알고리즘 개념 및 비교

20 기 인턴 강은구

목차

1. 알고리즘 기본 개념

- **알고리즘 (Algorith)**
  - 어떤 문제를 해결하기 위해 정해진 순서와 절차에 따라 작업을 수행하는 방법.
  - "문제를 풀기 위한 요리 레시피" 같은 것
- **입력(intput)**
  - 알고리즘이 처리해야 하는 자료
  - [5, 4, 3, 2, 1]
- **효율성(Efficiency)**
  - 알고리즘이 얼마나 빠르고 적은 자원으로 동작하는지
- **시간 복잡도(Time Complexity)**
  - 알고리즘이 실행되는 데 걸리는 시간의 이론적 크기
  - O(n), O(n^2), O(log n), ...
  - N은 입력 데이터 크기
- **공간 복잡도(Space Complexity)**
  - 알고리즘이 실행되는 동안 필요한 메모리 양

2. 시간 복잡도 기초

- O(1): 입력 크기와 상관없이 항상 일정한 시간(배열 원소 하나 접근)
- O(n): 입력 크기에 비례해서 실행 시간 증가(선형 탐색)
- O(log n): 입력을 절반씩 줄여가는 경우(이진 탐색)
- O(n log n): 큰 데이터를 효율적으로 처리(퀵 정렬, 병합 정렬)
- O(n^2): 중첩 반복문이 있을 때 자주 등장(버블 정렬, 선택 정렬)

3. 알고리즘 종류 개요

- 정렬 알고리즘: 데이터를 순서대로 배열
- 탐색 알고리즘: 원하는 값을 찾기
- 트리 알고리즘: 계층 구조 데이터를 다룸
- 그래프 알고리즘: 여러 지점과 연결을 다룸
- 최단 경로 알고리즘: 출발점에서 도착점까지 가장 짧은 길 찾기

1.2 **알고리즘**

1. **알고리즘의 조건**
    - 정확성: 항상 올바른 결과를 요구
    - 유한성: 반드시 끝나야 함
    - 명확성: 각 단계가 애매하지 않고 명확
    - 입력: 0 개 이상 종재
    - 출력: 최소 1 개 이상 존재
2. **시간 복잡도/공간 복잡도**
    - 시간 복잡도: 알고리즘이 얼마나 빨리 실행되는지(입력 크기 n 에 따른 실행 횟수)
    - 공간 복잡도: 얼마나 많은 메모리를 쓰는지
3. **자료 구조와 알고리즘 관계**
    - 자료구조: 데이터를 저장하고 관리하는 방법
    - 알고리즘: 자료구조를 이용해 문제를 해결하는 방법
    - 예시:

i. 배열 + 정렬 알고리즘; 데이터를 순서대로 정리

ii. 트리 자료구조 + 탐색 알고리즘: 빠른 검색

iii. 그래프 자료구조: 최단 경로 알고리즘: 길 찾기

1.3 알고리즘과 자료 구조의 관계

1. **자료구조(Data Structure)**
   ① 데이터를 효율적으로 저장하고 관리하는 방법
   ② 배열, 연결리스트, 스택, 큐, 트리, 그래프, ...
2. **알고리즘(Algorithm)**
   ① 자료구조를 활용해서 문제를 해결하는 절차
   ② 정렬, 탐색, 최단경로 탐색, ...
3. **관계**
   ① 자료구조는 데이터를 담는 그릇
   ② 알고리즘은 그릇 안에 든 데이털르 효율적으로 꺼내거나 조작하는 방법
4. **예시**
   ① 배열(Array) +정렬 알고리즘: 배열에 저장된 데이터를 선택 정렬, 퀵 정렬 등으로 정리
   ② 트리(Tree) + 탐색 알고리즘: 이진 탐색 트리(BST)에서 원하는 값을 빠르게 찾음
   ③ 그래프(Gragp) + 최단 경로 알고리즘: 지도 데이터를 그래프로 표현하고, 다익스트라 A* 알고리즘으로 최단 경로 계산
5. **정리**
   ① 자료구조 없이는 알고리즘이 효율적으로 동작할 수 없음
   ② 알고리즘 선택은 자료 구조에 따라 달라짐(배열 vs 연결 리스트-> 탐색/삽입 성능 차이)

