search - Wikipedia
https://en.wikipedia.org/wiki/Brute-force_search

[11] [14] [15] [16] Introduction to Recursion - GeeksforGeeks
https://www.geeksforgeeks.org/introduction-to-recursion-2/

[12] [13] How Recursion Works — Explained with Flowcharts and a Video
https://www.freecodecamp.org/news/how-recursion-works-explained-with-flowcharts-and-a-video-de61f40cb7f9/

[17] [18] Divide and Conquer Algorithm - GeeksforGeeks
https://www.geeksforgeeks.org/dsa/divide-and-conquer/

[19] Merge sort - Wikipedia
https://en.wikipedia.org/wiki/Merge_sort

[20] [21] Binary Exponentiation - Algorithms for Competitive Programming
https://cp-algorithms.com/algebra/binary-exp.html

[22] Matrix Exponentiation - GeeksforGeeks
https://www.geeksforgeeks.org/dsa/matrix-exponentiation/

[24] [25] [27] [28] Backtracking Algorithm - GeeksforGeeks
https://www.geeksforgeeks.org/dsa/backtracking-algorithms/

[26] N Queen Problem - GeeksforGeeks
https://www.geeksforgeeks.org/dsa/n-queen-problem-backtracking-3/