알고리즘	시간 복잡도
Computing Binominal Coefficients	$\Theta(nk)$
Path Counting Problem	$\Theta(nm)$
Permutations and Combinations	$O(m)$
Coin-Collecting Problem	$\Theta(nm)$
Chained Matrix Multiplication(minimum)	$\Theta(n^3)$
Chained Matrix Multiplication(optimal)	$\Theta(n)$
Chained Matrix Order	$\Theta(n)$
Optimal Binary Search Tree	$\Theta(n^3)$
Optimal Binary Search Tree(reculsive)	$\Theta(n)$
multi matrix	$O(n^4)$
Floyd	$\Theta(n^3)$
Floyd shortest path	$\Theta(n)$
Coin Change	$\Theta(nm)$
Hungarian	O(n^3)(최악, 최적)
dpKnapsack	$\Theta(nW)$
backtrack(knapsack)	$O(2^n)$최악 $O(2^{n/2})$
Back Tracking Algorithm for Traveling salesman problem	$\Theta(n^2 2^n)$
Back Tracking Promising	$\frac{(n-1)^n - 1}{n-2}$

Week9

1. Computing Binominal Coefficients

: 이항 계수 계산에는 파스칼의 삼각형 또는 이항 공식을 사용하여 집합에서 항목을 선택하는 방법의 수를 결정하는 작업이 포함됩니다. 파스칼의 삼각형은 시각적 표현과 룩업 표를 제공하는 반면, 이항 공식은 직접 계산을 위한 수학적 표현을 제공합니다.

- Dynamic Programming Aprroach

큰 문제를 작은문제로 나누어 푸는 문제를 일컫는 말입니다. 동적 계획법이란 말 때문에 어떤 부분에서 동적으로 프로그래밍이 이루어지는 찾아볼 필요가 없습니다. DP의 경우 작은 부분 문제의 답이 항상 같아야 한다.

방법 : 모든 작은 문제들은 한번만 풀어야 합니다. 따라서 정답을 구한 작은 문제를 어딘가에 메모해 놓습니다. 다시 그보다 큰 문제를 풀어나갈 때 똑같은 작은 문제가 나타나면 앞서 메모한 작은 문제의 결과 값을 이용합니다.

조건 : - 작은 문제가 반복이 일어나는 경우.
- 같은 문제는 구할 때마다 정답이 같다.

위와 같은 조건을 만족하는 경우에만 동적프로그래밍을 사용할 수 있습니다. 작은 문제의 결과 값이 항상 같다는 점을 이용해서 큰문제를 해결하는 방법이니 당연합니다.

Copute Binominal Coefficient

Algorithm

```
1:  procedure BC( n, k)
2:      integer i, j
3:      integer B[0..n][0..k]
4:      for (i = 0; i <= n; i++) do
5:          for (j = 0; j <= min(i, k); j++) do
6:              if (j == 0 || j == i) then
7:                  B[i][j] = 1
8:              else
9:                  B[i[j] = B[i − 1][j − 1] + B[i − 1][j]
10:             end if
11:         end for
12:     end for
13: end procedure
```

2. Path Counting Problem

: 경로 계산 문제는 특정 제약 조건 또는 규칙에 따라 주어진 그래프 또는 그리드에서 시작점에서 끝점까지 가능한 경로 또는 경로의 수를 결정하는 것을 포함합니다. 이러한 유형의 문제들은 종종 조합론, 그래프 이론, 그리고 컴퓨터 과학에서 발생합니다.

Path Counting Problem

- Dynamic programming based algorithm

```
1:  procedure COUNTPATHDP(n,m)
2:      T[n][m]
3:      for (int i = 0; i < n; i++) do
4:          T[i][0] = 1
5:      end for
6:      for (int j = 0; j < m; i++) do
7:          T[0][j] = 1
8:      end for
9:      for (int i = 1; i < n; i++) do
10:         for (int j = 1; j < m; i++) do
11:             T[i][j] = T[i-1][j] + T[i][j-1]
12:         end for
13:     end for
14: end procedure
```

Permutations and Combinations

```
1: procedure COUNTPATHCMN(n,m) paths=1
2:     for (i=n; i< m+n-1; i++) do
3:         paths = paths × i
4:         paths = paths/i
5:     end for
6:     return paths
7: end procedure
```

나이트 투어:

기사의 여행은 체스에서 전형적인 경로 계산 문제입니다. 체스판과 기사가 어떤 사각형 위에 놓였을 때, 목표는 기사가 각 사각형을 정확히 한 번 방문할 수 있도록 하는 일련의 움직임을 찾는 것입니다.

기사의 여행 문제를 해결하는 한 가지 방법은 역추적을 사용하는 것입니다. 사각형 위에 있는 기사부터 시작해서 그 위치에서 가능한 모든 동작을 시도합니다. 이동이 유효하고 방문하지 않은 사각형으로 이어지는 경우 해당 이동을 수행하고 반복적으로 계속합니다. 모든 사각형이 방문된 경우 경로 카운트를 증분합니다. 백트래킹을 사용하면 솔루션을 찾거나 모든 가능성을 소진할 때까지 다양한 경로를 탐색할 수 있습니다.

