**학습활동보고서 # 1**

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

|  |  |
| --- | --- |
| 학습활동 주제 및 목표 | 알고리즘: 효율, 분석, 차수 |

(필요한 항목 및 내용을 자유로운 양식으로 작성하세요)

1. 학습활동 –내용을 상세히 작성합니다.
   1. **알고리즘이란?**

**알고리즘의 정의**

• 어떤 문제를 컴퓨터로 풀기 위한 효율적인 절차

• 문제를 푸는 단계별 절차를 명확하게 기술

**알고리즘**

• 어떤 문제의 모든 입력 사례에 대해서 해답을 찾아주는 단계별 절차

• 입력 파라미터에 어떤 입력 사례가 주어지더라도 해답을 찾을 수 있어야 함

**순차 탐색 문제**

알고리즘: 모든 S에 대해서 x의 인덱스를 찾아주는 단계별 절차

⁃S의 첫째 원소에서 시작하여 x를 찾을 때까지(x가 없는 경우 끝까지)

⁃각 원소를 차례로 x와 비교한다.

⁃만약, x를 찾으면 x의 인덱스를 리턴하고,

⁃x를 찾지 못하면 0을 리턴한다.

Algorithm 1.1: Sequential Search

1. def seqsearch (n, S, x):
2. location = 1
3. **while** (location <= n and S[location] != x):
4. location += 1
5. **if** (location > n):
6. location = 0
7. **return** location
8. S = [0, 10, 7, 11, 5, 13, 8]
9. x = 5
10. location = seqsearch(len(S) - 1, S, x)
11. print('location =', location)

**리스트(배열) 원소의 합 구하기**

알고리즘: S의 모든 원소를 차례대로 sum에 더하는 절차

⁃ sum을 0으로 초기화

⁃ 모든 S의 원소에 대해서 sum += S[i]를 실행

⁃ sum의 값을 리턴

Algorithm 1.2: Add Array Elements

1. def sum (n, S):
2. result = 0
3. **for** i in range(1, n + 1):
4. result = result + S[i]
5. **return** result
6. S = [-1, 10, 7, 11, 5, 13, 8]
7. sum = sum(len(S) - 1, S)
8. print('sum =', sum)

**리스트의 정렬 문제**

알고리즘: 모든 S에 대해서 S′을 찾아주는 단계별 절차

⁃ 교환 정렬, 삽입 정렬, 선택 정렬, 합병 정렬, 퀵 정렬, 기타 등등. ⁃ 여러 가지 정렬 알고리즘 중에서 교환 정렬 방법으로 구현.

Algorithm 1.3: Exchange Sort

1. def exchange(S):
2. n = len(S)
3. **for** i in range(n - 1):
4. **for** j in range(i + 1, n):
5. **if** (S[i] > S[j]):
6. S[i], S[j] = S[j], S[i] # swap
7. S = [-1, 10, 7, 11, 5, 13, 8]
8. print('Before =', S)
9. exchange(S)
10. print('After =', S)

교환 정렬

• i번째 자리에 있는 수와 (i+1)번째부터 n번째 자리에 있는 수를 차례대로 비교

• 주어진 자리의 수가 i번째 자리에 있는 수보다 작은 경우, 두 수를 교환

• for-i 루프를 한 번 수행하면 그 중 가장 작은 수가 첫번째 자리에 들어감

• 두번째 루프를 수행하면 둘째 자리에 둘째로 작은 수가 들어감

• 이런 과정을 통해 for-i루프가 모두 수행되면 비내림차순 정렬이 됨

**행렬의 곱셈 문제**

두 n×n 행렬의 곱을 구하시오

Algorithm 1.4: Matrix Multiplicaton

1. def matrixmult (A, B):
2. n = len(A)
3. C = [[0] \* n **for** \_ in range(n)]
4. **for** i in range(n):
5. **for** j in range(n):
6. **for** k in range(n):
7. C[i][j] += A[i][k] \* B[k][j]
8. **return** C
9. A = [[2, 3], [4, 1]]
10. B = [[5, 7], [6, 8]]
11. print('A =', A)
12. print('B =', B)
13. C = matrixmult(A, B)
14. print('C =', C)

