ChatGPT

알고리즘 스터디: 3주차 완전탐색, 재귀, 분할정복, 백트래킹

새로운 주제로 들어가기 전에, 2주차에 배운 내용을 간략히 되짚어 보겠습니다. 지난 시간에는 배열, 연결 리스트, 스택, 큐, 집합, 맵 등과 같은 기본적인 자료구조와 문자열 조작 기법에 대해 집중적으로 다루었습니다. 예를 들어 배열과 연결 리스트를 비교했는데, 배열은 임의 접근(random access)을 $O(1)$에 제공하는 대신 삽입과 삭제 비용이 크고, 연결 리스트는 삽입/삭제는 빠르지만 순차 접근만 가능하다는 점을 언급했습니다【1†】. 또한, 흔히 쓰이는 문자열 알고리즘도 연습했습니다. 예를 들어, 문자열이 앞뒤가 같은지 확인하는 팰린드롬 검사【2†】나, 두 문자열이 동일한 문자로 이루어졌는지(순서가 다른 애너그램인지) 확인하는 알고리즘【3†】 등이 있습니다. 이러한 자료 구조와 조작 기법에 대한 기반이 알고리즘 문제를 해결하는 데 기초가 됩니다. 이를 바탕으로, 3주차에서는 완전탐색, 재귀, 분할정복, 백트래킹 등 핵심 알고리즘 설계 패러다임으로 주제를 전환합니다. 이러한 기법들을 익히면 코딩 면접 문제를 효과적으로 해결할 수 있는 여러 가지 접근 방식을 사용할 수 있게 됩니다.

완전탐색 (Brute Force)

완전탐색(혹은 브루트 포스 기법)은 문제 해결을 위한 가장 기본적인 방법입니다. 가능한 모든 경우를 차례대로 시도하면서 해결책을 찾는 방식입니다. 완전탐색 알고리즘은 문제의 요구 사항을 만족하는 해답인지 확인하기 위해 모든 가능한 후보 해답을 체계적으로 탐색합니다【4†】. 다시 말해, 어떤 지름길이나 휴리스틱을 사용하지 않고 가능한 모든 결과를 전부 생성하여 테스트하는 방식입니다. 이 방법은 가능한 모든 경우를 빠짐없이 살펴보므로, 해답이 있는 경우 반드시 찾고, 해답이 없음을 정확히 판단할 수 있다는 것을 보장합니다. Ken Thompson이 '의심스러울 때는 완전탐색을 사용하라'고 말했듯이, 완전탐색은 기본적인 기준점이나 최후의 수단으로 사용할 수 있습니다. 하지만 해답을 확실히 찾을 수 있다는 보장에는 대가가 매우 큽니다. 입력 크기가 커지면 가능한 후보의 수가 기하급수적으로 늘어나기 때문입니다(이를 조합 폭발(combinatorial explosion)이라고 합니다). 예를 들어 8-퀸 문제를 완전탐색으로 풀면 체스판의 64칸 중 퀸 8개를 놓는 모든 경우($64 \choose 8$)를 시도해야 합니다【5†】. 마찬가지로 외판원 문제를 완전탐색으로 풀 때는 N개의 도시 순열 N!을 모두 확인해야 합니다. 이런 방법들은 입력 크기가 커지면 비현실적이 됩니다. 시간 복잡도가 지수(exp.) 또는 계승(factorial)이 되는 경우가 많기 때문입니다【6†】. 요약하면, 완전탐색은 효율성을 포기하는 대신 단순성과 확실성을 얻는 방식입니다.

완전탐색에도 몇 가지 활용 사례와 장점이 있습니다: - 구현이 쉽고 이해하기 쉬워 – 초기 해답을 구하거나 최적화된 알고리즘과 비교할 기준으로 유용합니다【7†】. - 문제 크기가 충분히 작아 합리적인 시간 내에 모든 경우를 탐색할 수 있는 경우 잘 작동합니다【8†】. 그런 때 단순함이 오히려 장점이 될 수 있습니다. - 정확성이 가장 중요한 애플리케이션의 경우, 휴리스틱이나 복잡한 최적화가 오히려 위험을 초래할 수 있습니다. 완전탐색은 언제나 결국 정답을 찾습니다【9†】. - 인터뷰 환경에서는 기본 풀이로 사용할 수 있습니다. 우선 완전탐색으로 문제를 해결하면 문제 이해도를 보여줄 수 있고, 그 다음에 이를 최적화하는 방법에 대해 논의할 수 있습니다.

시간 복잡도 분석: 완전탐색 알고리즘은 일반적으로 모든 알고리즘 중에서도 최악의 시간 복잡도를 가집니다. 간단한 선형 탐색의 경우에는 $O(n)$이 될 수 있지만, 대부분의 완전탐색 해법은 고차 다항식 또는 지수적 복잡도를 가집니다. 예를 들어, n개 원소 집합의 모든 부분집합을 완전탐색하면 $O(2^n)$이고, 모든 순열을 확인하는 것은 $O(n!)$입니다【10†】. 이 때문에 완전탐색은 일반적으로 입력 크기가 작거나, 가지치기(pruning) 기법과 결합된 경우에만 현실적입니다 (가지치기는 뒤에서 백트래킹에서 다룹니다). 경험적으로, 완전탐색에서 10^7개 이상의 경우를 시도해야 하는 경우 코딩 테스트에서는 너무 느릴 수 있습니다.

