**학습활동보고서 # 3**

|  |  |  |  |
| --- | --- | --- | --- |
| 대학/학부/학과 | 엘텍공과대학 | 전공 | 컴퓨터공학과 |
| 학번 | 1871001 | 이름 | ZHU ZHAOLING |

|  |  |
| --- | --- |
| 학습활동 주제 및 목표 | 동적 계획법 |

(필요한 항목 및 내용을 자유로운 양식으로 작성하세요)

1. 학습활동 –내용을 상세히 작성합니다.

**3.0 동적계획법**

**▪ 동적계획법으로 문제 풀기**

1. 문제를 해결할 수 있는 재귀 관계식을 구한다.

2. 가장 작은 입력사례로부터 상향식 방법으로 문제를 해결한다.

**▪ 분할정복법 .vs. 동적계획법**

• 문제를 작은 사례로 분할하여 해결한다는 점에서 동일

• 분할정복: 재귀 호출을 통해 분할하여 정복(Top-Down)

• 동적계획: 메모이제이션을 통해 상향식으로 정복(Bottom-Up)

• ex) 피보나치 수열

**3.1 이항 계수**

**▪ 이항 계수 문제**

• 이항 계수의 정의

⁃ ,for 0≤k≤n.

⁃ 문제점: n!, k!의 값은 매우 크기 때문에 계산이 어렵다.

• 이항 계수의 재귀적 정의: 분할정복(Divide-and-Conquer)

⁃

**Algorithm 3.1: Binomial Coefficient (Divide-and-Conquer)**

1. **def** bin (n, k):
2. **if** (k == 0 **or** n == k):
3. **return** 1
4. **else**:
5. **return** bin(n - 1, k - 1) + bin(n - 1, k)
7. **for** n **in** range(10):
8. **for** k **in** range(n + 1):
9. **print**(bin(n, k), end = " ")
10. **print**()

**▪ 이항 계수의 성질: 파스칼의 삼각형**

**▪ 이항 계수 구하기: 동적계획(Dynamic Programming)**

• 1단계: 재귀 관계식을 찾는다.

⁃ 이항 계수의 재귀적 정의를 찾았다.

⁃

• 2단계: 상향식 방법으로 해답을 구한다.

⁃ 파스칼의 삼각형이 가진 특성을 이용한다.

⁃

⁃

**Algorithm 3.2: Binomial Coefficient (Dynamic Programming)**

1. **def** bin2 (n, k):
2. B = [[0] \* (k + 1) **for** \_ **in** range(n + 1)]
3. **for** i **in** range(n + 1):
4. **for** j **in** range(min(i, k) + 1):
5. **if** (j == 0 **or** j == i):
6. B[i][j] = 1
7. **else**:
8. B[i][j] = B[i - 1][j - 1] + B[i - 1][j]
9. **return** B[n][k]
11. **for** n **in** range(10):
12. **for** k **in** range(n + 1):
13. **print**(bin2(n, k), end = " ")
14. **print**()

▪ 연습문제 3.4: 효율적인 이항계수 계산

• 다음 성질을 이용하면 성능을 더 개선할 수 있다.

⁃ : k가 n/2보다 클 경우에 적용

• 2차원 리스트(배열)를 사용할 필요가 있는가?

⁃ 1차원 리스트(배열)만으로도 구현이 가능하다.

1. **def** bin3 (n, k):
2. **if** (k > n // 2):
3. k = n - k
4. B = [0] \* (k + 1)
5. B[0] = 1
6. **for** i **in** range(1, n + 1):
7. j = min(i, k)
8. **while** (j > 0):
9. B[j] = B[j] + B[j - 1]
10. j -= 1
11. **return** B[k]
13. **for** n **in** range(10):
14. **for** k **in** range(n + 1):
15. **print**(bin3(n, k), end=" ")
16. **print**()
17. **print**(bin3(9, 5))

**3.2 최단 경로 (플로이드 알고리즘)**

▪ 최단 경로: 동적계획(Dynamic Programming)

• 1단계: 재귀 관계식을 찾는다.

⁃ D: 각 정점의 쌍이 가지는 최단 경로의 길이를 나타내는 행렬

⁃ D[i][j]: 에서 로 가는 최단 경로의 길이

⁃ 목표: 인접 행렬 W에서 최단 경로의 행렬 D와의 재귀 관계식 구하기

• 2단계: 상향식 방법으로 해답을 구한다.

⁃ 초기화: ,

⁃ 최종 목표: .

