**학습활동보고서 # 2**

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

|  |  |
| --- | --- |
| 학습활동 주제 및 목표 | 분할 정복법 |

(필요한 항목 및 내용을 자유로운 양식으로 작성하세요)

1. 학습활동 –내용을 상세히 작성합니다.

**2.1 이분 검색**

**▪ 이분 검색 문제**

• 정렬되지 않은 리스트에서 주어진 키가 존재하는가?

⁃ 순차 탐색

• 정렬된 리스트에서 주어진 키가 존재하는가?

⁃ 이분 검색

• Algorithm 1.5: 반복적인 방법으로 이분 검색

• Algorithm 2.1: 재귀적인 방법으로 이분 검색

**▪ 이분 검색: 분할정복(Divide-and-Conquer)**

• 문제: 정렬된 리스트 S에 어떤 키 x가 존재하는가?

• 해답: 존재하면 S에서 x의 위치, 아니면 0을 리턴

• 알고리즘: 분할정복

⁃ S의 정가운데 원소와 x를 비교하여 같으면 해당 위치를 리턴, 아니면:

⁃ [Divide] 정가운데 원소를 기준으로 S를 두 개의 리스트로 분할

⁃ [Conquer] x가 정가운데 원소보다 크면 오른쪽, 작으면 왼쪽을 재귀 호출

⁃ [Obtain] 선택한 리스트에서 얻은 답을 리턴

**Algorithm 2.1: Binary Search (Recursive)**

1. **def** location (S, low, high):
2. **if** (low > high):
3. **return** 0
4. **else**:
5. mid = (low + high) // 2
6. **if** (x == S[mid]):
7. **return** mid
8. **elif** (x < S[mid]):
9. **return** location(S, low, mid - 1)
10. **else**:
11. **return** location(S, mid + 1, high)
12. S = [-1, 10, 12, 13, 14, 18, 20, 25, 27, 30, 35, 40, 45]
13. x = 18
14. loc = location(S, 1, len(S) - 1)
15. **print**('S =', S)
16. **print**('x =', x)
17. **print**('loc = ', loc)

**2.2 합병 정렬**

**▪ 합병 정렬: 분할정복(Divide-and-Conquer)**

• [Divide]

⁃ 원소가 n개인 S를 n/2개의 원소를 가진 두 개의 리스트로 분할

• [Conquer]

⁃ 왼쪽의 리스트와 오른쪽의 리스트를 각각 재귀적으로 합병 정렬

• [Combine]

⁃ 각각 정렬된 두 개의 리스트를 정렬된 하나의 리스트로 합병하여 리턴

**Algorithm 2.2: Merge Sort**

1. **def** mergesort (S):
2. n = len(S)
3. **if** (n <= 1):
4. **return** S
5. **else**:
6. mid = n // 2
7. U = mergesort(S[0 : mid])
8. V = mergesort(S[mid : n])
9. **return** merge(U, V)

**Algorithm 2.3: Merge**

1. **def** merge(U, V):
2. S = []
3. i = j = 0
4. **while** (i < len(U) **and** j < len(V)):
5. **if** (U[i] < V[j]):
6. S.append(U[i])
7. i += 1
8. **else**:
9. S.append(V[j])
10. j += 1
11. **if** (i < len(U)):
12. S += U[i : len(U)]
13. **else**:
14. S += V[j : len(V)]
15. **return** S
16. S = [27, 10, 12, 20, 25, 13, 15, 22]
17. **print**('Before: ', S)
18. X = mergesort(S)
19. **print**(' After: ', X)

**Algorithm 2.4: Merge Sort 2 (Enhanced Merge Sort)**

1. **def** mergesort2 (S, low, high):
2. **if** (low < high):
3. mid = (low + high) // 2
4. mergesort2(S, low, mid)
5. mergesort2(S, mid + 1, high)
6. merge2(S, low, mid, high)