예제 – Two-Sum (완전탐색): 간단한 예로 고전적인 "두 합(Two-Sum)" 문제를 생각해보겠습니다: 정수 배열이 주어졌을 때, 두 개의 서로 다른 원소를 골라 그 합이 목표값이 되는 경우가 있는지 구하는 문제입니다. 완전탐색이라면 배열의 모든 가능한 두 원소 쌍을 확인할 것입니다(이중 반복문 사용). 이 방법은 $O(n^2)$ 시간 복잡도를 가지며, 적당한

n에는 괜찮지만 n이 커지면 성능이 급격히 나빠집니다. 다음은 C++14, Java 17, Python 3.10으로 구현한 완전 탐색 방식의 Two-Sum 코드입니다:

C++14: brute-force two-sum (check all pairs) – $O(n^2)$

```
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 – $O(n^2)$

```
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 – $O(n^2)$

```
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
```

각 언어 버전 모두 모든 두 인덱스 조합을 순회하며 합을 검사합니다. 장점은 코드가 간단하다는 점입니다. 단점은 이중 반복문으로 인해 시간 복잡도가 2차원이 된다는 것입니다. 배열이 매우 큰 경우, 평균적으로 $O(n)$ 시간에 해결할 수 있는 해시 집합을 사용하는 최적화된 방법이 실제로는 더 선호됩니다. 그러나 인터뷰 상황에서는 완전탐색 풀이를 먼저 작성하는 것이 종종 좋은 첫 번째 단계가 됩니다. 올바른 풀이 방법을 보인 후에 이를 어떻게 최적화할 수 있을지 논의할 수 있기 때문입니다.

정리하자면, 완전탐색은 모든 경우를 탐색하는 방식입니다. 최적화된 많은 알고리즘이 불가능한 후보를 가지치기하거나 더 똑똑한 탐색 방법을 사용해 완전탐색을 바탕으로 만들어지기 때문에, 완전탐색을 이해하는 것은 매우 중요합니다. 새로운 문제를 만났을 때 항상 “완전탐색 해답은 무엇인가?”를 생각해보는 것이 현명합니다. 이는 정답을 찾을 때 기준점이 되어줄 뿐만 아니라, 최적화를 위한 아이디어를 얻는 데 도움이 되기도 합니다. 단, 순수한 완전탐색은 일반적으로 확장성이 떨어지므로, 입력 크기가 작을 때나 더 나은 방법으로 개선하는 과정에서, 또는 가지치기 기법과 결합하여 선택적으로 사용하는 것이 좋습니다.

재귀 (Recursion)

재귀(Recursion)는 문제를 해결하기 위해 함수가 직접적 또는 간접적으로 자기 자신을 하위 루틴으로 호출하는 알고리즘 기법입니다. 프로그래밍 관점에서는, 재귀 함수란 그 정의 내에서 자기 자신을 호출하는 함수를 말합니다【11†】. 재귀적 접근은 문제를 동일한 유형의 더 작은 하위 문제로 나누어 해결하고, 그 결과를 결합하여 원래 문제를 해결한다는 아이디어를 기반으로 합니다. 모든 재귀 알고리즘은 두 가지 핵심 요소를 갖습니다: - **기저 사례(Base Case):** 직접 풀 수 있는 하나 이상의 종료 조건입니다. 기저 사례에 도달하면 함수는 추가적인 재귀 호출 없이 결과를 반환합니다. 이는 무한 재귀를 방지합니다. - **재귀 사례(Recursive Case):** 함수가 더 작거나 단순한 입력으로 자신을 호출하는 부분으로, 기저 사례를 향해 진행합니다. 각 재귀 호출은 원래 문제의 일부인 작은 하위 문제를 해결합니다.

고전적인 예로 팩토리얼 계산을 들 수 있습니다. n의 팩토리얼(기호 $n!$)은 재귀적으로 다음과 같이 정의됩니다: - 기저 사례: $0! = 1$ (관례상 $1! = 1$도 포함됩니다) - 재귀 사례: $n! = n \times (n-1)!$ (n > 1일 때)

이 정의를 사용하면 팩토리얼의 재귀 구현이 바로 나옵니다: $n \le 1$이면 1을 반환하고, 그렇지 않으면 $n \times factorial(n-1)$을 반환합니다. 이제 C++, Java, Python 코드로 이를 살펴보겠습니다.