**1.2 알고리즘의 효율성**

**알고리즘의 효율성**

• 알고리즘의 성능: 시간과 공간 사용량의 효율성

• 알고리즘의 성능은 컴퓨터의 실행 속도나 메모리의 가격에 무관

**순차 탐색 .vs. 이분 검색**

• 입력 리스트의 조건에 따른 탐색 알고리즘의 선택

⁃ 정렬되지 않은 리스트에서 키 찾기: 순차 탐색

⁃ 정렬된 리스트에서 키 찾기: 이분 검색

**▪ 이분 검색**

• 주어진 리스트 S와 키 x에 대해서,

• 먼저 x를 리스트의 중앙에 위치한 원소와 비교

• 만약 같으면, 찾았으므로 알고리즘을 종료

• 만약 x가 그 원소보다 작으면 x는 왼쪽에 있을 것이므로

⁃ 왼쪽 리스트에 대해서 이진 탐색 실행 (재귀 호출)

• 만약 x가 그 원소보다 크면 x는 오른쪽에 있을 것이므로

⁃ 오른쪽 리스트에 대해서 이진 탐색 실행 (재귀 호출)

• 더 이상 찾을 리스트가 없으면 알고리즘을 종료

Algorithm 1.5: Binary Search (Iterative)

1. def binsearch(n, S, x):
2. low = 1
3. high = n
4. location = 0
5. **while** (low <= high and location == 0):
6. mid = (low + high) // 2
7. **if** (x == S[mid]):
8. location = mid
9. elif (x < S[mid]):
10. high = mid - 1
11. **else**:
12. low = mid + 1
13. **return** location
14. S = [-1, 5, 7, 8, 10, 11, 13]
15. x = 2
16. # x = 7
17. # x = 13
18. location = binsearch(len(S) - 1, S, x)
19. print('S =', S)
20. print('x =', x)
21. print('location =', location)

**순차 탐색과 이분 검색 알고리즘의 효율성 비교**

• 순차 탐색: 크기가 n인 리스트에서 n번의 비교를 수행

• 이분 검색: 크기가 n인 리스트에서 lgn + 1번의 비교를 수행

**▪ 피보나치 수열의 n번째 항 구하기**

• 피보나치 수열: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

• 피보나치 수열의 (재귀적) 정의

⁃

⁃

**Algorithm 1.6: Finding the 𝒏-th Fibonacci Term (Recursive)**

1. def fib (n):
2. **if** (n <= 1):
3. **return** n
4. **else**:
5. **return** fib(n - 1) + fib(n - 2)
6. **for** i in range(11):
7. print(fib(i), end=" ")

**▪ Algorithm 1.6의 효율성**

• 재귀적 정의 이용: 작성하기도 쉽고 이해하기도 쉬움

• 그러나 너무 비효율적이다.

**Algorithm 1.6의 비효율성을 개선하려면?**

• 같은 값을 중복해서 재귀적으로 계산하지 않도록 해야 함

• 아직 계산하지 않은 피보나치 항의 값은 계산을 해야 함

• 이미 계산한 피보나치 항의 값은 리스트에 저장

• 이미 계산되어 저장된 피보나치 항은 필요할 때 꺼내쓰면 된다

**Algorithm 1.7: Finding the n-th Fibonacci Term (Iterative)**

1. **def** fib2 (n):
2. f = [0] \* (n + 1)
3. **if** (n > 0):
4. f[1] = 1
5. **for** i **in** range(2, n + 1):
6. f[i] = f[i - 1] + f[i - 2]
7. **return** f[n]
8. **for** i **in** range(11):
9. **print**(fib2(i), end=" ")