⁃ 상향식 계산:

**Algorithm 3.3: Floyd's Algorithm for Shortest Paths**

1. **def** floyd (W):
2. D = W
3. n = len(W)
4. **for** k **in** range(n):
5. **for** i **in** range(n):
6. **for** j **in** range(n):
7. D[i][j] = min(D[i][j], D[i][k] + D[k][j])
8. **return** D
10. INF = 999
11. W = [
12. [0, 1, INF, 1, 5],
13. [9, 0, 3, 2, INF],
14. [INF, INF, 0, 4, INF],
15. [INF, INF, 2, 0, 3],
16. [3, INF, INF, INF, 0]
17. ]
18. D = floyd(W)
19. **for** i **in** range(len(D)):
20. **print**(D[i])

**Algorithm 3.4: Floyd's Algorithm for Shortest Paths 2**

1. **def** floyd2 (W):
2. n = len(W)
3. D = W
4. P = [[-1] \* n **for** \_ **in** range(n)]
5. **for** k **in** range(n):
6. **for** i **in** range(n):
7. **for** j **in** range(n):
8. **if** (D[i][j] > D[i][k] + D[k][j]):
9. D[i][j] = D[i][k] + D[k][j]
10. P[i][j] = k
11. **return** D, P

**Algorithm 3.5: Print Shortest Path**

1. **def** path (P, u, v):
2. **if** (P[u][v] != -1):
3. path(P, u, P[u][v])
4. **print**('v%d'%(P[u][v]), end='-> ')
5. path(P, P[u][v], v)
6. D, P = floyd2(W)
7. **for** i **in** range(len(D)):
8. **print**(D[i])
9. **for** i **in** range(len(P)):
10. **print**(P[i])
12. u = 4
13. v = 2
14. **print**('shortest path from v%d to v%d:'%(u, v), end=' ')
15. **print**('v%d'%(u), end='-> ')
16. path(P, u, v)
17. **print**('v%d'%(v), end=' ')

**3.4 연쇄 행렬 곱셈**

**▪ 연쇄 행렬 곱셈: 단순무식하게 풀기(Brute-Force Approach)**

• 모든 경우의 수에 대해서 계산해 보고 최적의 순서를 선택

• 연쇄 행렬 곱셈에서 가능한 경우의 수는?

⁃ 카탈란 수:

⁃ 연쇄 행렬 곱셈이 가지는 경우의 수 =

• 𝑛개의 항에 괄호를 씌우는 모든 경우의 수

**▪ 연쇄 행렬 곱셈: 동적계획(Dynamic Programming)**

• 1단계: 재귀 관계식을 찾는다.

⁃ : 연쇄 행렬을 곱하는데 필요한 곱셈의 최소 회수 행렬

⁃ 행렬을 곱하는데 필요한 곱셈의 최소 회수

⁃ 목표:

• 2단계: 상향식 방법으로 해답을 구한다.

⁃ 초기화: (주대각선을 0으로)

⁃ 최종 목표:.

⁃ 상향식 계산: 대각선 1번, 대각선 2번, ⋯ , 대각선 − 1번

**Algorithm 3.6: Chained Matrix Multiplication**

1. **def** minmult (d):
2. n = len(d) - 1
3. M = [[-1] \* (n + 1) **for** \_ **in** range(n + 1)]
4. P = [[-1] \* (n + 1) **for** \_ **in** range(n + 1)]
5. **for** i **in** range(1, n + 1):
6. M[i][i] = 0
7. **for** diagonal **in** range(1, n):
8. **for** i **in** range(1, n - diagonal + 1):
9. j = i + diagonal
10. M[i][j], P[i][j] = minimum(M, d, i, j)
11. **return** M, P
13. **def** minimum (M, d, i, j):
14. minValue = INF
15. minK = 0
16. **for** k **in** range(i, j):
17. value = M[i][k] + M[k + 1][j]
18. value += d[i - 1] \* d[k] \* d[j]
19. **if** (minValue > value):
20. minValue = value
21. minK = k
22. **return** minValue, minK

**Algorithm 3.7: Print Optimal Order**

1. **def** order (P, i, j):
2. **if** (i == j):
3. **print**('A%d'%(i), end='')
4. **else**:
5. k = P[i][j]
6. **print**('(', end ='')
7. order(P, i, k)
8. order(P, k + 1, j)
9. **print**(')', end ='')
11. INF = 999
12. d = [5, 2, 3, 4, 6, 7, 8]
13. M, P = minmult(d)
14. **print**('M = ')
15. **for** i **in** range(1, len(M)):
16. **print**(M[i][1:])
17. **print**('P = ')
18. **for** i **in** range(1, len(P)):
19. **print**(P[i][1:])
21. **print**('minimum order: ', end ='')
22. order(P, 1, len(d) - 1)

**3.5 최적 이진검색트리**

**▪ 최적 이진검색트리: 단순무식하게 풀기(Brute-Force Approach)**

• 모든 경우의 수에 대해서 계산해 보고 최적의 이진트리 선택

• 이진검색트리의 가능한 경우의 수는?