C++14: 재귀를 이용한 팩토리얼

```cpp
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: 재귀를 이용한 팩토리얼

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

Python 3.10: 재귀를 이용한 팩토리얼

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

세 가지 버전 모두 같은 논리를 따릅니다: 문제(자연수 n의 팩토리얼)를 더 작은 문제(자연수 $n-1$의 팩토리얼)로 줄여나가다가 $n = 1$ 또는 $n = 0$에 도달하면 재귀 호출이 해제되어 결과가 반환됩니다. 예를 들어 `factorial(5)`를 호출하면 `factorial(4)`를 호출하고, `factorial(4)`는 `factorial(3)`을 호출하는 식으로 내려갑니다. 결국 `factorial(1)`이 1을 반환하면 호출 스택이 풀리면서 결과를 차례로 곱해 나갑니다: 1 → 1×2 → 2×3 → 6×4 → 24×5 → 120. 자기 자신을 호출하며 문제를 나누는 방식이 바로 재귀의 본질입니다.

재귀 작동 방식 (스택 프레임): 재귀 함수가 자신을 호출하면 현재 호출이 일시 중지되고 함수의 새로운 인스턴스가 시작됩니다. 런타임은 이 함수 호출들을 관리하기 위해 호출 스택을 사용합니다. 각 호출에는 자신의 매개변수와 지역 변수가 있으며, 서로 간섭하지 않습니다. 예를 들어 `factorial(3)`을 살펴보겠습니다: `factorial(3)`을 호출하면 `factorial(2)`의 결과를 기다립니다. `factorial(2)`는 `factorial(1)`을 호출합니다. `factorial(1)`이 기저 사례에 도달하여 1을 반환하면, 그제서야 이전 호출들이 차례로 다시 진행되고 결과를 계산하게 됩니다. 함수 호출들은 LIFO(Last In, First Out) 방식으로 쌓였다가 제거됩니다【12†】【13†】. 각 재귀 호출은 기저 사례에 도달할 때까지 스택에 쌓이고, 기저 사례에 도달하면 호출들이 하나씩 반환되며 스택에서 제거됩니다. 모든 재귀 체인은 반드시 기저 사례에 도달해야 합니다. 그렇지 않으면 호출이 무한히 반복되어 스택 오버플로 오류(재귀의 무한 루프)【14†】【15†】가 발생할 수 있습니다.

재귀 vs 반복 (복잡도): 자주 묻는 질문 중 하나는 "재귀를 사용하는 것과 반복문을 사용하는 것의 성능 차이(패널티)가 무엇인가?" 입니다. 많은 경우, 단순 재귀는 반복문 버전과 동일한 빅오 시간 복잡도를 가지면서도, 호출 스택 때문에 더 많은 메모리를 사용합니다. 예를 들어 1부터 N까지의 합을 재귀로 계산하면 시간 복잡도는 $O(N)$이고 공간 복잡도는 $O(N)$입니다(스택 공간 때문). 반복문을 사용하면 시간 복잡도는 $O(N)$이지만 공간 복잡도는 $O(1)$입니다【16†】. 팩토리얼 예제에서도 재귀와 반복 구현 모두 N번의 곱셈을 수행하므로 시간 복잡도는 $O(N)$입니다. 그러나 재귀 버전은 $O(N)$ 스택 공간(각 호출이 다음 호출을 기다림)을 필요로 하는 반면, 반복 버전은 상수 공간만을 사용합니다.

재귀 사용 시기: 재귀는 유사한 하위 문제들로 정의할 수 있는 문제에 적합합니다. 많은 분할정복 알고리즘(예: 퀵 정렬, 병합 정렬)은 재귀적입니다. 재귀는 트리와 그래프 순회(예: 깊이 우선 탐색 DFS)에 자연스럽게 사용되기도 합니다. 트리 구조 자체가 재귀적인 특성(노드의 자식들이 또 다른 서브트리) 때문입니다. 특정 수학적 수열(예: 피보나치 수열)은 재귀로 쉽게 표현할 수 있습니다(단, 피보나치 수열을 단순 재귀로 구현하면 비효율적이라는 단점이 있습니다). 재귀는 우아한 해법을 만들기도 합니다. 예를 들어, 하노이 탑 문제나 조합/순열 생성 알고리즘은 재귀를 사용하면 반복문보다 개념적으로 명확해질 때가 많습니다.

그러나 재귀를 사용할 때는 한계에 유의해야 합니다. 재귀 깊이가 너무 깊어지면 스택 오버플로로 인해 프로그램이 비정상 종료할 수 있습니다 (예: 1만 단계 이상의 재귀 호출은 스택 한계를 초과할 수 있습니다). 꼬리 재귀 최적화(TCO)가 지원되는 언어에서는 이 문제를 완화할 수 있습니다(안타깝게도 C++와 Java는 꼬리 호출 최적화를 지원하지 않지만, Python은 내부적으로 일부 꼬리 호출을 최적화합니다). 성능이 중요한 코드에서는 함수 호출 오버헤드를 피하기 위해 반복문을 사용한 풀이가 더 나을 수 있습니다. 재귀 함수는 반드시 올바른 기저 사례를 갖고, 각 재귀 단계마다 그 기저 사례로 나아가도록 설계되어야 합니다 (무한 재귀를 방지하기 위함입니다).

재귀 호출 추적 – 예시: 이해를 돕기 위해 간단한 재귀 함수를 추적해보겠습니다. 아래는 배열의 합을 재귀로 계산하는 파이썬 코드입니다:

```
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
```

`recursive_sum([3,1,4,1])`을 호출하면, 먼저 3에 `recursive_sum([1,4,1])`의 결과를 더합니다. `recursive_sum([1,4,1])`은 1에 `recursive_sum([4,1])`의 결과를 더합니다. `recursive_sum([4,1])`은 4에 `recursive_sum([1])`의 결과를 더합니다. `recursive_sum([1])`은 1에 `recursive_sum([])`의 결과를 더합니다. `recursive_sum([])`은 비어 있는 배열이므로 0을 반환합니다 (기본 사례). 그러면 호출 스택이 풀리면서 각 덧셈이 완료됩니다: 0 + 1 → 1, 1 + 4 → 5, 5 + 1 → 6, 6 + 3 → 9. 결과로 9가 반환됩니다. 각 호출은 다음 호출의 결과를 기다린다는 점에서 호출 스택의 작동 방식을 잘 보여줍니다. 재귀는 처음엔 직관적이지 않을 수 있지만, 이와 같이 호출을 추적해보거나 디버거/출력문을 사용하면 흐름을 이해하는 데 큰 도움이 됩니다.

분할 정복 (Divide and Conquer)

분할정복은 문제를 더 작고 독립적인 하위 문제들로 나누고, 이를 각각 해결한 뒤 그 결과를 결합하여 원래 문제를 푸는 알고리즘 설계 패러다임입니다【17†】. 간단히 말해: - **분할(Divide):** 문제를 같은 유형의 두 개 이상의 독립된 하위 문제로 분할합니다. - **정복(Conquer):** 각 하위 문제를 재귀적으로 풉니다(하위 문제가 충분히 작아 직접 풀 수 있는 경우가 되면 재귀를 멈춥니다). - **결합(Combine):** 하위 문제들의 결과를 합쳐 원래 문제의 최종 해답을 만듭니다.

이 접근 방식은 매우 강력합니다. 왜냐하면, 문제를 절반으로 줄여 푸는 방식은 복잡도를 크게 낮출 수 있기 때문입니다. 많은 효율적인 알고리즘이 분할정복을 재귀와 함께 사용하며, 마스터 정리(Master Theorem)와 같은 수학적 도구를 이용하여 재귀식의 실행 시간을 분석합니다.

분할정복 알고리즘의 유명한 예로는 병합 정렬, 퀵 정렬, 이진 탐색, 제곱을 이용한 거듭제곱, 스트라센(Strassen)의 행렬 곱셈 등이 있습니다【18†】. 여기서는 몇 가지 핵심 예제를 살펴보겠습니다: 병합 정렬, 빠른 거듭제곱, 행렬 거듭제곱.

병합 정렬 (Merge Sort)

병합 정렬은 분할정복을 이용한 대표적인 정렬 알고리즘입니다. 배열을 두 개의 절반으로 나누어 각각 정렬한 뒤, 정렬된 두 절반을 합쳐 최종 정렬된 배열을 얻습니다. 알고리즘은 다음과 같이 요약할 수 있습니다: 1. **분할(Divide):** 배열(또는 리스트)을 두 개의 거의 같은 크기로 나눕니다 (왼쪽 절반과 오른쪽 절반). 2. **정복(Conquer):** 왼쪽 절반을 재귀적으로 정렬하고, 오른쪽 절반도 재귀적으로 정렬합니다. 3. **결합(Combine):** 정렬된 두 절반을 하나의 정렬된 배열로 합칩니다.

재귀는 하위 배열의 크기가 0이나 1이 될 때 종료됩니다. 크기가 0이나 1인 배열은 그 자체로 이미 정렬된 상태이기 때문입니다(기저 사례). 실제 중요한 작업은 병합 단계에서 이루어집니다. 병합 단계는 두 개의 정렬된 리스트를 받아 하나의 정렬된 리스트로 효율적으로 합치는 작업입니다. 병합 정렬은 배열을 재귀적으로 분할한 다음 정렬된 하위 배열들을 병합합니다. 다음 그림은 정렬되지 않은 숫자 리스트가 단일 요소로 분할된 후, 다시 정렬된 상태로 병합되는 과정을 보여줍니다.

그림에서 알 수 있듯 원래 배열은 반복해서 분할되어 단일 요소 배열이 됩니다(단일 요소는 정의상 정렬된 상태입니다). 그런 다음 병합 과정이 시작됩니다. 인접한 단일 요소들이 쌍으로 병합되어 정렬된 쌍을 만들고, 그 쌍들이 병합되어 정렬된 네 요소 묶음을 만들며, 이 과정을 전체 배열이 정렬될 때까지 반복합니다. 이러한 이진 분할로 인해 n개의 요소를 정렬할 때 재귀 트리는 $\log_2(n)$ 단계가 생기며, 각 단계에서는 모든 하위 배열 병합에 총 $O(n)$의 시간이 소요됩니다. 결과적으로 병합 정렬은 최악의 경우와 평균적으로 모두 $O(n \log n)$ 시간에 실행되며, 병합을 위해 $O(n)$ 크기의 추가 공간이 필요합니다【19†】. 병합 정렬은 매우 효율적인 정렬 알고리즘이며, 중요하게도 같은 값의 상대적 순서를 유지하는 안정 정렬이면서도 최악의 경우에도 $O(n \log n)$을 보장합니다. (반면 퀵 정렬은 평균적으로는 더 빠르지만 최악의 경우 $O(n^2)$을 가집니다.)

시간 복잡도 분석: 병합 정렬의 시간 복잡도를 $T(n)$이라 하면, n개의 요소를 정렬할 때 절반 크기의 두 하위 문제로 분할하고 결과를 병합하는데 $O(n)$의 작업을 수행합니다. 따라서 점화식은 다음과 같습니다:

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

마스터 정리나 점화식 전개에 따르면, $T(n)$은 $O(n \log n)$이 됩니다. 공간 복잡도는 병합 시 사용하는 임시 배열 때문에 $O(n)$입니다(더 복잡한 로직의 인-플레이스(in-place) 버전도 있습니다). 병합 정렬의 예측 가능한 $O(n \log n)$ 성능은 최악의 경우 성능 보장이 중요한 상황(예: 연결 리스트 정렬, 디스크 외부 정렬 등)에서 특히 유용합니다.

병합 정렬 구현 예제를 살펴보겠습니다. C++14, Java 17, Python 3.10으로 병합 정렬을 구현해 보겠습니다. 배열의 [l..r] 구간을 정렬하는 재귀 함수와, 두 개의 정렬된 하위 배열을 병합하는 헬퍼 함수를 작성합니다.

C++14: 병합 정렬 (재귀 구현)

```
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++];
}

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
}
```

이 C++ 예제에서 `mergeSort(arr, 0, n-1)`를 호출하면 전체 배열이 정렬됩니다. [l, r] 구간을 두 부분으로 나누고, 각 부분을 재귀적으로 정렬한 후 `merge` 함수를 호출하여 두 부분을 합칩니다. `merge` 함수는 왼쪽 부분과 오른쪽 부분의 임시 배열을 만들어 $O(n)$ 시간에 이 둘을 병합합니다.

Java 17: 병합 정렬 (재귀 구현)

```
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
}
```

자바 버전도 유사합니다. `mergeSort(arr, 0, n-1)`를 호출하여 시작합니다. 마찬가지로 임시 배열에 복사한 뒤 병합합니다. (면접에서는 인덱스 계산으로 제자리 병합을 할 수도 있지만, 임시 배열을 사용하는 것이 구현상 더 쉽고 안전합니다.)

Python 3.10: 병합 정렬 (재귀 구현)

