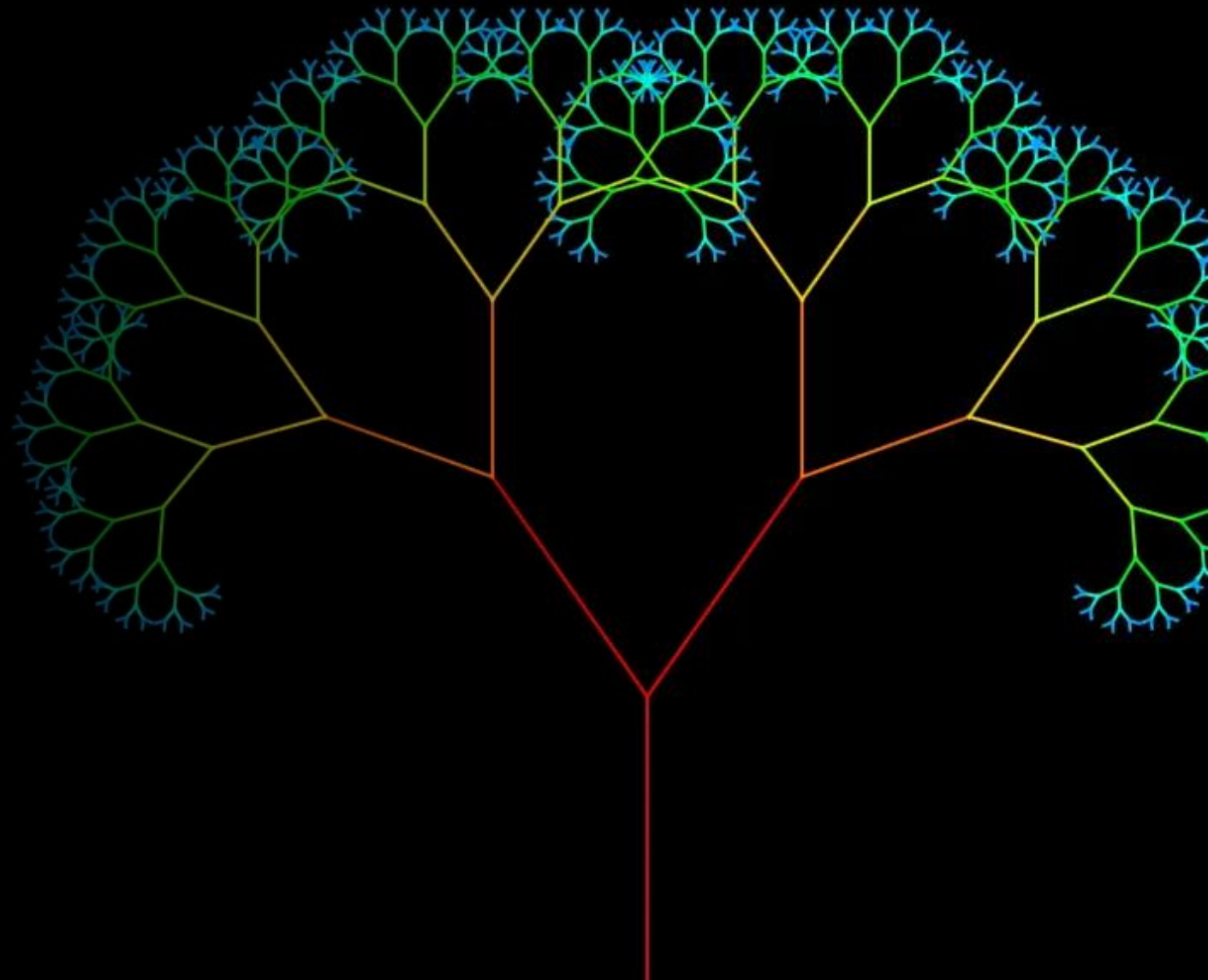
코드에 대한 다양한 관점

재귀를 중심으로 톺아보기

재귀에 따른 알고리즘 기법



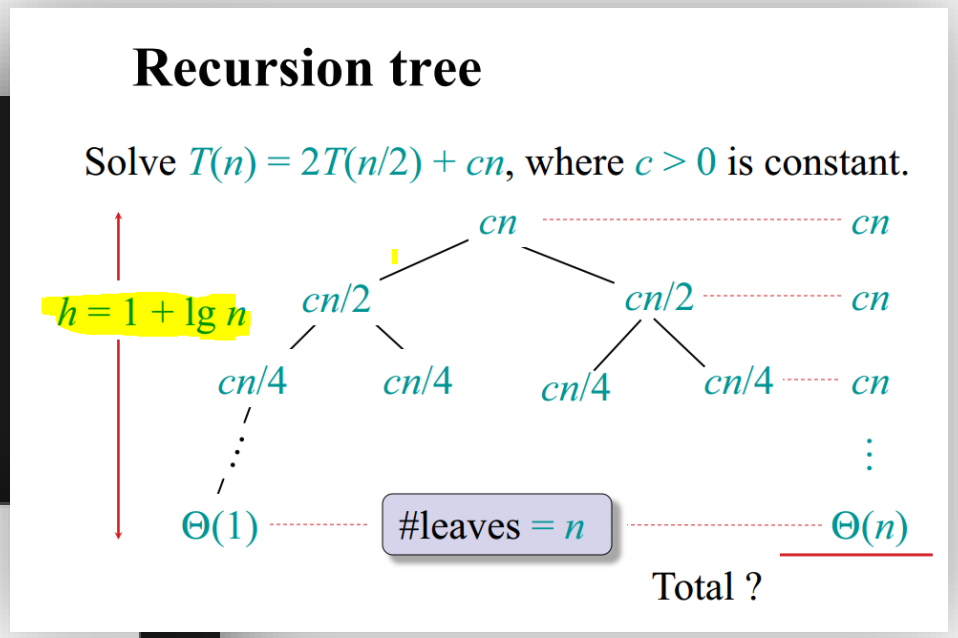
Brute-Force
Dynamic Programming
Decrease and Conquer
Divide and Conquer

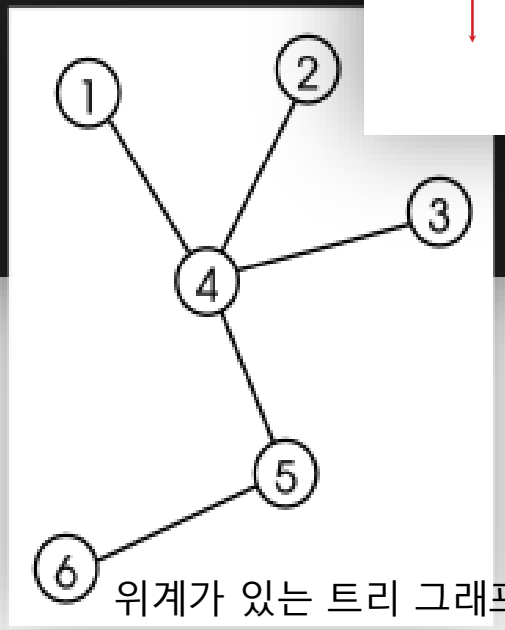
종료 조건, 재귀 조건, 문제 분할 및 결합

```python
def recursive_function(problem):
    # 1) 종료 조건 확인
    if is_small_enough(problem):
        return solve_directly(problem)

    # 2) 문제 분할 or 축소
    subproblem1, subproblem2 = divide_or_reduce(problem)

    # 3) 재귀 호출
    result1 = recursive_function(subproblem1)
    result2 = recursive_function(subproblem2)

    # 4) 결과 결합
    return combine(result1, result2)
```