자료구조	관련 알고리즘	예시
배열	정렬 알고리즘	선택 정렬, 퀵 정렬
트리	탐색 알고리즘	이진 탐색 트리, AVL
그래프	최단경로 알고리즘	다익스트, A*

1.4 알고리즘 평가 기준

1. **시간 복잡도(Time Complexity)**
   ① 알고리즘이 문제를 해결하는 데 걸리는 실행 시간을 입력 크기 n 의 함수로 표현
   ② 보통 연산 횟수를 기준으로 계산
   ③ 표기 방법: 빅오 표기법(Big-O Notation)
   - O(1): 입력 크기와 상관 없이 항상 일정한 시간(배열 원소 접근)
   - O(n): 입력 크기에 비례(선형 탐색)
   - O(log n): 입력 크기를 절반씩 줄여가는 경우(이진 탐색)
   - O(n log n): 효율적인 정렬 알고리즘(퀵정렬, 병합 정렬)
   - O(n^2): 이중 반복문이 있는 경우(버블 정렬, 선택 정렬)
2. **공간 복잡도(Space Complexity)**
   ① 알고리즘이 실행되는 동안 필요한 메모리 공간의 크기
   ② 사용되는 공간 요소
   - 고정 공간: 코드, 변수, 상수 등(입력 크기와 무관)
   - 가변 공간: 입력 데이터, 동적 할당 메모리, 재귀 호출 스택 등
3. **예시**
   ① 버블 정렬: O(n^2)시간, O(1)공간(제자리 정렬)
   ② 병합 정렬: O(n log n) 시간, O(n)공간(추기 배열 필요)
   ③ 이진 탐색: O(log n)시간, O(1) 공간
4. **정리**
   ① 빠른 알고리즘은 시간 복잡도가 낮음
   ② 메모리 절약형 알고리즘은 공간 복잡도가 낮음
   ③ 실제 문제 해결에서는 시간과 공간을 균형 있게 고려해야 함

1.5 빅오 (Big-O)표기법과 주요 복잡도 비교

1. 빅오 표기법

- 알고리즘의 실행 시간이 입력 크기(n)에 따라 어떻게 증가하는지를 나타내는 수학적 표현
- 프로그램의 실제 실행 시간을 측정하는 것이 아니라, 성능의 상한선을 분석하는 방법

2. 주요 복잡도 정류

- O(1): 상수 시간→입력 크기와 상관없이 일정한 시간
- O(log n): 로그시간→입력을 전반씩 줄여 탐색
- O(n): 선형 시간→모든 원소를 한 번씩 확인
- O(n log n): 준선형 시간→중첩 반복문
- O(n^2): 이차 시간→중첩 반복문
- O(2^n): 지수 시간→ 부분집합을 전부 탐색
- O(n!): 계승 시간→가능한 모든 순열을 탐색

3. 복잡도 증가 예시

복잡도	n=10	n=100	n=1000
O(1)	1	1	1
O(log n)	3	7	10
O(n)	10	100	1000
O(n log n)	33	664	9966
O(n^2)	100	10000	1000000
O(2^n)	1024	2^100	2^1000
O(n!)	10!	100!	1000!