```
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
```

파이썬에서는 명확성을 위해 새로운 리스트를 반환합니다. 배열을 절반으로 나누고 각각을 재귀적으로 정렬한 후 이 둘을 합칩니다. 파이썬의 리스트 슬라이싱과 `extend` 메서드를 이용하면 병합 단계가 간결해집니다. 전체 복잡도는 여전히 $O(n \log n)$입니다.

Walkthrough: - `[38, 27, 43, 3, 9, 82, 10]` 배열을 병합 정렬한다고 가정합시다. - 배열을 `[38, 27, 43, 3]`과 `[9, 82, 10]`으로 나눕니다. - `[38, 27, 43, 3]`을 재귀적으로 정렬하여 `[38, 27]`과 `[43, 3]`으로 분할합니다. - 단일 원소가 될 때까지 분할한 후, `[38]`과 `[27]`을 합쳐 `[27, 38]`을 만들고, `[43]`과 `[3]`을 합쳐 `[3, 43]`을 만듭니다. - 이어서 `[27, 38]`과 `[3, 43]`을 합쳐 `[3, 27, 38, 43]`을 만듭니다. - 한편 `[9, 82, 10]`은 `[9, 82]`와 `[10]`으로 분할됩니다. `[9, 82]`는 다시 `[9]`와 `[82]`로 분할되어 `[9, 82]`가 되고, 이것을 `[10]`과 병합하여 `[9, 10, 82]`를 만듭니다. - 마지막으로 `[3, 27, 38, 43]`과 `[9, 10, 82]`를 병합하여 `[3, 9, 10, 27, 38, 43, 82]`를 얻습니다.

병합 정렬은 각 병합 단계에서 두 하위 리스트가 이미 정렬되어 있음을 보장합니다. 따라서 두 리스트를 한 번씩만 순회하여 병합할 수 있어 선형 시간에 병합이 가능합니다. 이러한 정렬된 병합이 $O(n \log n)$ 복잡도의 이유입니다. 병합 정렬의 성능은 예측 가능하고 구현이 비교적 단순하기(특히 재귀적 구현이 간결함), 분할정복의 전형적인 예가 됩니다.

빠른 거듭제곱 (Exponentiation by Squaring)

분할정복의 고전적 응용 중 하나는 빠른 거듭제곱(fast exponentiation)입니다. 이 방법은 이진 거듭제곱(binary exponentiation) 또는 제곱을 통한 거듭제곱(exponentiation by squaring)으로도 알려져 있습니다. 목표는 a^n(수 a의 n제곱)을 a를 n번 곱하는 단순한 방법보다 훨씬 빠르게 계산하는 것입니다. 핵심은 지수를 이진수로 표현하여 곱셈 횟수를 $O(\log n)$으로 줄이는 것입니다【20†】.

이 방법은 다음 수학적 성질을 사용합니다:
- $a^{2k} = (a^k)^2$. (지수가 짝수인 경우, 절반에 해당하는 거듭제곱 결과를 제곱하면 됩니다.)
- $a^{2k+1} = a \times (a^k)^2$. (지수가 홀수인 경우, 하나의 a를 떼어내고 나머지를 제곱합니다.)

이를 통해 거듭제곱을 재귀적으로 정의할 수 있습니다:
- **기저 사례:** $a^0 = 1$ (0제곱은 1)
- **n이 짝수인 경우:** $a^n = (a^{n/2})^2$. (지수를 절반으로 계산한 후 제곱합니다.)
- **n이 홀수인 경우:** $a^n = a \times (a^{(n-1)/2})^2$. (지수가 홀수면 a를 하나 분리하고 나머지를 제곱합니다.)

이 재귀적 방법은 각 단계에서 지수를 대략 반으로 줄입니다. 결과적으로 곱셈 횟수를 $O(n)$에서 $O(\log n)$으로 줄일 수 있습니다. 예를 들어, a^{13}을 계산하려면 13을 이진수로 표현하면 1101(8 + 4 + 0 + 1)입니다. 따라서 $a^{13} = a^8 \times a^4 \times a^1$이 됩니다. 여기서 $a^8 = (a^4)^2$, $a^4 = (a^2)^2$와 같은 방식으로 계산합니다. 총 12번 곱하는 대신, 대략 $\lceil \log_2 13 \rceil = 4$번의 제곱 연산과 홀수 지수에 대한 몇 번의 추가 곱셈으로 해결할 수 있습니다. n이 커질수록 이러한 절약은 매우 커집니다.

이제 세 가지 언어로 빠른 거듭제곱 함수를 구현해 봅시다.

C++14: 이진 거듭제곱 (재귀 구현)

```
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: 이진 거듭제곱 (재귀 구현)

```
static long fastPower(long x, long 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: 이진 거듭제곱 (재귀 구현)