Divide and Conquer: 문제 쪼개기

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
                    절반으로 나누어 재귀 실행
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])    # 왼쪽 절반
    right = merge_sort(arr[mid:]) # 오른쪽 절반

    # 두 정렬된 배열 합치기
    return merge(left, right)
```

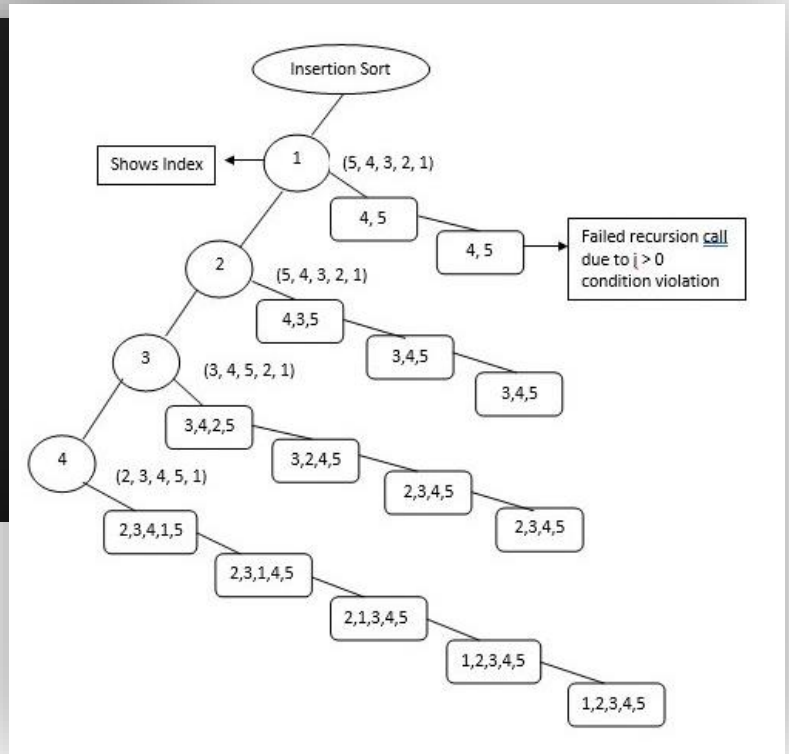


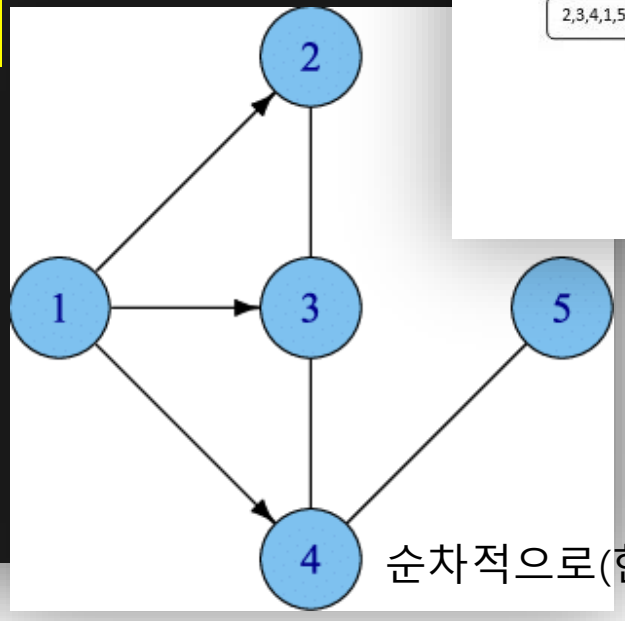


위계가 있는 트리 그래프

Decrease and Conquer: 서서히 해결

```python
def insertion_sort_recursive(arr, n=None):
    if n is None:
        n = len(arr)
    # base case: 길이 1이면 정렬 끝
    if n <= 1:
        return
    순차적으로 하나씩 감소 재귀(이후 점점 크게 해결)
    # 일단 n-1 개는 이미 정렬되어 있다고 가정하고(재귀)
    insertion_sort_recursive(arr, n-1)

    # 이제 n번째 원소를 끼워넣음
    key = arr[n-1]
    j = n-2
    while j >= 0 and arr[j] > key:
        arr[j+1] = arr[j]
        j -= 1
    arr[j+1] = key
```



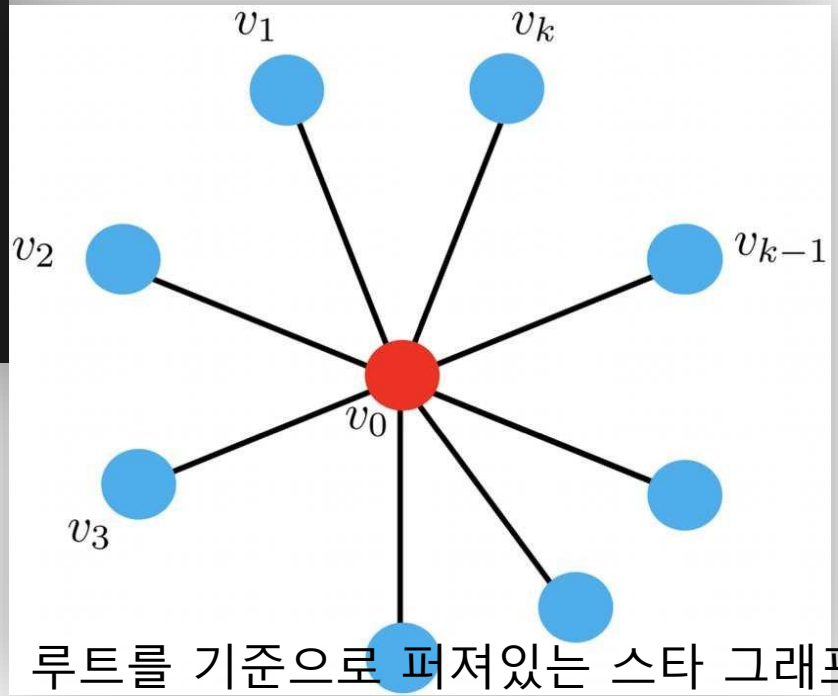


순차적으로(한줄로) 연결된 체인 그래프

Brute-Force: 가능한 한 전부 탐색

```python
def knapsack_brute_force(weights, values, capacity):
    def helper(index, current_weight, current_value):
        # 종료 조건: 모든 아이템을 확인했을 때
        if index == len(weights):
            return current_value if current_weight <= capacity else 0

        # 1) 현재 아이템을 선택하지 않는 경우
        exclude = helper(index + 1, current_weight, current_value)

        # 2) 현재 아이템을 선택하는 경우 (무게 초과 방지)
        include = 0
        if current_weight + weights[index] <= capacity:
            include = helper(index + 1, current_weight + weights[index], current_value + values[index])