⁃ 카탈란 수:

⁃ 의 키로 만들 수 있는 이진 트리의 수

**▪ 최적 이진검색트리: 동적계획(Dynamic Programming)**

• 1단계: 재귀 관계식을 찾는다.

⁃ 이진검색트리를 만드는데 최적 검색비용의 행렬

⁃ 이진검색트리를 만드는데 최적 검색 비용

⁃ 목표: 로 분할하는 재귀 관계식 찾기

• 2단계: 상향식 방법으로 해답을 구한다.

⁃ 초기화:

⁃ 최종 목표:.

⁃ 상향식 계산: 대각선 1번, 대각선 2번, ⋯ , 대각선 번

**Algorithm 3.9: Optimal Binary Search Tree**

1. **def** optsearchtree (p):
2. n = len(p) - 1
3. A = [[-1] \* (n + 1) **for** \_ **in** range(n + 2)]
4. R = [[-1] \* (n + 1) **for** \_ **in** range(n + 2)]
5. **for** i **in** range(1, n + 1):
6. A[i][i - 1] = 0
7. A[i][i] = p[i]
8. R[i][i - 1] = 0
9. R[i][i] = i
10. A[n + 1][n] = 0
11. R[n + 1][n] = 0
12. **for** diagonal **in** range(1, n):
13. **for** i **in** range(1, n - diagonal + 1):
14. j = i + diagonal
15. A[i][j], R[i][j] = minimum(A, p, i, j)
16. **return** A, R
18. **def** minimum (A, p, i, j):
19. minValue = INF
20. minK = 0
21. **for** k **in** range(i, j + 1):
22. value = A[i][k - 1] + A[k + 1][j]
23. **for** m **in** range(i, j + 1):
24. value += p[m]
25. **if** (minValue > value):
26. minValue = value
27. minK = k
28. **return** minValue, minK
30. INF = 999
31. keys = [0, 10, 20, 30, 40, 50]
32. p = [0, 35, 12, 22, 8, 15]
33. A, R = optsearchtree(p)
34. **print**('A = ')
35. **for** i **in** range(1, len(A)):
36. **print**(A[i])
37. **print**('R = ')
38. **for** i **in** range(1, len(R)):
39. **print**(R[i])

**Algorithm 3.10: Build Optimal Binary Search Tree**

1. **def** tree (R, i, j):
2. k = R[i][j]
3. **if** (k == 0):
4. **return** None
5. **else**:
6. node = BSTNode(keys[k])
7. node.left = tree(R, i, k - 1)
8. node.right = tree(R, k + 1, j)
9. **return** node
11. **class** BSTNode:
12. **def** \_\_init\_\_ (self, key):
13. self.key = key
14. self.left = None
15. self.right = None
16. **def** preorder (node):
17. **if** (node **is** None):
18. **return**
19. **else**:
20. **print**(node.key, end = ' ')
21. preorder(node.left)
22. preorder(node.right)
23. **def** inorder (node):
24. **if** (node **is** None):
25. **return**
26. **else**:
27. inorder(node.left)
28. **print**(node.key, end = ' ')
29. inorder(node.right)
31. root = tree(R, 1, len(p) - 1)
32. **print**('inorder: ', end = ' ')
33. inorder(root)
34. **print**('\npreorder: ', end = ' ')
35. preorder(root)
36. 느낀 점 - 이번 학습활동으로 배운 점 혹은 시행착오를 분석한 후, 다음 학습활동 반영합니다.

동적계획법과 분할정복법의 가장 큰 차이점은 동적계획법으로 해결하는 문제에 적합하고 분해 후 얻어지는 하위 문제들이 서로 독립적이지 않은 경우가 많다는 점이다(즉, 다음 하위 단계의 솔루션은 이전 하위 단계의 솔루션을 기반으로 하고 추가 솔루션이 수행되다.).