```
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
```

이 함수들은 모두 분할정복 방식을 사용합니다. $x^{n/2}$를 계산하여 제곱하는 것이 “정복(conquer)” 단계이고, n이 홀수면 추가로 x를 곱합니다. 재귀 깊이는 n을 절반씩 줄이기 때문에 $O(\log n)$입니다. 따라서 곱셈 횟수도 대략 $\log_2(n)$에 비례합니다. 예를 들어 2^{32}를 구하는 경우, 단순 계산법은 31번의 곱셈이 필요하지만, 이진 거듭제곱을 사용하면 이진수 100000(=32)이므로 단 6번의 곱셈만으로 해결할 수 있습니다. 일반적으로 이진 거듭제곱은 $O(\log n)$ 시간에 동작하며【21†】, 이는 $O(n)$보다 훨씬 우수합니다. 이 기법은 특히 크립토그래피에서 큰 지수를 다룰 때 모듈러 거듭제곱(large exponent mod M)이나 반복 제곱이 필요한 알고리즘(예: 이후에 다룰 행렬 거듭제곱) 등 많은 분야에서 매우 유용합니다.

예를 들어 3^{13}을 계산하는 방법은 다음과 같습니다: - **n=13 (홀수):** $3^{(13-1)/2} = 3^6$을 계산하고 제곱한 뒤 3을 곱합니다. - **3^6을 구하기 위해:**
- **n=6 (짝수):** 3^3을 계산하고 제곱합니다.
- **3^3을 구하기 위해:**
- **n=3 (홀수):** $3^{(3-1)/2} = 3^1$을 계산하고 제곱한 뒤 3을 곱합니다.
- 3^1은 기저 사례로 3입니다.
- 3^1을 제곱하면 9이고, 여기에 3을 곱하면 27이 되므로 $3^3 = 27$입니다.
- 이제 3^3을 제곱합니다: $27^2 = 729$, 따라서 $3^6 = 729$입니다.
- 3^6을 제곱합니다: $729^2 = 531441$. 여기에 3을 곱합니다: $531441 \times 3 = 1{,}594{,}323$, 이는 3^{13}입니다.
- 이 과정을 통해 12번의 곱셈 대신 몇 번의 곱셈만 수행했습니다.

대부분의 언어에는 `pow`(파이썬)나 `Math.pow`(자바) 같은 내장 거듭제곱 함수가 있으며, 내부적으로 이진 거듭제곱을 사용합니다. 하지만 스스로 구현하는 방법을 아는 것이 중요합니다. 인터뷰에서 거듭제곱 계산 최적화에 대해 논의하거나, 바이너리 탐색과 같이 분할(반으로 나누기) 전략이 필요한 알고리즘의 일부로 이진 거듭제곱을 설명할 수 있습니다.

행렬 거듭제곱 (Matrix Exponentiation)

거듭제곱을 위한 분할정복 기법은 수에만 국한되지 않습니다. 임의의 결합 법칙이 성립하는 연산에 적용할 수 있으며, 특히 행렬 곱셈에 유용합니다. 행렬 거듭제곱(Matrix Exponentiation)도 동일한 제곱 기법을 사용하여 행렬 M의 n제곱을 빠르게 계산합니다【22†】. 이는 선형 점화식(예: 피보나치 수열, 동적 계획법의 상태 전이 등)을 빠르게 계산할 때 특히 유용합니다. 개념은 간단합니다: 행렬의 n제곱을 구할 때, 행렬을 n번 곱하지 않고 $\log_2(n)$번만 제곱하면 된다는 것입니다.

예를 들어 피보나치 수열은 행렬 거듭제곱으로 계산할 수 있습니다. 피보나치 점화식은 2×2 행렬로 표현할 수 있습니다: $\begin{pmatrix}1 & 1\ 1 & 0\end{pmatrix}$. 이 행렬을 M이라 할 때, M^n의 (0,0) 성분은 F_{n+1}을, (0,1) 성분은 F_n을 나타냅니다. 분할정복을 사용하면 M^n을 n번 곱하는 대신 $O(\log n)$번 곱셈으로 계산할 수 있습니다. 2×2 행렬 곱셈은 연산 수가 상수이므로, 피보나치 예시에서는 $O(\log n)$ 시간에 계산할 수 있습니다【22†】. 일반적인 $d\times d$ 행렬의 경우, 각 행렬 곱셈은 $O(d^3)$ 연산이 필요하므로 전체 복잡도는 $O(d^3 \log n)$이 됩니다. 이 또한 n이 큰 경우 $O(n)$과 비교하면 큰 이점을 가져다줍니다. 특히 알고리즘 대회에서는 모듈러 연산 하에서 점화식을 빠르게 풀거나, 동적 계획법의 전이 행렬을 n단계까지 거듭제곱하는 데 행렬 거듭제곱 기법이 자주 사용됩니다.

복잡도: 이진 제곱을 이용한 행렬 거듭제곱은 $O(\log n)$번의 행렬 곱셈이 필요합니다. 2×2 행렬 곱셈은 연산 수가 상수이므로, 피보나치 예시에서는 $O(\log n)$ 시간에 계산할 수 있습니다【22†】. 일반적인 $d\times d$ 행렬의 경우, 각 행렬 곱셈은 $O(d^3)$이므로 전체 복잡도는 $O(d^3 \log n)$이 됩니다. 이 또한 n이 큰 경우 $O(n)$과 비교하면 큰 이점을 가져다줍니다. 특히 알고리즘 대회에서는 모듈러 연산 하에서 점화식을 빠르게 풀거나, 동적 계획법의 전이 행렬을 n단계까지 거듭제곱하는 데 행렬 거듭제곱이 자주 사용됩니다.

백트래킹 (Backtracking)