        # 최대 가치 반환
        return max(exclude, include)

    return helper(0, 0, 0)
```

두 가지 가능한 경우를 모두 재귀



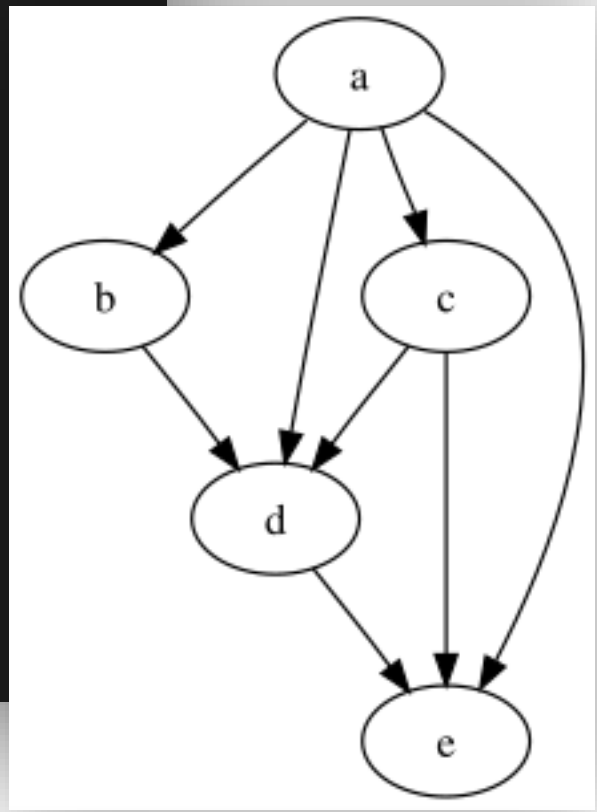
루트를 기준으로 퍼져있는 스타 그래프

Dynamic Programming: 중복 계산 저장

(메모이제이션)

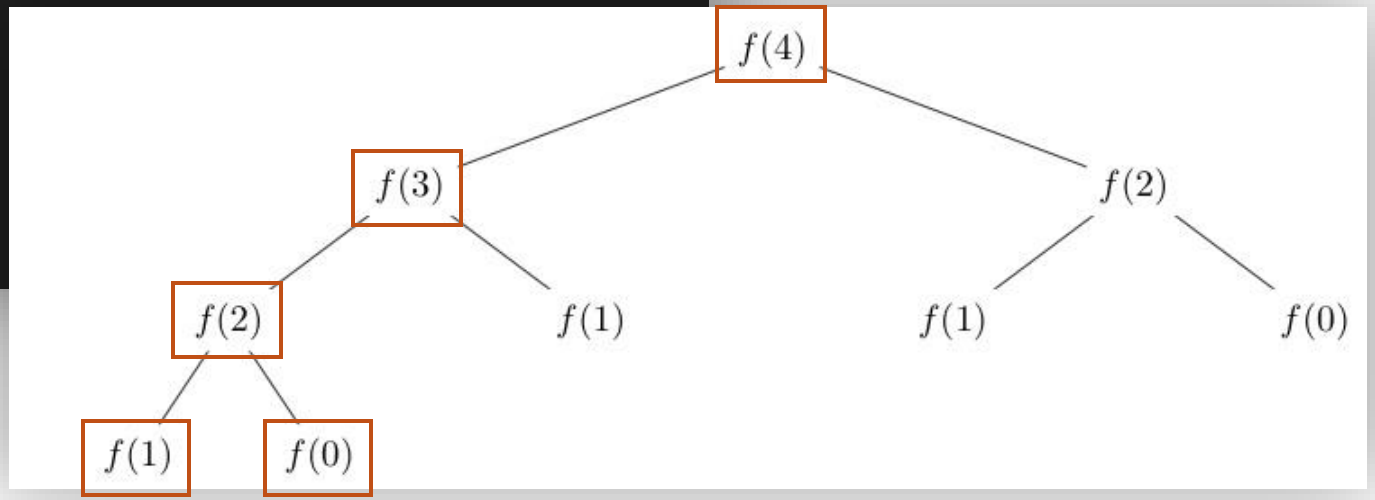
```python
memo = {}
def fib_dp(n):
    if n < 2:
        return n
    if n in memo: 메모에 속하면 재귀 건너뜀
        return memo[n]

    # 재귀로 브루트 포스처럼 계산하지만...
    result = fib_dp(n-1) + fib_dp(n-2)
    memo[n] = result
    return result
```

재귀 계산 값을 메모



+Bottom-up: 재귀의 반복문화