**Algorithm 2.5: Merge2 (Enhanced Merge)**

1. **def** merge2 (S, low, mid, high):
2. U = []
3. i = low
4. j = mid + 1
5. **while** (i <= mid **and** j <= high):
6. **if** (S[i] < S[j]):
7. U.append(S[i])
8. i += 1
9. **else**:
10. U.append(S[j])
11. j += 1
12. **if** (i <= mid):
13. U += S[i : mid + 1]
14. **else**:U += S[j : high + 1]
15. **for** k **in** range(low, high + 1):
16. S[k] = U[k - low]
17. S = [27, 10, 12, 20, 25, 13, 15, 22]
18. **print**('Before: ', S)
19. mergesort2(S, 0, len(S) - 1)
20. **print**(' After: ', S)

**2.3 분할정복 설계방법**

**▪ 분할정복 설계 전략**

• 분할: 문제의 입력사례를 둘 이상의 작은 입력사례로 분할

• 정복: 작은 입력사례들을 각각 정복

작은 입력사례들이 충분히 작지 않으면 재귀 호출

• 통합: 필요하면, 작은 입력사례의 해답을 통합하여 원래 입력사례의 해답을 도출

**2.4 퀵 정렬 (분할 교환 정렬)**

**▪ 퀵 정렬: 분할 정복(Divide-and-Conquer)**

• 내부(in-place) 정렬: 추가적인 리스트를 사용하지 않는 정렬

• 추가적인 리스트를 생성하지 않고 정렬할 수 없을까?

⁃ Hoare(1962), Quick Sort Algorithm

**• QUICK-SORT**

⁃ [Divide] 기준 원소(pivot)를 정해서 기준원소를 기준으로 좌우로 분할

⁃ [Conquer] 왼쪽의 리스트와 오른쪽의 리스트를 각각 재귀적으로 퀵 정렬

⁃ [Obtain] 정렬된 리스트를 리턴

**Algorithm 2.6: Quick Sort**

1. **def** quicksort (S, low, high):
2. **if** (high > low):
3. pivotpoint = partition(S, low, high)
4. quicksort(S, low, pivotpoint - 1)
5. quicksort(S, pivotpoint + 1, high)

▪ 기준 원소(pivot)는 어떻게 정할까?

• 편의상, 일단 리스트의 첫 원소를 기준원소로 정하도록 하자

▪ 기준 원소로 어떻게 리스트를 나눌 수 있을까?

• 두 개의 인덱스(𝑖, 𝑗)를 이용해서 비교(compare)와 교환(swap)

**Algorithm 2.7: Partition (for Quick Sort)**

1. **def** partition (S, low, high):
2. pivotitem = S[low]
3. j = low
4. **for** i **in** range(low + 1, high + 1):
5. **if** (S[i] < pivotitem):
6. j += 1;
7. S[i], S[j] = S[j], S[i] # swap
8. pivotpoint = j
9. S[low], S[pivotpoint] = S[pivotpoint], S[low] # swap
10. **return** pivotpoint
12. S = [15, 22, 13, 27, 12, 10, 20, 25]
13. **print**('Before:', S)
14. quicksort(S, 0, len(S) - 1)
15. **print**(' After:', S)
17. S = [15, 22, 13, 27, 12, 10, 20, 25]
18. **print**('Before:', S)
19. partition(S, 0, len(S) - 1)
20. **print**(' After:', S)

**▪ partition() 함수의 다른 구현 방법**

1. S = [26, 5, 37, 1, 61, 11, 59, 15, 48, 19]
2. [26, 5, 37, 1, 61, 11, 59, 15, 48, 19]:2,9
3. [26, 5, 19, 1, 61, 11, 59, 15, 48, 37]:4,7
4. [26, 5, 19, 1, 15, 11, 59, 61, 48, 37]:6,5
5. [11, 5, 19, 1, 15, 26, 59, 61, 48, 37]
6. **def** partition2 (S, low, high):
7. pivotitem = S[low]
8. i = low + 1
9. j = high
10. **while** (i <= j):
11. **while** (S[i] < pivotitem):
12. i += 1
13. **while** (S[j] > pivotitem):
14. j -= 1
15. **if** (i < j):
16. S[i], S[j] = S[j], S[i] # swap
17. pivotpoint = j
18. S[low], S[pivotpoint] = S[pivotpoint], S[low] # swap
19. **return** pivotpoint
21. **def** quicksort2 (S, low, high):
22. **if** (high > low):
23. pivotpoint = partition2(S, low, high) # print(S)
24. quicksort2(S, low, pivotpoint - 1)
25. quicksort2(S, pivotpoint + 1, high)
26. **print**('Before:', S)
27. quicksort2(S, 0, len(S) - 1)
28. **print**(' After:', S)
29. **print**('Before:', S)
30. partition2(S, 0, len(S) - 1)
31. **print**(' After:', S)