3. Coin-collecting Problem

몇 개의 동전이 n 보드의 셀에 놓여있습니다. 셀당 하나의 동전보다 많지 않습니다. 보드의 왼쪽 상단 셀에 위치한 로봇은 가능한 많은 동전을 모아서 가져와야 합니다.

Coin-collecting Problem

```
1: procedure RobotCoinCollection(C[1...n, 1...m])
2:     F[1,1] = C[1,1]
3:     for (j = 2; j <= m; j++) do
4:         F[1,j] = F[1,j-1] + C[1,j]
5:     end for
6:     for (i = 2; <= n; i++) do
7:         F[i,1] = F[i,1] + C[i,1]
8:         for (j = 2; j <= m; j++) do
9:             F[i,j] =
   max{F[i-1,j] + C[i,j], F[i,j-1] + C[i,j]}
10:        end for
11:    end for
12: end procedure
```

4. Chained Matrix Multiplication

행렬 체인 곱셈이라고도 하는 연쇄 행렬 곱셈은 필요한 스칼라 곱셈의 수를 최소화하기 위해 여러 행렬을 특정 순서로 함께 곱하는 과정을 말합니다. 이 문제는 최적화, 컴퓨터 그래픽 및 알고리즘 설계와 같은 다양한 응용 분야에서 발생합니다.

행렬 체인 곱셈 알고리듬은 문제의 최적 하위 구조 속성을 활용하며, 여기서 더 큰 문제에 대한 최적의 솔루션을 하위 문제에 대한 최적의 솔루션으로 구성할 수 있습니다. 동적 프로그래밍을 통해 알고리듬은 다양한 하위 문제에 필요한 최소 스칼라 곱셈 수를 효율적으로 계산하여 전체 문제에 대한 최적의 솔루션으로 이어집니다.

중첩 루프로 인해 알고리즘의 시간 복잡도는 $O(n^3)$입니다. 여기서 n은 시퀀스의 행렬 수입니다. 이를 통해 비교적 작은 규모에서 중간 규모의 문제 인스턴스에 대해 계산적으로 효율적인 접근 방식을 제공합니다.

Print Optimal Order

Pseudo-code

```
1: procedure ORDER(integer i, integer j)
2:     if (i == j) then
3:         print(A, i)
4:     else
5:         k = P[i][j]
6:         print(()
7:         order(i, k)
8:         order(k + 1, j)
9:         print( ))
10:    end if
11: end procedure
```

5. Optimal Binary Search Tree

최적 이진 검색 트리(최적 BST 또는 최적 검색 트리라고도 함)는 지정된 키 집합에 대한 평균 검색 시간을 최소화하는 이진 검색 트리입니다. 이진 검색 트리에서 각 노드에는 키가 있으며 키는 효율적인 검색, 삽입 및 삭제 작업을 수행할 수 있도록 구성되어 있습니다.

최적의 이진 검색 트리에서 키는 예상 검색 비용을 최소화하는 방식으로 배열됩니다. 검색 비용은 키 깊이의 가중 합계로 정의되며, 여기서 가중치는 각 키에 액세스하는 빈도를 나타냅니다. 목표는 가장 낮은 예상 검색 비용을 산출하는 이진 검색 트리를 구성하는 것입니다.

* 코드 시험에 나옴

Optimal Binary Search Tree (REC)

```
1:  procedure TREE( R[][], i, j)
2:      Integer k = R[i][j]
3:      node-pointer p
4:      if (k == 0) then
5:          return null
6:      else
7:          p = new  nodetype
8:          p-> key = Key[k]
9:          p-> left = tree(R[][], i, k − 1)
10:         p-> right = tree(R[][], k + 1, j)
11:         return p
12:     end if
13: end procedure
```

재귀 속성은 자체 참조 또는 자체 반복을 포함하는 함수, 알고리즘 또는 구조의 특성을 나타냅니다. 다시 말해서, 그것은 그 자체로 어떤 것을 정의하는 속성입니다.

재귀 알고리즘 또는 함수에서, 문제에 대한 해결책은 동일한 문제의 작은 인스턴스를 해결함으로써 얻어집니다. 함수 또는 알고리즘은 직접 솔루션이 알려진 기본 사례에 도달할 때까지 종종 수정된 입력을 사용하여 반복적으로 자체 호출합니다. 재귀적 특성은 복잡한 문제를 더 간단한 하위 문제로 분해할 수 있게 하며, 이는 독립적으로 해결된 다음 결합하여 최종 해결책을 얻을 수 있습니다.