```python
def fibonacci_topological(n):
    if n < 2:
        return n
    fib = [0]*(n+1)
    fib[1] = 1
    for i in range(2, n+1):
        fib[i] = fib[i-1] + fib[i-2]
    return fib[n]
```



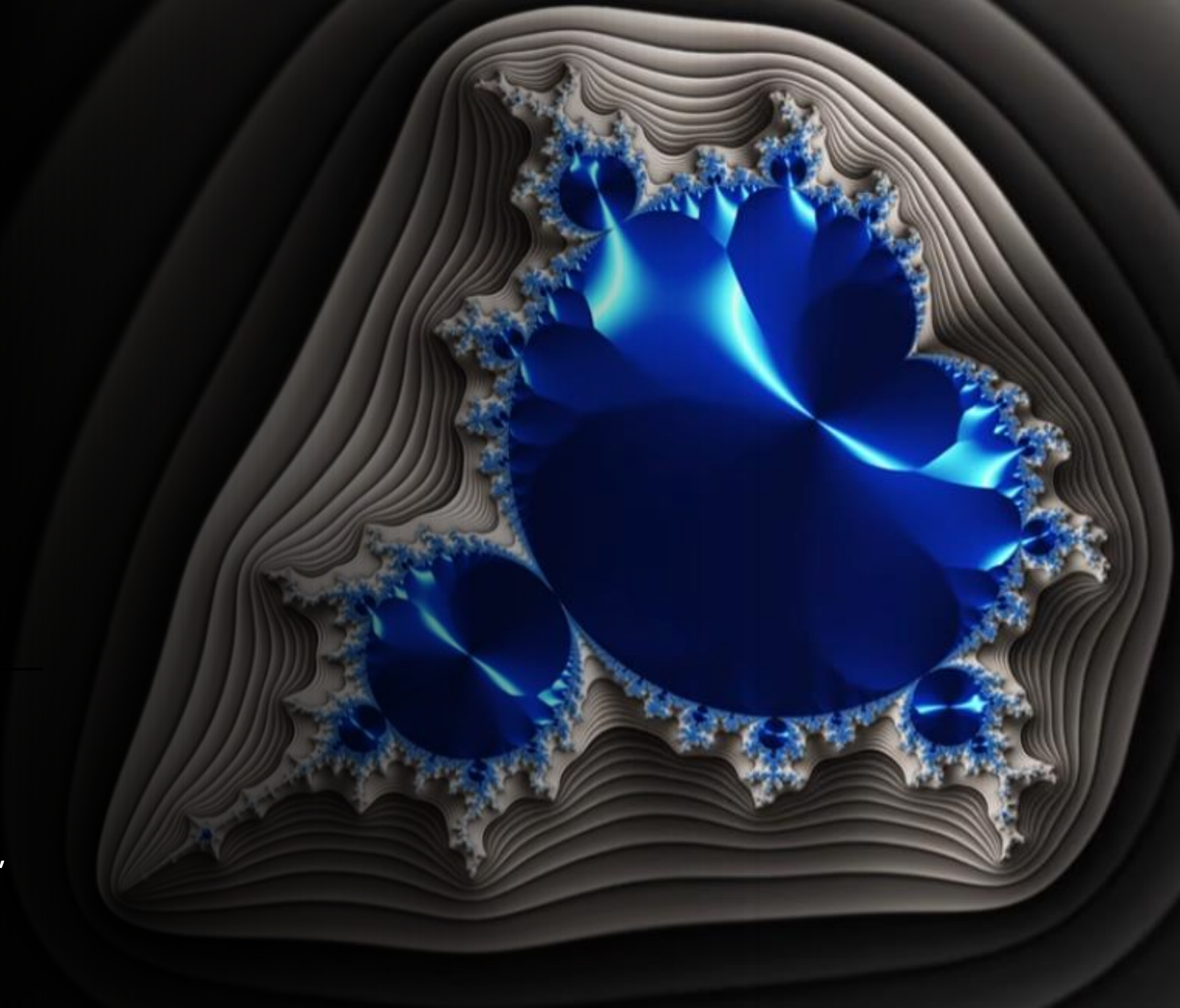
다만 재귀의 밑단과 위상 정렬을 알아야만 가능!

재귀가 주는 관점과 접근

Bottom-Up/Top-Down

Problem Definition & Solving

Approximation, On-line, Parallel, Machine Learning, Quantum



재귀는 어떻게 이루어지는가?

- **형태적으로** 열매를 향해 형성, **기능적으로** 뿌리를 향해 완성

1) 문제 간 **위계 질서** 파악 – 하위 문제 정의와 문제 간 관계
2) **문제 형태**에 따른 기법 변화 - 자료구조에 따른 재귀 기법
3) 재귀를 **적용한 사례** 및 폭넓은 접근

재귀 트리의 열매와 뿌리에 대해

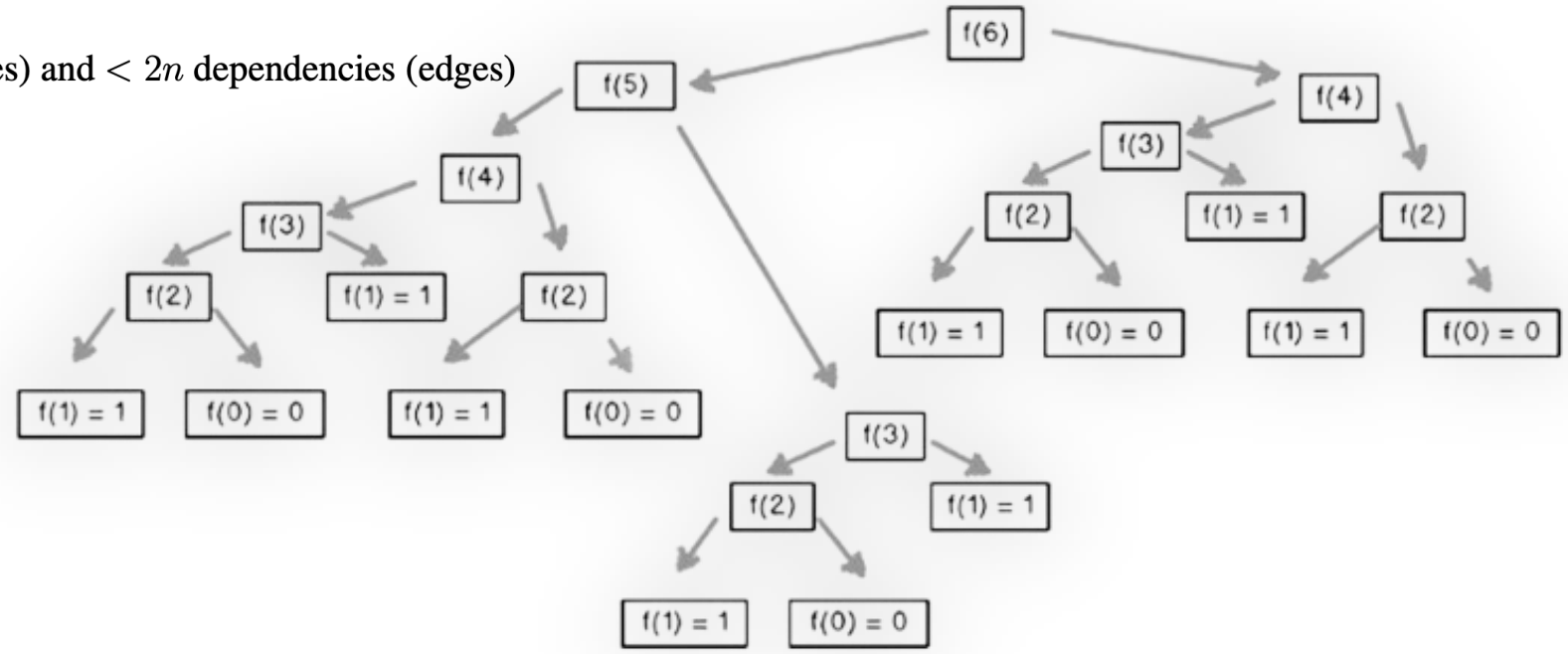
1.재귀 문제는 어떻게 구축해야 하는가?(SubProblem, Relation)
2.재귀의 흐름은 어떻게 진행되는가?(Topological Order)
3.어디까지 베이스 케이스로 잡아야 하는가?(Base Case)
4.어디부터 재귀 케이스를 시작해야 하는가?(Original Case)
5.재귀 시간 복잡도는 어떻게 이루어지는가?(Time-Complexity)

재귀 트리의 열매와 뿌리에 대해

- **S**ubproblems: $F(i) =$ the ith Fibonacci number F_i for $i \in \{0, 1, \ldots, n\}$
- **R**elation: $F(i) = F(i-1) + F(i-2)$ (definition of Fibonacci numbers)
- **T**opo. order: Increasing i
- **B**ase cases: $F(0) = 0, F(1) = 1$
- **O**riginal prob.: $F(n)$
- For Fibonacci, $n + 1$ subproblems (vertices) and $< 2n$ dependencies (edges)
- **T**ime to compute is then $O(n)$ additions

Top-Down

Bottom-Up



재귀 트리의 열매와 뿌리에 대해

- 문제 크기나 종료 조건을 알기 어려울 때 유리하게 작용하기도 함.
- 문제 크기가 점진적으로 줄어드는 경우(각 단계에서 몇 번의 반복이 필요한지 불분명한 경우), 종료 조건(Base Case)이 복잡한 경우, 탐색 경로가 다수 존재하는 경우에서 유리함.

- 간결한 재귀 코드에서 나오는 호출 스택의 흐름이 있는데, 이를 이해하는 것이 개발 역량에서 중요함.

1) 문제 간 위계 질서

- 문제 간 위계를 정의하는 것은 문제 풀이의 방향성과 효율성을 결정짓는 핵심 요소.

1. **Brute Force**는 매 인덱스마다 같은 위계를 가지며, 모든 가능한 경우의 수를 시도. 이 방식은 문제의 중복을 고려하지 않기 때문에 효율성이 부족.

2. **Dynamic Programming(DP)**은 문제 간의 관계를 방향성 있는 위계로 정의하고, 중복되는 하위 문제를 제거하거나 결과를 재활용함으로써 효율적으로 문제를 해결.

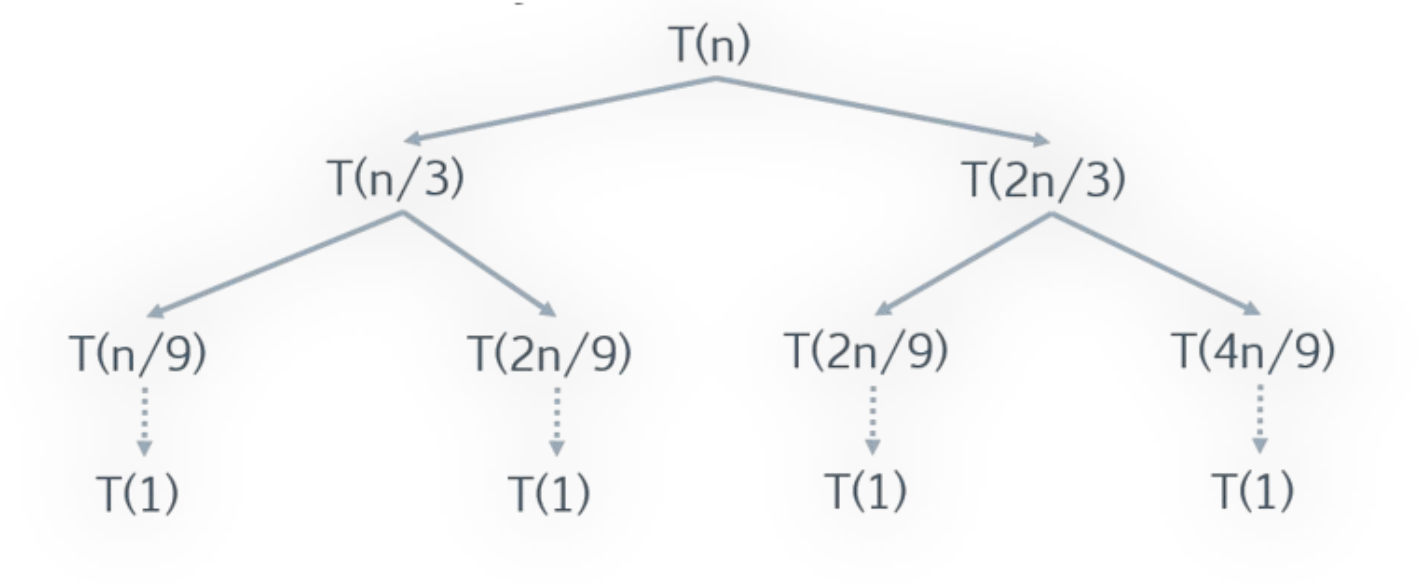
결론적으로, 효율적인 문제 해결을 위해서는 각 문제의 상황(조건)과 형태(자료구조), 문제 간의 관계를 명확히 정의하고 이를 기반으로 적절한 알고리즘 기법을 선택.

2) 문제 형태에 따른 기법 변화

- **Divide and Conquer**는 분할된 하위 문제를 독립적으로 처리하고 결과를 병합하는 방식으로, 트리 구조(분할된 하위 문제들 간의 계층적 관계)에서 자주 사용.
- **Decrease and Conquer**는 문제를 한 단계씩 축소하며 해결하는 방식으로, 순차적으로 연결된 체인(Chain) 그래프 구조에서 자주 사용.