**2.5 쉬트라쎈의 행렬 곱셈**

**▪ 행렬 곱셈 문제**

• 문제: 두 n×n 행렬의 곱을 구하시오

• Algorithm 1.4

⁃ 행렬 곱셈의 정의에 충실하게. 시간 복잡도는

• 행렬 곱셈의 시간 복잡도를 더 줄일 수 있을까?

⁃ Strassen(1969)

• Algorithm 2.8

⁃ 쉬트라센의 방법을 사용. 시간 복잡도는

**▪ 쉬트라쎈의 방법: 분할 정복(Divide-and-Conquer)**

• 큰 행렬을 네 개의 부분 행렬로 나누어서 정복하라.

**Algorithm 2.8: Strassen's Matrix Multiplication**

1. **def** strassen (A, B):
2. n = len(A)
3. **if** (n <= threshold):
4. **return** matrixmult(A, B)
5. A11, A12, A21, A22 = divide(A)
6. B11, B12, B21, B22 = divide(B)
7. M1 = strassen(madd(A11, A22), madd(B11, B22))
8. M2 = strassen(madd(A21, A22), B11)
9. M3 = strassen(A11, msub(B12, B22))
10. M4 = strassen(A22, msub(B21, B11))
11. M5 = strassen(madd(A11, A12), B22)
12. M6 = strassen(msub(A21, A11), madd(B11, B12))
13. M7 = strassen(msub(A12, A22), madd(B21, B22))
14. **return** conquer(M1, M2, M3, M4, M5, M6, M7)
16. **def** divide(A):
17. n = len(A)
18. m = n // 2
19. A11 = [[0] \* m **for** \_ **in** range(m)]
20. A12 = [[0] \* m **for** \_ **in** range(m)]
21. A21 = [[0] \* m **for** \_ **in** range(m)]
22. A22 = [[0] \* m **for** \_ **in** range(m)]
23. **for** i **in** range(m):
24. **for** j **in** range(m):
25. A11[i][j] = A[i][j]
26. A12[i][j] = A[i][j + m]
27. A21[i][j] = A[i + m][j]
28. A22[i][j] = A[i + m][j + m]
29. **return** A11, A12, A21, A22
31. **def** conquer(M1, M2, M3, M4, M5, M6, M7):
32. C11 = madd(msub(madd(M1, M4), M5), M7)
33. C12 = madd(M3, M5)
34. C21 = madd(M2, M4)
35. C22 = madd(msub(madd(M1, M3), M2), M6)
36. m = len(C11)
37. n = 2 \* m
38. C = [[0] \* n **for** \_ **in** range(n)]
39. **for** i **in** range(m):
40. **for** j **in** range(m):
41. C[i][j] = C11[i][j]
42. C[i][j + m] = C12[i][j]
43. C[i + m][j] = C21[i][j]
44. C[i + m][j + m] = C22[i][j]
45. **return** C
46. **def** madd (A, B):
47. n = len(A)
48. C = [[0] \* n **for** \_ **in** range(n)]
49. **for** i **in** range(n):
50. **for** j **in** range(n):
51. C[i][j] = A[i][j] + B[i][j]
52. **return** C
53. **def** msub (A, B):
54. n = len(A)
55. C = [[0] \* n **for** \_ **in** range(n)]
56. **for** i **in** range(n):
57. **for** j **in** range(n):
58. C[i][j] = A[i][j] - B[i][j]
59. **return** C
60. # Algorithm 1.4: Matrix Muiltiplication
61. **def** matrixmult (A, B):
62. n = len(A)
63. C = [[0] \* n **for** \_ **in** range(n)]
64. **for** i **in** range(n):
65. **for** j **in** range(n):
66. **for** k **in** range(n):
67. C[i][j] += A[i][k] \* B[k][j]
68. **return** C
70. A = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 1, 2, 3], [4, 5, 6, 7], ]
71. B = [[8, 9, 1, 2], [3, 4, 5, 6], [7, 8, 9, 1], [2, 3, 4, 5], ]
72. threshold = 2
73. **print**('threshold =', threshold)
74. **print**('A =', A)
75. **print**('B =', B)
76. C = strassen(A, B)
77. **for** i **in** range(len(C)):
78. **print**('C[%d] = '%(i), C[i])