재귀적 특성은 일반적으로 컴퓨터 과학과 수학의 다양한 분야에서 발견됩니다:

1. 재귀 함수: 자신의 정의 내에서 스스로를 부르는 기능. 각 재귀 호출은 직접 솔루션이 반환되는 기본 사례에 도달할 때까지 문제 크기를 줄입니다. 예를 들어 요인 함수, 피보나치 시퀀스 계산 및 퀵 정렬 또는 병합 정렬과 같은 재귀 정렬 알고리즘이 있습니다.

2. 재귀적 데이터 구조: 동일한 데이터 구조 유형의 인스턴스에 대한 참조를 포함하는 데이터 구조입니다. 예를 들어 링크된 목록, 이진 트리 및 그래프가 있습니다. 재귀 속성을 사용하면 데이터 구조를 계층적으로 표현하고 통과할 수 있습니다.

3. 재귀적 정의: 자체 정의에서 자신을 참조하는 수학적 객체 또는 구조의 정의입니다. 예를 들어 음이 아닌 정수 n의 요인은 n! = n * (n-1)!로 재귀적으로 정의할 수 있습니다. 여기서 (n-1)!은 동일한 문제의 작은 인스턴스입니다.

4. 재귀적 문제 해결 기술: 문제를 더 작은 하위 문제로 나누고 이를 재귀적으로 해결하는 기술입니다. 여기에는 분할 및 정복 알고리즘, 동적 프로그래밍, 역추적 등이 포함됩니다.

재귀 속성은 반복적이거나 자기 참조적인 특성을 나타내는 문제에 대한 우아하고 간결한 해결책을 제공할 수 있습니다. 그러나 종료를 보장하고 무한 재귀를 방지하기 위해 재귀 알고리즘과 구조를 신중하게 설계하는 것이 필수적입니다.

동적 프로그래밍(DP):
동적 프로그래밍의 재귀적 속성은 문제를 중복된 하위 문제로 나누고 이러한 하위 문제의 결과를 테이블(일반적으로 메모화 테이블 또는 동적 프로그래밍 테이블이라고 함)에 저장함으로써 문제를 해결하는 것을 포함합니다. 그런 다음 더 큰 문제에 대한 솔루션이 더 작은 하위 문제에 대한 솔루션에서 구축되어 중복 계산을 제거합니다. 동적 프로그래밍의 재귀 속성은 반복 계산을 피함으로써 효율적인 계산을 가능하게 합니다.

역추적:
역추적은 솔루션을 점진적으로 구축한 다음 잘못된 선택이 발견되면 취소(역추적)하여 문제에 대한 가능한 모든 해결책을 탐색하는 일반적인 알고리즘 기술입니다. 역추적의 재귀적 속성은 문제 공간에 대한 트리와 같은 탐색을 통한 솔루션의 재귀적 검색에 있습니다. 각 단계에서 알고리즘은 선택을 하고, 나머지 선택지를 재귀적으로 탐색하며, 막다른 골목에 이르게 되면 선택지를 철회합니다. 역추적의 반복 호출은 유효한 솔루션을 찾거나 모든 가능성이 소진될 때까지 가능한 모든 경로를 탐색하는 데 도움이 됩니다.

분기 및 경계:
분기 및 경계는 문제를 더 작은 하위 문제로 나누고 지금까지 발견된 최상의 솔루션을 추적하여 최적화 문제를 해결하기 위한 알고리즘 기술입니다. 분기 및 경계의 재귀 속성은 서로 다른 하위 문제로 분기하여 검색 공간을 재귀적으로 탐색하는 데 있습니다. 각 하위 문제는 평가되고 경계 또는 휴리스틱을 기반으로 일부 하위 문제는 제거되는(즉, 더 이상 탐구할 가치가 없는 것으로 간주됨) 반면 다른 하위 문제는 재귀적으로 탐구됩니다. 이 기술은 종종 가장 유망한 하위 문제의 우선순위를 매기기 위해 우선순위 대기열 또는 유사한 데이터 구조를 사용합니다.

그리디 알고리즘:
탐욕스러운 알고리즘은 글로벌 최적을 찾기 위해 각 단계에서 지역적으로 최적의 선택을 합니다. 탐욕스러운 알고리즘이 모든 문제에 대해 항상 최적의 해결책을 제공하는 것은 아니지만, 한 단계에서 이루어진 선택이 후속 단계에서 이루어진 선택에 영향을 미친다는 점에서 종종 재귀적 특성을 가집니다. 이러한 재귀적 특성은 각 결정이 현재 상태와 해당 상태의 로컬 최적 선택에 기초하여 이루어진다는 사실에서 발생합니다. 그러나 동적 프로그래밍 또는 역추적과 달리 탐욕 알고리즘의 재귀 속성은 가능한 모든 솔루션을 명시적으로 탐색하거나 전체 검색 트리를 유지하는 것을 포함하지 않습니다. 대신 현재 사용 가능한 정보를 기반으로 각 단계에서 최선의 선택을 하는 데 중점을 둡니다.