- **Brute Force**는 모든 가능한 조합을 탐색하는 방식으로, 선택마다 갈래가 뻗어 나가는 트리 구조에서 사용.
- **Dynamic Programming**은 중복된 하위 문제를 저장해 결과를 재활용하는 방식으로, 방향성이 있지만 순환이 없는 그래프(DAG) 구조에서 사용.

2) 문제 형태에 따른 기법 변화

- 각 재귀 기법의 사고 흐름을 기하적 구조(RECURSION TREE)로 이해하면, 문제를 작은 단위로 나누고 결합하는 로직을 명확히 이해할 수 있음.
- 흥미롭게 생각해볼만 한 포인트!



3) 재귀를 적용한 사례 및 폭넓은 접근

- 근사 알고리즘: NP-Hard와 같은 풀기 어려운 문제를 "**최적해에 가까운(근사도)**" 답을 내기 위해 사용
- 재귀적 구조: 문제를 작게 분할하거나, **가지치기**를 통해 근사 해를 도출

분할 정복(Divide and Conquer) + 근사 접근

: 도시들을 몇 개의 지역(클러스터)으로 분할 후 재귀하여, 각 클러스터 내에서 가까운 근사 해를 얻음. 이후 이 해들을 붙이는 식(병합)으로 전체 투어를 구성.

무차별 대입법(Brute-Force) + 근사도 검사

: 완전 탐색을 재귀로 돌리되, "현재 경로 비용이 이미 근사 해보다 커졌다" 싶으면 중단(백트래킹). 이렇게 가지치기하면 지수 시간에서 상당 부분 절약 가능.

3) 재귀를 적용한 사례 및 폭넓은 접근

- 온라인 알고리즘: 입력(데이터)이 순차적으로 들어와 과거에 내린 결정을 바꿀 수 없는 상태, 동시에 미래 입력조차도 모르는 상태에서 의사 결정을 내리는 것
- 재귀는 문제의 **종료 조건(Base Case)**과 **구조적 특성(SubProblem, Relation)**을 이해하고, 이를 기반으로 문제를 점진적으로 분할하여 해결하는 방식. 그러나, 온라인 알고리즘은 미래 입력을 알 수 없기 때문에, 종료 조건이나 문제 분할의 논리를 정의하기 어려운 경우가 많음.

3) 재귀를 적용한 사례 및 폭넓은 접근

- **병렬 알고리즘**: 각 연산을 **병렬 프로세서**를 통해 처리하는 알고리즘
- 재귀적 구조: 재귀는 자연스레 분할 정복(Divide and Conquer) 구조를 가지는데, 분할된 하위 문제들이 서로 **독립적**이라면, 쉽게 병렬 처리로 넘어갈 수 있음.

Parallel Merge Sort

: 리스트를 절반으로 나눈 뒤(재귀적 분할), 각각 독립적으로 정렬을 수행하고 최종적으로 합침. 이때 하위 문제들을 각각 다른 스레드/프로세스로 진행.

Parallel DP

: LCS 문제에서 테이블을 만들 때, 각 칸(혹은 대각선 방향으로) 독립 계산이 가능하다면, 병렬로 업데이트를 진행할 수도 있음.

3) 재귀를 적용한 사례 및 폭넓은 접근

$0 \le k < m + n$를 만족하는 어떠한 k에서, $k = i + j$, $0 \le i < m$, and $0 \le j < n$을 만족하는 i와 j를 찾을 수 있고, subarray C[0]-C[k-1]은 subarray A[0]-A[i-1]과 subarray B[0]-B[j-1]의 merge 결과이다.

Parallel Merge Sort

- **co-rank(두 개 인덱스) function**

각 스레드 별로 배열을 나누어서 할당하고, 각 스레드에 할당된 subarray를 생성. 이후 각 스레드에 의해 생성된 output 요소의 rank(인덱스)를 알 수 있고, 그러면 각 스레드는 co-rank 함수를 사용.

- **Subproblems:** $S(i, j) =$ sorted array on elements of $A[i : j]$ for $0 \le i \le j \le n$
- **Relation:** $S(i, j) = \text{merge}(S(i, m), S(m, j))$ where $m = \lfloor (i + j)/2 \rfloor$
- **Topo. order:** Increasing $j - i$
- **Base cases:** $S(i, i + 1) = [A[i]]$
- **Original:** $S(0, n)$
- **Time:** $T(n) = 2\,T(n/2) + O(n) = O(n \lg n)$

폭넓은 접근