**2.6 큰 정수의 계산법**

**▪ 큰 정수의 산술 문제**

• 문제: 특정 컴퓨터/언어가 표현할 수 없는 큰 정수의 산술 연산

• 가정: 10진수 체계에서의 덧셈과 곱셈

• 10진수를 소프트웨어적으로 표현하는 방법은?

⁃ 리스트를 이용하여 각 자리수(digit)를 하나의 원소로 저장

⁃ 567,832: S = [2, 3, 8, 7, 6, 5]

**▪ 큰 정수의 덧셈**

• n개의 자릿수(digit) 각각을 더하면서 올림수(carry)를 고려

**Algorithm 2.9: Arithmetic for Large Integers**

1. **def** ladd (u, v):
2. n = len(u) **if** (len(u) > len(v)) **else** len(v)
3. result = []
4. carry = 0
5. **for** k **in** range(n):
6. i = u[k] **if** (k < len(u)) **else** 0
7. j = v[k] **if** (k < len(v)) **else** 0
8. value = i + j + carry
9. carry = value // 10
10. result.append(value % 10)
11. **if** (carry > 0):
12. result.append(carry)
13. **return** result
15. u = [2, 3, 8, 7, 6, 5]
16. v = [3, 2, 7, 3, 2, 4, 9]
17. **print**(567832 + 9423723)
18. **print**(ladd(u, v)[::-1])

▪ 큰 정수의 곱셈: 단순무식한 (Brute-Force) 방법

• 초등학교에서 배운 방법

▪ 큰 정수의 곱셈: 분할정복(Divide-and-Conquer)

• n개의 자릿수(digit)로 된 숫자를 n/2개의 자릿수로 분할

• 둘 중 하나의 자릿수는 [n/2]이고, 다른 하나는 [n/2]가 됨

▪ 자릿수가 분할된 두 정수의 곱셈

• 두 개의 정수 u, v를 분할하여 곱셈 연산을 함

**Algorithm 2.9: Arithmetic for Large Integers**

1. **def** prod (u, v):
2. n = len(u) **if** (len(u) > len(v)) **else** len(v)
3. **if** (len(u) == 0 **or** len(v) == 0):
4. **return** [0]
5. **elif** (n <= threshold):
6. **return** lmult(u, v)
7. **else**:
8. m = n // 2
9. x = div(u, m);y = rem(u, m)
10. w = div(v, m); z = rem(v, m)
11. p1 = prod(x, w)
12. p2 = ladd(prod(x, z), prod(w, y))
13. p3 = prod(y, z)
14. **return** ladd(ladd(exp(p1, 2\*m), exp(p2, m)), p3)

**▪ 큰 정수의 지수 곱셈과 나눗셈**

• 10의 지수 m으로 곱하기

⁃ 왼쪽으로 m자릿수만큼 쉬프트

• 10의 지수 m으로 나눈 나머지와 몫

⁃ 1의 자리에서 m의 자리까지가 나머지

⁃ m+1에서 m의 자리까지가 몫

**Algorithm 2.9: Arithmetic for Large Integers**

1. **def** exp (u, m):
2. **if** (u == [0]):
3. **return** [0]
4. **else**:
5. **return** ([0] \* m) + u
6. **def** div (u, m):
7. **if** (len(u) < m):
8. u.append(0)
9. **return** u[m : len(u)]
10. **def** rem (u, m):
11. **if** (len(u) < m):
12. u.append(0)
13. **return** u[0 : m]
14. u = [2, 3, 8, 7, 6, 5]
15. v = [3, 2, 7, 3, 2, 4, 9]
16. **print**(exp(u, 3))
17. **print**(div(u, 3))
18. **print**(rem(u, 3))