백트래킹은 모든 가능성을 체계적으로 탐색하면서, 유효한 해답을 찾을 수 없는 경로는 즉시 되돌아가는(백트랙하는) 알고리즘 기법입니다. 해답을 한 단계씩 구성하며, 어느 단계에서든 현재 경로가 유효한 해답으로 이어질 수 없다고 판단되면 해당 경로를 포기(백트랙)합니다. 본질적으로 백트래킹은 선택들의 상태 공간 트리(state-space tree)에 대한 깊이 우선 탐색(DFS)이며, 불가능한 분기는 가지치기(pruning)하는 방식입니다. 이는 브루트 포스처럼 모든 가능성을 무작정 탐색하는 대신, 불가능한 분기들을 미리 제거합니다【23†】.

더 구체적으로 말하면, 백트래킹은 해답을 한 단계씩 구성해 나갑니다(예: 배열, 문자열, 설정을 하나씩 채워 넣음). 각 단계에서 현재까지의 부분 해답이 여전히 유효한지, 해결로 이어질 수 있을지 판단합니다. 유효하면 다음 단계를 추가하기 위해 재귀적으로 진행합니다. 유효하지 않으면(제약을 위반하는 경우) 마지막 단계에서 한 단계 되돌아가(백트랙) 다음 옵션을 시도합니다. 이러한 "시도 → 실패 시 백트랙 → 다음 시도" 과정을 해답을 찾거나 모든 옵션이 소진될 때까지 반복합니다【24†】.

“백트래킹은 미로를 탐색하는 것과 같다. 막다른 길에 다다르면 한 단계 되돌아가 다른 길을 선택한다.”는 설명이 있습니다. 이 방법은 조합적 탐색 문제에 자주 사용됩니다: 모든 순열과 조합 구하기, N-퀸, 스도쿠, 십자말 풀이, 미로 찾기, 그래프 색칠, 해밀토니안 경로 찾기 등입니다. 본질적으로 백트래킹은 최적화된 완전탐색입니다. 여전히 많은 경우의 수를 탐색할 수 있지만, 실패를 조기에 인지하여 가능성이 없는 많은 경로를 미리 제거합니다.

백트래킹의 작동 원리: 일반적인 백트래킹 알고리즘은 다음과 같이 설명할 수 있습니다: 1. **선택(Choose):** 가능한 옵션 중 하나를 선택합니다 (예: 다음 위치에 넣을 숫자 선택, 경로의 방향 선택 등). 2. **제약 검사(Constraint Check):** 선택한 옵션이 제약 조건을 위반하지 않는지 확인합니다. 만약 위배한다면 해당 경로를 가지치기하고 즉시 되돌아갑니다(더

이상 탐색하지 않음). 3. **재귀(Explore):** 선택이 유효하면 이를 적용하고 다음 단계(다음 위치/요소)를 위해 재귀 호출을 수행합니다. 4. **백트랙(Backtrack):** 재귀 탐색이 끝나면 (해답을 찾았거나 실패로 돌아왔다면) 선택을 취소(이전 상태 복원)하고, 다음 가능한 옵션을 시도합니다. 5. **반복:** 모든 옵션을 시도하거나 해답을 찾을 때까지 위 과정을 반복합니다.

이 방식은 선택들의 상태 공간 트리를 깊이 우선 탐색(DFS)하며【25†】, 부분 구성이 유효하지 않으면 즉시 가지치기하여 탐색을 중단합니다. 브루트 포스가 무조건 모든 경우를 탐색하는 데 비해, 백트래킹은 가능성이 없는 경로를 미리 건너뜀으로써 작업량을 크게 줄입니다.

아래 그림은 4-퀸 퍼즐의 백트래킹 탐색 트리입니다. 각 레벨은 퀸을 놓는 행을 나타냅니다. 퀸끼리 공격이 가능한 위치(빨간 X로 표시)는 폐기되고, 이전 행으로 돌아가(백트랙) 다른 열을 시도합니다. 유효한 해답(녹색 체크 표시된)이 하나 발견됩니다. 이 과정을 통해 $4^4 = 256$가지 가능한 배치 중 대부분을 충돌을 감지하여 초기 단계에서 제거하기 때문에 백트래킹의 강력함을 알 수 있습니다.

응용 분야: 백트래킹은 순열, 조합, 부분집합 생성 등 조합적 생성 문제, 스케줄링과 같은 제약 만족 문제(스도쿠, N-퀸, 십자말 풀이 등), 그리고 특정 조건을 만족하는 순서나 구성을 찾는 탐색 문제에 적합합니다. 많은 NP-완전 문제(예: SAT, 그래프 색칠, 해밀토니안 경로)에도 백트래킹 해법이 있습니다. 최악의 경우에는 여전히 지수 시간 복잡도를 가질 수 있지만, 대부분의 실제 상황에서는 백트래킹이 탐색 공간의 많은 부분을 제거하여, 단순 완전탐색으로는 해결하기 어려운 문제까지 해결할 수 있게 해줍니다. 백트래킹의 효율성은 제약 조건의 강도에 달려 있습니다. 제약이 강할수록 가지치기가 많이 일어나므로 탐색해야 할 공간이 크게 줄어듭니다.

백트래킹 vs 완전탐색: 완전탐색과 백트래킹의 차이를 살펴보면, 완전탐색은 무조건 모든 경우를 나열하지만, 백트래킹도 모든 경우를 탐색하지만 문제 특성에 맞는 논리를 이용하여 불필요한 경로를 건너뜁니다【23†】. 만약 부분 해답의 유효성을 검사할 수 없어 가지치기가 불가능하다면, 백트래킹도 사실상 완전탐색과 동일합니다. 그러나 퍼즐처럼 부분 할당만으로도 잘못된 경로를 알아낼 수 있는 경우에는 백트래킹을 통해 작업량을 크게 줄일 수 있습니다.

예제 – 순열 생성: 조합적 생성 문제의 전형적인 예로, 숫자 리스트의 모든 순열을 생성하는 백트래킹을 살펴보겠습니다. 순열은 한 번에 한 요소씩 구성하고, 이후 교환하여 다른 조합을 탐색해야 하므로 전형적인 백트래킹 문제입니다. C++, Java, Python으로 방법을 보여드리겠습니다.

C++14: 리스트의 모든 순열을 생성하는 백트래킹

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

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);
}
```