```c
int co_rank(int k, int* A, int m, int* B, int n)
{
    int i = (k < m) ? k : m; // i = min(k, m);
    int j = k-i;
    int i_low = (0 > (k-n)) ? 0 : k-n; // i_low = max(0, k-n);
    int j_low = (0 > (k-m)) ? 0 : k-m; // j_low = max(0, k-m);
    int delta;
    bool active = true;

    while (active) {
        if (i > 0 && j < n && A[i-1] > B[j]) {
            delta = ((i - i_low + 1) >> 1);
            j_low = j;
            j = j + delta;
            i = i - delta;
        }
        else if (j > 0 && i < m && B[j-1] >= A[i]) {
            delta = ((j - j_low + 1) >> 1);
            i_low = i;
            i = i + delta;
            j = j - delta;
        }
        else {
            active = false;
        }
    }

    return i;
}
```

https://junstar92.tistory.com/268

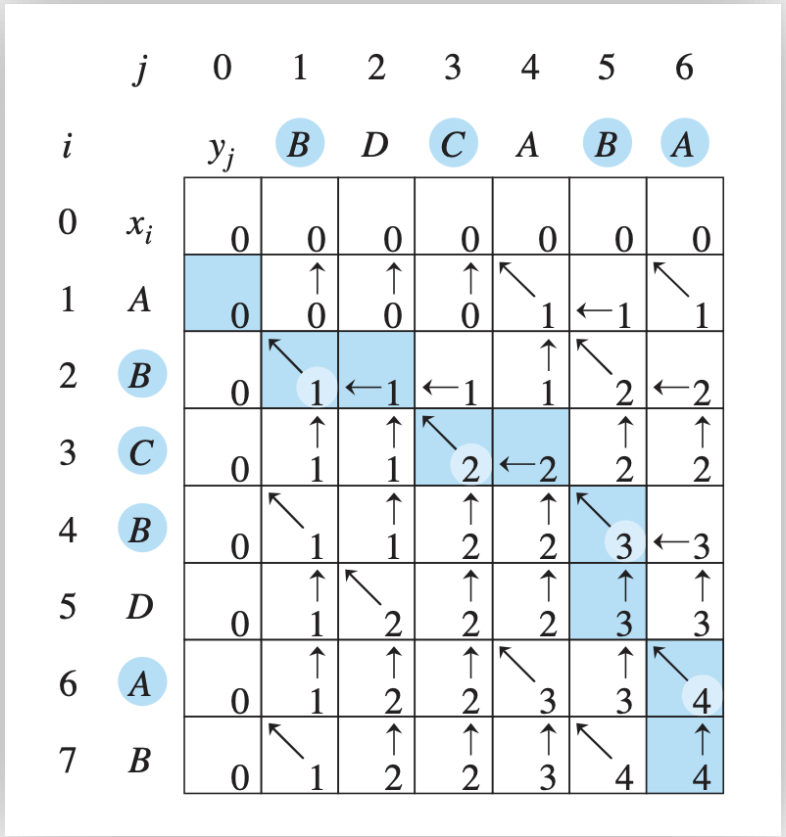
C with CUDA kernel

```c
__global__
void merge_basic_kernel(int* A, int m, int* B, int n, int* C)
{
    int tid = blockDim.x*blockIdx.x + threadIdx.x;
    int k_curr = tid * ceil((m+n)/(float)(blockDim.x*gridDim.x));
    int k_next = min((tid+1) * (int)ceil((m+n)/(float)(blockDim.x*gridDim.x)), m+n);
    int i_curr = co_rank(k_curr, A, m, B, n);
    int i_next = co_rank(k_next, A, m, B, n);
    int j_curr = k_curr - i_curr;
    int j_next = k_next - i_next;