**▪ 임계값과 단순 곱셈**

• 임계값 (threshold): 특정 자리수까지 (threshold = 1)

• 단순 곱셈: 전통적인 방법으로 곱셈

**Algorithm 2.9: Arithmetic for Large Integers**

1. **def** lmult (u, v):
2. i = u[0] **if** (0 < len(u)) **else** 0
3. j = v[0] **if** (0 < len(v)) **else** 0
4. value = i \* j
5. carry = value // 10
6. result = []
7. result.append(value % 10)
8. **if** (carry > 0):
9. result.append(carry)
10. **return** result
11. **print**(lmult([8], [7])[::-1])
12. **print**(8 \* 7)
13. u = [2, 3, 8, 7, 6, 5]
14. v = [3, 2, 7, 3, 2, 4, 9]
15. **print**(567832 \* 9423723)
16. **print**(prod(u, v)[::-1])

**▪ 큰 정수의 곱셈 알고리즘으로 우리가 한 일은?**

• 기본 연산: 한 자릿수에서의 단위 연산(총 𝑚번 실행)

• 입력 크기: 두 정수의 자릿수(𝑛개의 자릿수)

• 최선/최악/평균

⁃최악의 경우는 두 정수에 모두 0이 하나도 없을 때

• Algorithm 2.9의 시간 복잡도 분석:

⁃prod()함수에서 재귀 호출을 네 번 한다는 것에 주목

⁃n

⁃

**▪ Algorithm 2.9의 효율성 개선**

• 재귀 호출을 4번이나 하니까 효율성이 개선될 수 없다.

• 재귀 호출의 횟수를 줄일 수는 없을까?

**Algorithm 2.10: Large Integer Multiplication 2 (Enhanced)**

1. **def** prod2 (u, v):
2. n = len(u) **if** (len(u) > len(v)) **else** len(v)
3. **if** (len(u) == 0 **or** len(v) == 0):
4. **return** [0]
5. **elif** (n <= threshold):
6. **return** lmult(u, v)
7. **else**:
8. m = n // 2
9. x = div(u, m); y = rem(u, m)
10. w = div(v, m); z = rem(v, m)
11. r = prod2(ladd(x, y), ladd(w, z))
12. p1 = prod2(x, w)
13. p3 = prod2(y, z)
14. p2 = lsub(r, ladd(p1, p3))
15. **return** ladd(ladd(exp(p1, 2\*m), exp(p2, m)), p3)

▪ prod2()의 시간 복잡도는?

• 재귀 호출의 숫자를 3회로 줄임

• )

1. u = [2, 3, 8, 7, 6, 5]
2. v = [3, 2, 7, 3, 2, 4, 9]
3. **print**(567832 \* 9423723)
4. **print**(prod(u,v)[::-1])
5. **print**(prod2(u,v)[::-1])
7. **def** lsub (u, v):
8. n = len(u) **if** (len(u) > len(v)) **else** len(v)
9. result = []
10. borrow = 0
11. **for** k **in** range(n):
12. i = u[k] **if** (k < len(u)) **else** 0
13. j = v[k] **if** (k < len(v)) **else** 0
14. value = i - j + borrow
15. **if** (value < 0):
16. value += 10
17. borrow = -1
18. **else**:
19. borrow = 0
20. result.append(value % 10)
21. **if** (borrow < 0):
22. **print**("음의 정수는 처리 못함.")
23. **return** result