이 C++ 예제에서 `backtrackPermute`는 `arr[l..r]`의 모든 순열을 생성합니다. 함수는 먼저 위치 l에 올 수 있는 모든 요소를 차례로 선택하여(l번째와 i번째를 swap) 재귀적으로 나머지를 순열을 수행합니다. 만약 `l == r`가 되면, 하나의 완전한 순열이 만들어진 것이므로 출력하거나 저장합니다. 핵심은 원소를 교환한 뒤, 이후 다시 되돌려 놓는(swap back) 부분입니다. 이것이 바로 백트래킹 단계로, 이후 다른 원소를 시도할 수 있게 현재 상태를 복원합니다. 이 알고리즘의 시간 복잡도는 $O(n \times n!)$인데, $n!$개의 순열을 생성하고 각 순열을 출력하거나 저장하는 데 길이 n를 요구하기 때문입니다. 하지만 이 방식은 순열 공간을 중복 없이 효율적으로 탐색합니다.

Java 17: 백트래킹을 이용한 순열 생성

```
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);
  }
}
```

이 자바 구현도 같은 방식으로 작동합니다. 현재 위치의 원소와 나머지 원소들을 교환한 뒤, 재귀 호출하고, 다시 교환하여 원래 상태로 복원합니다. 도우미 `swap` 함수를 사용하여 코드가 간단해졌습니다.

Python 3.10: 백트래킹으로 순열 생성

```
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)
```

파이썬의 튜플 할당을 이용하면 교환(swap)이 매우 간편해집니다. 이 예제에서도 같은 패턴이 나타납니다: 각 위치 l에 대해 가능한 선택을 고르고, 재귀적으로 탐색한 후, 선택을 원래대로 되돌립니다.

이 예제는 일반적인 백트래킹 패턴을 보여줍니다: **선택 → 탐색 → 선택 취소**. 모든 순열을 생성하는 것은 시간 복잡도 $O(n!)$로 매우 높지만, 백트래킹은 자연스러운 해결 방법입니다. n이 비교적 작을 때(예: $n=6,7$ 정도)에는 충분히 실행 가능합니다. 인터뷰 환경에서는 n이 작거나(순열이나 조합 문제에서 보통 $n\le 10\sim15$ 정도) 추가적인 제약으로 탐색이 많이 줄어드는 경우 백트래킹 해법이 허용되곤 합니다.

또 다른 예 – **N-퀸 문제(개략):** 제약 검사를 동반한 백트래킹을 보기 위해 N-퀸 문제를 생각해봅시다. 퀸을 행 단위로 놓습니다. 각 행마다 모든 열을 시도해보고, 새로 놓는 퀸이 기존 퀸들과 충돌하지 않는지 확인합니다 (같은 열이나 대각선에 두 퀸이 있으면 충돌). 충돌이 있으면 그 열을 건너뛰어(가지치기) 다음 열을 시도합니다. 충돌이 없으면 퀸을 배치하고 다음 행으로 이동합니다. 모든 행에 퀸을 성공적으로 배치하면 하나의 해답을 찾은 것입니다. 만약 어떤 행에서 놓을 수 있는 위치가 없다면, 이전 행으로 돌아가(백트랙) 퀸을 다음 가능한 열로 이동시킵니다. 이렇게 하면 이전 예제처럼 부분 구성(지금까지 배치된 퀸들)을 단계마다 검증하며 잘못된 경로를 배제합니다【27†】【28†】. 4-퀸 문제의 상태 공간 트리는 그 전형적인 그림으로, $4^4=256$가지 가능한 배치 중 대부분을 충돌을 감지하여 초기 단계에서 제거합니다.

백트래킹 사용 시기: 백트래킹은 조합적으로 가능한 모든 경우의 공간을 탐색해야 하고, 해답 후보를 단계적으로 구성하고 부분적으로 검증할 수 있을 때 사용합니다. 부분 해답이 잘못된 경우(제약을 위반하는 경우)를 조기에 감지할 수 있다면, 백트래킹은 시간을 크게 절약합니다. 퍼즐과 같은 문제나 제약 만족 문제(예: 스도쿠 풀기: 칸을 채우고 충돌이 생기면 백트랙), 그리고 조합적 객체(부분집합, 순열, 조합, 분할 문제 등)를 생성할 때 특히 유용합니다. 다만, 탐색 공간이 매우 넓고 제약이 약해 가지치기가 잘 일어나지 않으면 여전히 느릴 수 있습니다. 이럴 때는 선택 순서 정하기, 제약 해결에서 앞서보기(forward-checking) 같은 추가 휴리스틱이나 최적화를 사용해 백트래킹 성능을 향상시킵니다.

결론 및 요점

이번 주 주제인 완전탐색, 재귀, 분할정복, 백트래킹은 알고리즘 설계를 위한 기본 전략 도구들입니다: - 완전탐색은 정답의 기준을 제공하지만, 종종 효율성을 높이기 위한 최적화가 필요합니다. - 재귀는 문제를 하위 문제로 분할하여 분할정복과 백트래킹을 우아하게 구현할 수 있게 해줍니다. - 분할정복은 문제를 나누어 효율성을 크게 높입니다 (예: 정렬과 거듭제곱 계산을 $O(n^2)/O(n)$에서 $O(n \log n)/O(\log n)$으로 개선). - 백트래킹은 방대한 탐색 공간을 가지치기하여 효과적으로 탐색하며, 조합적 탐색 문제에서 필수적인 기법입니다.

각 방법론은 전형적인 활용 사례와 시간 및 구현 복잡도에 따른 장단점을 갖습니다. 알고리즘 문제 해결사로서, 주어진 문제에 어떤 패러다임이 적합한지 인식하고 효율적으로 구현할 수 있어야 합니다. 연습이 중요합니다: 재귀 풀이를 작성해보고 필요하면 반복문으로 바꿔 보고, 작은 퍼즐에 백트래킹을 적용해 보고, 분할정복 알고리즘의 점화식을 분석해 보세요. 이러한 기법들을 숙달하면 많은 코딩 인터뷰 문제와 알고리즘 시험에서 우수한 준비 상태를 갖추게 될 것입니다. 처음에는 완전탐색으로 시작하여 필요에 따라 재귀, 분할정복, 백트래킹을 적용해 문제를 해결해 나갈 수 있을 것입니다. 행복한 코딩 되세요! 다음 시간에 만나요!

[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/