    sequentialMerge(A+i_curr, i_next-i_curr, B+j_curr, j_next-j_curr, C+k_curr);
}
```

3) 재귀를 적용한 사례 및 폭넓은 접근

Parallel DP: LCS 문제에서 테이블을 만들 때, 각 칸(혹은 대각선 방향으로) 독립 계산이 가능하다면, 병렬로 업데이트를 진행할 수도 있음

1. **Subproblems**
   - $x(i, j) =$ length of longest common subsequence of suffixes $A[i:]$ and $B[j:]$
   - For $0 \le i \le |A|$ and $0 \le j \le |B|$
2. **Relate**
   - Either first characters match or they don't
   - If first characters match, some longest common subsequence will use them
   - (if no LCS uses first matched pair, using it will only improve solution)
   - (if an LCS uses first in $A[i]$ and not first in $B[j]$, matching $B[j]$ is also optimal)
   - If they do not match, they cannot both be in a longest common subsequence
   - **Guess** whether $A[i]$ or $B[j]$ is not in LCS
   - $x(i, j) = \begin{cases} x(i+1, j+1) + 1 & \text{if } A[i] = B[j] \\ \max\{x(i+1, j), x(i, j+1)\} & \text{otherwise} \end{cases}$
   - (draw subset of all rectangular grid dependencies)
3. **Topological order**
   - Subproblems $x(i, j)$ depend only on strictly larger i or j or both
   - Simplest order to state: Decreasing $i + j$
   - Nice order for bottom-up code: Decreasing i, then decreasing j
4. **Base**
   - $x(i, |B|) = x(|A|, j) = 0$ (one string is empty)
5. **Original problem**
   - Length of longest common subsequence of A and B is $x(0, 0)$
   - Store parent pointers to reconstruct subsequence
   - If the parent pointer increases both indices, add that character to LCS



3) 재귀를 적용한 사례 및 폭넓은 접근

병렬 업데이트 방식

- **행 단위 계산(Sequential DP):**
일반적으로 $dp[i][j]$는 위에서 아래로, 왼쪽에서 오른쪽으로 순차적으로 계산됩니다.

- **대각선 단위 계산(Parallel DP):**
병렬 처리를 위해 각 대각선의 셀들을 한 번에 계산합니다.
  - 대각선의 식: $i + j = k$
  - 모든 셀이 동일한 대각선에 있다면, **이전 대각선의 값을 활용해 병렬적으로 업데이트**할 수 있습니다.

3. 병렬 DP(Parallel DP) 적용

병렬 처리가 가능한 이유

1. **DP 테이블의 계산 의존성:**
   - 각 셀 $dp[i][j]$는 $dp[i-1][j]$, $dp[i][j-1]$, $dp[i-1][j-1]$ 값에만 의존합니다.
   - 즉, 동일한 대각선(예: $i + j = k$) 상에 있는 값들은 서로 독립적입니다.
   - 따라서, 한 대각선의 값을 병렬적으로 계산할 수 있습니다.
2. **병렬 업데이트 흐름:**
   - 대각선 단위로 테이블을 채우는 방식으로 진행.
   - (i, j)가 있는 셀은 모두 **동일한 대각선(i+j=k)**에 있으며, 대각선 내의 값들은 서로 독립적으로 계산 가능합니다.
   - 병렬 처리를 통해 각 대각선을 동시에 계산하면, 수행 시간을 크게 단축할 수 있습니다.

3) 재귀를 적용한 사례 및 폭넓은 접근

- **머신 러닝**

결정 트리(Decision Tree) 학습 과정

: 결정 트리는 "어떤 기준(속성)을 토대로 데이터를 둘(이상)로 분할", 이후 그 하위 집합에 대해 반복해서 분할하며 트리를 만들어 나가는 과정. 분할 정복으로, 가장 좋은 분할 기준을 찾고, 그 기준으로 분할한 뒤 (SubProblem, Relation), 남은 데이터에 대해 재귀적으로 트리 생성.

+딥 러닝 역전파(Backpropagation)

: DP의 중복 계산 저장(Memoization)을 통해, 레이어마다 도출되는 편미분 값을 저장하고 재활용. 실제로는 반복문 형태가 일반적이지만, 간단히 생각해보기 좋음(재귀로 사용되면 어떨지. Bottom-up을 통해 반복문화).

3) 재귀를 적용한 사례 및 폭넓은 접근

결정 트리(Decision Tree) 학습 과정

- 정보 이득(Information Gain): 데이터가 특정 기준으로 분할되었을 때, **순도가 증가**하거나(다양성이 감소), **불확실성이 감소**되는 정도를 측정하는 값.
- 정보이론에서, 정보 이득은 부모 집합과 자식 집합의 엔트로피(불확실성을 나타내는 척도) 차이를 기반으로 계산됨.

$$Entropy(A) = -\sum_{k=1}^{m} p_k \log_2 (p_k)$$

이 식을 바탕으로 검은색 박스로 둘러쌓인
A 영역의 엔트로피를 구해보겠습니다.
전체 16개(m=16) 중에
빨간색 동그라미(범주=1)는 10개, 파란색(범주=2)은 6개이군요.
그럼 A 영역의 엔트로피는 다음과 같습니다.

$$Entropy(A) = -\frac{10}{16}\log_2 (\frac{10}{16}) - \frac{6}{16}\log_2 (\frac{6}{16}) \approx 0.95$$

여기서 A 영역에 검은 점선을 그어
두 개의 부분집합(R1, R2)으로 분할한다고 가정합니다.
두 개 이상 영역에 대한 엔트로피 공식은 아래 식과 같습니다.
이 공식에 의해 A 영역의 엔트로피를 아래와 같이 각각 구할 수 있습니다.
(Ri=분할 전 레코드 가운데 분할 후 i 영역에 속하는 레코드의 비율)

$$Entropy(A) = \sum_{i=1}^{d} R_i \left(-\sum_{k=1}^{m} p_k \log_2 (p_k) \right)$$

$$Entropy(A) = 0.5 \times \left(-\frac{7}{8}\log_2 (\frac{7}{8}) - \frac{1}{8}\log_2 (\frac{1}{8}) \right) + 0.5 \times \left(-\frac{3}{8}\log_2 (\frac{3}{8}) - \frac{5}{8}\log_2 (\frac{5}{8}) \right) \approx 0.75$$

이때 엔트로피의 감소는 불확실성의 감소를 뜻하며, 곧 순도 증가, 정보 이득을 뜻함!

https://swancity.tistory.com/17

3) 재귀를 적용한 사례 및 폭넓은 접근

```python
import numpy as np
from collections import Counter

class DecisionTree:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.tree = None

    def fit(self, X, y):
        # 재귀적으로 트리 생성
        self.tree = self._build_tree(X, y, depth=0)

    def _build_tree(self, X, y, depth):
        # 종료 조건: 데이터가 모두 같은 클래스이거나, 최대 깊이에 도달한 경우
        if len(set(y)) == 1 or (self.max_depth is not None and depth >= self.max_depth):
            return Counter(y).most_common(1)[0][0]  # 가장 빈도가 높은 클래스를 반환

        # 가장 좋은 분할 기준 찾기
        best_feature, best_threshold = self._find_best_split(X, y)

        # 데이터를 분할
        left_indices = X[:, best_feature] <= best_threshold
        right_indices = X[:, best_feature] > best_threshold

        # 재귀적으로 왼쪽/오른쪽 서브트리 생성
        left_subtree = self._build_tree(X[left_indices], y[left_indices], depth + 1)
        right_subtree = self._build_tree(X[right_indices], y[right_indices], depth + 1)

        # 현재 노드 반환
        return {"feature": best_feature, "threshold": best_threshold,
                "left": left_subtree, "right": right_subtree}
```

```python
def _find_best_split(self, X, y):
    # 간단히 구현: 분할 기준으로 정보 이득(Information Gain)을 사용
    n_features = X.shape[1]
    best_gain = -1
    best_feature, best_threshold = None, None

    for feature in range(n_features):
        thresholds = np.unique(X[:, feature])
        for threshold in thresholds:
            # 데이터를 분할
            left_indices = X[:, feature] <= threshold
            right_indices = X[:, feature] > threshold