**2.7 분할정복과 트로미노 퍼즐**

**▪ 분할 정복: Divide-and-Conquer**

• 분할: 4개의 사분면으로 분할

⁃ X가 없는 사분면의 모서리 채우기

• 정복: 채워진 네 개의 사분면을 재귀 호출

1. **def** tromino(board, srow, scol, size, xrow, xcol):
2. **if** (size == 1):
3. **return**
4. **else**:
5. mrow = srow + (size // 2)
6. mcol = scol + (size // 2)
7. xrow1, xcol1 = mrow - 1, mcol -1
8. xrow2, xcol2 = mrow - 1, mcol
9. xrow3, xcol3 = mrow, mcol - 1
10. xrow4, xcol4 = mrow, mcol
11. **if** (xrow < mrow **and** xcol < mcol): # 1사분면
12. fillCenterExcept(board, mrow, mcol, 1)
13. xrow1, xcol1 = xrow, xcol
14. **elif** (xrow < mrow **and** xcol >= mcol): # 2사분면
15. fillCenterExcept(board, mrow, mcol, 2)
16. xrow2, xcol2 = xrow, xcol
17. **elif** (xrow >= mrow **and** xcol < mcol): # 3사분면
18. fillCenterExcept(board, mrow, mcol, 3)
19. xrow3, xcol3 = xrow, xcol
20. **elif** (xrow >= mrow **and** xcol >= mcol): # 4사분면
21. fillCenterExcept(board, mrow, mcol, 4)
22. xrow4, xcol4 = xrow, xcol
23. tromino(board, srow, scol, size // 2, xrow1, xcol1)
24. tromino(board, srow, mcol, size // 2, xrow2, xcol2)
25. tromino(board, mrow, scol, size // 2, xrow3, xcol3)
26. tromino(board, mrow, mcol, size // 2, xrow4, xcol4)
28. **def** fillCenterExcept(board, mrow, mcol, part):
29. **global** tromino\_count
30. tromino\_count += 1
31. **if** (part != 1):
32. board[mrow-1][mcol-1] = tromino\_count
33. **if** (part != 2):
34. board[mrow-1][mcol] = tromino\_count
35. **if** (part != 3):
36. board[mrow][mcol-1] = tromino\_count
37. **if** (part != 4):
38. board[mrow][mcol] = tromino\_count
40. **def** print\_board(board):
41. **for** i **in** range(m):
42. **for** j **in** range(m):
43. **if** (board[i][j] < 0):
44. **print**("%3s"%"X", end="")
45. **else**:
46. **print**("%3d"%board[i][j], end="")
47. **print**()
49. **import** random
50. m = 8
51. xrow = random.randint(0, m - 1)
52. xcol = random.randint(0, m - 1)
53. **print**(xrow, xcol)
54. board = [[0] \* m **for** \_ **in** range(m)]
55. board[xrow][xcol] = -1
56. tromino\_count = 0
57. tromino(board, 0, 0, m, xrow, xcol)
58. print\_board(board)

2.느낀 점 - 이번 학습활동으로 배운 점 혹은 시행착오를 분석한 후, 다음 학습활동 반영합니다.

어려운 문제 해결

분할 정복 알고리즘은 복잡한 문제를 해결하는 좋은 도구로, 문제를 여러 하위 문제로 분해하고 하위 문제를 하나씩 해결한 다음 결합하여 큰 문제에 대한 답을 구성할 수 있다. 예를 들어 하노이 타워 문제가 분할 정복 알고리즘을 채택하면 높이가 n인 타워의 문제를 높이가 n-1인 타워로 변환하여 문제를 해결할 수 있다. 문제가 쉽게 처리될 수 있도록 단순화될 때까지.

알고리즘 효율성

사람들은 Karatsuba 고속 곱셈 알고리즘, 빠른 정렬 알고리즘 및 병렬 알고리즘, 행렬 곱셈의 Strassen 알고리즘, 고속 푸리에 변환 등과 같은 효율적인 분할 정복 알고리즘이 많이 있음을 발견했다.

분할 정복 접근 방식은 소프트웨어 설계의 모듈식 접근 방식과 매우 유사합니다. 큰 문제를 해결하기 위해 다음을 수행할 수 있다.

1) 두 개 이상의 작은 문제로 나누다.

2) 각각의 작은 문제를 개별적으로 해결하십시오.

3) 작은 질문에 대한 답변을 결합하여 원래 질문에 대한 답변을 얻다. 작은 문제는 일반적으로 원래 문제와 유사하며 분할 정복 전략을 사용하여 재귀적으로 해결할 수 있다.