4. 시각화

- O(1)과 O(log n)은 거의 증가하지 않음
- O(n)과 O(n log n)은 실제로 가장 실용적
- O(n^2)이상은 입력에 따라 기하급수적으로 증가

5. 정리

- 빅오 표기법은 알고리즘 성능을 비교하는 표준적인 방법
- 실제 문제 해겨에서는 O(n log n)이하 알고리즘을 선호
- O(n^2)이상은 작은 데이터만 적용 가능

2.1 선택 정렬

1. **개념**

   : 배열에서 가장 작은 값을 찾아 맨 앞의 원소와 교환하는 과정을 반복하는 정렬 방법

2. **동작 과정...[5, 4, 3, 2, 1]**
   - ① 전체에서 최솟값 1 을 찾아 맨 앞 5 와 교환->[1, 3, 5, 4, 2]
   - ② 남은 구간(3,5,4,2)에서 최솟값 2 를 찾아 3 과 교환->[1, 2, 5, 4, 3[
   - ③ 남은 구간 (5,4,3)에서 최솟값 3 을 찾아 5 와 교환->[1, 2, 3, 4, 5]
   - ④ 남은 구간 (4, 5) 생략(이미 완료)

3. **시간 복잡도**
   - ① 비교 횟수: 항상 n(n-1)/2
   - ② 교환 횟수: n-1 번
   - ③ 최선/최악/평균 경우가 모두 O(n^2)

4. **특징**
   - ① 구현이 매우 단순
   - ② 데이터 양이 많을 때는 비효율적
   - ③ 교환 횟수가 적어 교휜비용이 큰 경우에는 버블 정렬보다 유리

5. **정리**
   - ① 단순하지만 느린 정렬 알고리즘

2.2 버블 정렬(Bubble Sort)

1. **개념**
   ① 인접한 두 원소를 비교해서 앞의 값이 크면 교환하는 방식
2. **동작 과정**
   ① 첫 번째 원소부터 인접한 값끼리 비교
   - (5,3)->교환->[3,5,1,4,2]
   - (5,1)->교환->[3,1,5,4,2]
   - (5,4)->교환->[3,1,4,5,2]
   - (5.2)->교환->[3,1,4,2,5]

   ② 다음 반복에서 마지막 정렬된 원소를 제외하고 반복
3. **시간 복잡도**
   ① 이미 정렬된 경우: 교환 없음
   ② 평균/최악의 경우:
   ③ 교환과 비교가 모두 많이 일어나서 비효율적
4. **특징**
   ① 구현이 매우 간단
   ② 데이터가 거의 정렬된 경우에 매우 효율적
   ③ 실제 큰 데이터에는 반복횟수 증가하므로 사용 X

2.3 삽입 정렬(Insertion Sort)

1. **개념**
   ① 새로운 원소를 이미 정렬된 부분에 알맞은 위치에 삽입
2. **동작 과정...[5, 3, 1, 4, 2]**
   ① 첫 번째 원소 5 는 이미 정렬된 상태로 시작->[3, 5, 1, 4, 2]
   ② 3 을 5 앞에 삽입->[1, 3, 5, 4, 2]
   ③ 1 을 맨 앞으로 삽입[1, 3, 5, 4, 2]
   ④ 4 를 3 과 5 사이에 삽입[1, 3, 4, 5, 2]
   ⑤ 2 를 1 과 3 사이에 삽입[1, 2, 3, 4, 5]
3. **시간 복잡도**
   ① 이미 정렬된 경우: O(n)
   ② 평균/최악의 경우: O(n^2)
   ③ 작은 데이터는 빠르고 효율적
4. **특징**
   ① 구현이 간단하고 직관적
   ② 데이터가 거의 정렬되어 있으면 매우 효율적
   ③ 추가 메모리가 거의 필요 없음
   ④ 데이터 크기가 크면 비효율적
5. **정리**
   ① 작은 데이터 집합 또는 거의 정렬된 데이터에서 효율적
   ② 데이터가 많으면 비교 횟수가 기하급수적으로 늘어남

2.4 퀵 정렬(Quick Sort)