**1.3 알고리즘의 분석**

**알고리즘의 분석**

• 정확성 분석: 모든 입력 사례에 대해서 정확한 해답을 찾는다는 것을 증명

• 효율성 분석: 입력 크기가 커지는 정도에 따라 성능의 변화량을 증명

⁃ 시간 복잡도(time complexity): 시간을 기준으로 알고리즘의 효율성 분석

⁃ 공간 복잡도(space complexity): 공간을 기준으로 알고리즘의 효율성 분석

**▪ 알고리즘의 성능 분석**

• 퍼포먼스 측정: 실행 시간을 직접 측정 or 실행 명령의 숫자 세기

⁃ 컴퓨터의 성능이나 프로그래밍 언어에 따라 달라짐

• 복잡도 분석: 컴퓨터나 프로그래밍 언어와 무관하게 성능 분석

⁃ 입력 크기에 따른 단위 연산의 실행 횟수 세기

**▪ 복잡도 분석**

• 입력 크기: 문제가 가진 파라미터, 즉, 입력 사례의 크기

• 단위 연산: 알고리즘 실행의 기본이 되는 명령어들의 집합

**▪ Algorithm 1.2(배열 원소의 합)의 시간 복잡도 분석**

• 단위 연산: 리스트의 원소를 result에 더하는 명령

• 입력 크기: 리스트 S의 원소 개수(n)

• for 문장은 항상 n번 실행하므로 다음과 같이 표현

⁃ 시간 복잡도: T(n)=n

1. **def** sum (S):
2. n = len(S)
3. result = 0
4. **for** i **in** range(n):
5. result += S[i]
6. **return** result

**▪ Algorithm 1.3(교환 정렬)의 시간 복잡도 분석**

• 단위 연산: S[i]와 S[j]의 비교

• 입력 크기: 정렬할 리스트 S의 원소 개수(n)

• for-j 루프는 i에 따라 n−1번에서 1번까지 실행하므로 다음과 같이 계산

• 시간 복잡도:

1. **def** exchange (S):
2. n = len(S)
3. **for** i **in** range(n - 1):
4. **for** j **in** range(i + 1, n):
5. **if** (S[i] > S[j]):
6. S[i], S[j] = S[j], S[i] # swap

▪ **Algorithm 1.4(행렬 곱셈)의 시간 복잡도 분석**

• 단위 연산: 가장 안쪽 for 루프에 있는 곱셈

• 입력 크기: 행과 열의 개수(n)

• 3중 for 루프가 항상 n번 실행하므로 다음과 같이 계산

• 시간 복잡도:

1. **def** matrixmult (n, A, B):
2. C = [[0] \* n **for** \_ **in** range(n)]
3. **for** i **in** range(n):
4. **for** j **in** range(n):
5. **for** k **in** range(n):
6. C[i][j] += A[i][k] \* B[k][j]
7. **return** C

**▪ Algorithm 1.1(순차 탐색)의 시간 복잡도 분석**

• 단위 연산: 리스트의 원소와 주어진 키 x와의 비교 연산

• 입력 크기: 리스트 원소의 개수(n)

• 최악의 경우는 모두 비교: W(n) = n

• 최적의 경우는 한 번만 비교: B(n) = 1

• 평균의 경우: 주어진 키 x가 k번째에 있으면 k번을 비교함

⁃ 만약 어떤 키 x는 리스트 S에 골고루 분포해 있다고 한다면,

**1.4 알고리즘의 차수**

**▪ 차수(Order): 알고리즘의 궁극적인 성능 분류**

• 1차 시간 알고리즘: 시간 복잡도가 1차 함수인 알고리즘

• 2차 시간 알고리즘: 시간 복잡도가 2차 함수인 알고리즘

• 근본 원리: 모든 1차 시간 알고리즘은 궁극적으로 2차 시간 알고리즘보다 빠르다.

• 따라서, 시간 복잡도 함수의 차수로 알고리즘의 성능을 분류할 수 있다

**▪ 점근적 표기법: O, Θ,Ω**

• 빅오(O): 복잡도 함수의 점근적 상한을 표기

• 오메가(Ω): 복잡도 함수의 점근적 하한을 표기

• 쎄타(Θ)=차수: 복잡도 함수의 점근적 상한과 하한을 동시에 만족

**▪ Algorithm 1.3(교환 정렬)의 차수**

•

•

• 그러므로, 교환 정렬 알고리즘의 차수는

**▪ Algorithm 1.1(순차 탐색)의 차수**

• 최악의 경우:

• 최적의 경우:

• 평균의 경우:

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

효율, 분석, 차수의 세 가지 측면에서 알고리즘을 자세히 배웠다.