            # 정보 이득 계산
            gain = self._information_gain(y, y[left_indices], y[right_indices])
            if gain > best_gain:
                best_gain, best_feature, best_threshold = gain, feature, threshold

    return best_feature, best_threshold

def _information_gain(self, parent, left_child, right_child):
    # 엔트로피 기반 정보 이득 계산
    def entropy(labels):
        counts = np.bincount(labels)
        probabilities = counts / len(labels)
        return -np.sum([p * np.log2(p) for p in probabilities if p > 0])

    parent_entropy = entropy(parent)
    n = len(parent)
    left_entropy = entropy(left_child)
    right_entropy = entropy(right_child)
    return parent_entropy - (len(left_child) / n * left_entropy + len(right_child) / n * right_entropy)

def predict(self, X):
    return np.array([self._traverse_tree(x, self.tree) for x in X])

def _traverse_tree(self, x, tree):
    if not isinstance(tree, dict):
        return tree
    if x[tree["feature"]] <= tree["threshold"]:
        return self._traverse_tree(x, tree["left"])
    else:
        return self._traverse_tree(x, tree["right"])
```

3) 재귀를 적용한 사례 및 폭넓은 접근

1. 분할 기준 선택

결정 트리에서 각 노드는 데이터를 **두 그룹(왼쪽, 오른쪽)**으로 나누는 기준을 찾아야 합니다.
이때 가장 좋은 기준은 **정보 이득(Information Gain)**이 가장 큰 기준입니다.

$$(f, t) = \arg\max_{f,t} \text{InformationGain}(f, t)$$

- f: **특징(feature)**
데이터를 나누는 데 사용할 열(column).
- t: **임계값(threshold)**
데이터를 두 그룹으로 나누는 기준 값.

정보 이득(Information Gain) 계산:
데이터를 분할한 후, 얼마나 **불확실성이 감소**했는지를 나타냅니다.

$$\text{InformationGain}(f, t) = H(y) - \frac{|L|}{|y|} H(L) - \frac{|R|}{|y|} H(R)$$

- $H(y)$: 현재 노드에서 데이터의 **엔트로피(불확실성)**
$$H(y) = -\sum_{c \in C} p(c) \log_2 p(c)$$
$p(c)$: 클래스 c의 비율.
- L, R: 데이터를 왼쪽 그룹과 오른쪽 그룹으로 나눕니다.
- $|L|, |R|$: 각 그룹의 데이터 개수.
- **의미**:
  - 부모 노드(전체 데이터)의 불확실성에서, 분할된 데이터(왼쪽, 오른쪽)의 가중합 불확실성을 뺀 값입니다.
  - 정보 이득이 크면 데이터를 잘 분할한 것입니다.

2. 재귀적으로 트리 생성

한 번 데이터를 나눈 후, **왼쪽과 오른쪽 그룹**에 대해 같은 과정을 반복합니다.
이 과정을 **재귀적으로** 적용하여 트리를 만듭니다.

$$T(f, t) = \begin{cases} \text{Leaf Node}, & \text{if stopping condition is met} \\ \{f, t, T(L), T(R)\}, & \text{otherwise} \end{cases}$$

- **Leaf Node**: 더 이상 나눌 필요가 없을 때 트리의 끝(잎 노드)입니다.
- **Recursive Call**: 데이터를 L과 R로 나누고, 각 데이터셋에서 트리를 다시 생성합니다.

3. 종료 조건

분할을 멈추는 조건:

1. **데이터가 모두 같은 클래스일 때**
$H(y) = 0$
   - 예: 데이터가 모두 yes라면 더 이상 나눌 필요가 없습니다.
2. **트리의 최대 깊이에 도달했을 때**
$\text{Depth} \geq \text{MaxDepth}$
   - 예: 트리의 깊이를 5로 제한한 경우, 더 이상 분할하지 않습니다.

3) 재귀를 적용한 사례 및 폭넓은 접근

- 양자 알고리즘

- 완전히 고전적 재귀가 양자적 재귀는 아니지만, 큰 문제를 쪼갠 뒤(주파수, 주기 등) 하위 문제를 해결하고 결합한다는 사고가 비슷함.

Shor 알고리즘(소인수분해

Grover 알고리즘(검색)

5.1 Shor 알고리즘(소인수분해)

- **아이디어**:
  1. 특정 함수 $f(x) = a^x \bmod N$의 **주기(period)**를 찾으면, 그걸 바탕으로 N의 소인수를 구할 수 있음.
  2. "주기 찾기"에 **양자 푸리에 변환(QFT)** 사용.
  3. **QFT** 자체가 고전적으로는 **FFT(Fast Fourier Transform)** 방식으로 구현 가능(재귀 구조).
- 즉, **분할정복(FFT) + 양자 중첩**으로 "주기"를 효율적으로 찾아내는 것.

5.2 Grover 알고리즘(데이터베이스 검색)

- 고전적으로 O(N) 걸리는 탐색을, **양자 상태**를 이용하면 O(√N)에 가능.
- 단계별 반복(Iteration) 과정에서 **하위 상태**의 진폭을 갱신해 "정답 상태"를 더 두드러지게 만듦.
- 이를 재귀적으로 표현하기도 하는데, 고전적인 분할정복과는 다소 다르지만, "상태를 계속 업데이트해나간다"는 점에서 재귀적 시뮬레이션이 가능.

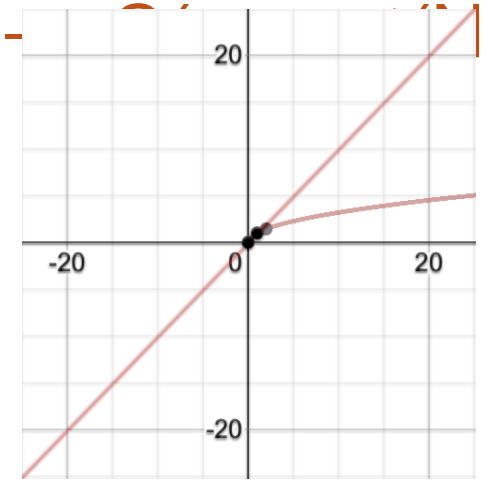
3) 재귀를 적용한 사례 및 폭넓은 접근

Grover 알고리즘(검색)

정답 상태의 진폭이 충분히 결정되었을 때, 탐색을 종료(Base Case)

재귀 호출은 Oracle과 [illegible] 진행(Recursive Case)

O(N) - [illegible]



2-1. 문제: 데이터베이스 검색

- 데이터베이스에서 특정 데이터를 찾는 문제입니다.
  - 예: 100개의 데이터 중에서 원하는 값을 찾기.

2-2. 고전 알고리즘의 한계

- 고전적인 탐색 알고리즘은 **모든 데이터를 하나씩 확인**해야 하므로, 평균적으로 $O(N)$ 시간이 걸립니다.

2-3. Grover 알고리즘의 핵심 아이디어

Grover 알고리즘은 **양자 상태를 이용해 데이터베이스를 더 빠르게 검색**합니다.

1. **양자 중첩으로 모든 데이터를 동시에 확인**
   - 고전적인 컴퓨터는 데이터를 하나씩 확인하지만, 양자 컴퓨터는 중첩(superposition)을 이용해 **모든 데이터를 동시에 탐색**할 수 있습니다.
2. **올바른 답을 증폭(증명)**
   - Grover 알고리즘은 **올바른 답이 포함된 상태를 반복적으로 증폭(강조)**하여, 최종적으로 올바른 답을 빠르게 찾습니다.
   - 반복 횟수는 $O(\sqrt{N})$으로, 고전적인 탐색보다 훨씬 빠릅니다.

발표를 마무리하며...

- 재귀 자체는 “코드를 자기 자신이 다시 호출한다”는 형태적인 특징.
- 어떤 논리(로직)와 사고방식을 적용하느냐에 따라 다양한 재귀 기법으로 나눠짐.

결론적으로 하나의 단순한 재귀 코드만으로도, 우리가 무엇을 중점적으로 생각하는지에 따라 전혀 다른 기법이 되며, 이 재귀적 사고가 개발자로서 상상력과 활용 아이디어에 큰 도움이 될 것.

감사합니다

CAS