1. **개념**
   ① 분할 정복(Divide and Conquer)방식의 정렬 알고릿므.
   ② 하나의 원소를 피벗(pivot, 기준값)으로 정하고, 피벗보다 작은 값은 왼쪽, 큰 값은 오른쪽을 분한
   ③ 각 부분 배열을 재귀적으로 정렬
2. **동작...[5, 3, 1, 4, 2]**
   ① 피벗 선택...5
      - [3, 1, 4, 2] / [ ]
   ② 왼쪽 구간 [3, 1, 4, 2]에서 피벗 선택...3
      - [1,2] / [4]
   ③ 각각 다시 정렬→ [1, 2, 3, 4, 5]
3. **시간 복잡도**
   ① 평균: O(n log n)
   ② 최대 n^2 회까지 커질 수 있음
   ③ 보통 형균 성능이 좋아 실무에서 많이 사용
4. **특징**
   ① 실제로 가장 많이 사용되는 정렬 알고리즘 중 하나
   ② 추가 메모리 사용이 거의 없음(제자리 정렬 가능)
   ③ 불안정 정렬(같은 값의 순서가 바뀔 수 있음).
   ④ 피벗 선택 방법(첫 원소, 마지막 원소, 무작위 선택 등)에 따라 성능이 달라짐.
5. **정리**
   ① 퀵 정렬은 실무에서 가장 많이 쓰이는 고속 정렬 알고리즘
   ② 평균 O(n log n)의 뛰어난 성능을 가지지만 최악의 경우 반복 횟수가 늘어남

2.5 병합 정렬(Merge Sort)

1. **개념**
   ① 분할 정복(Divide and Conquer)방식을 사용하는 정렬 알고리즘
   ② 배열을 절반으로 나누어 각각 정렬한 후, 두 개의 정렬된 배열을 합병하는 방식
2. **동작 과정...[5, 3, 1, 4, 2]**
   ① 배열을 반으로 나눔->[5,3,1],[4,2]
   ② 다시 나눔->[5], [3,1], [4], [2]
   ③ 원소가 하나가 될 때까지 나눈 후, 합치면서 정렬
      - [3,1]->[1,3]
      - [4,2]->[2,4]
      - [5] + [1,3]->[1,3,5]
      - [2,4]와 합쳐 최종 [1, 2, 3, 4, 5]
3. **시간 복잡도**
   ① 항상 O(n log n)
   ② 합병 과정에서 모든 원소를 비교하므로 일정한 성능 보장
4. **특징**
   ① 안정 정렬(같은 값의 순서가 유지됨).
   ② 성능이 안정적이고 대용량 데이터에 적합
   ③ 추가 메모리 공간 O(n)이 필요하다는 단점이 발생
5. **정리**
   ① 병합 정렬은 항상 일정한 성능이 보장되는 안정적인 정렬 알고리즘
   ② 추가 메모리가 필요하기 때문에 메모리 사용이 중요한 환경에서는 불리하다는 단점이 존재

2.6 힙 정렬

1. **개념**
   ① 힙(Heap)이라는 자료구조를 이용한 정렬 방법
   ② 힙 완전 이진 트리(모든 층이 꽉 차 있거나 마지막 층만 왼쪽부터 채워진 트리)형태
      - 최대 힙(Max Heap): 부모 노드>= 자식 노드
      - 최소 힙(Min Heap): 부모 노드<= 자식 노드
   ③ 힙 정렬은 최대 힙을 구성해서 가장 큰 값을 꺼내 배열 끝에 배치하는 과정을 반복
2. **동작 과정...[5, 3, 1, 4, 2]**
   ① 배열 최대 힙으로 구정->[5, 4, 1, 3, 2]
   ② 루트(최댓값 5)와 마지막 원소 교환->[2, 4, 1, 3, 5]
   ③ 남은 구간을 다시 최대 힙으로 만들고 반복
   ④ 최종 결과: [1, 2, 3, 4, 5]
3. **시간 복잡도**
   ① 힙 구성: O(n)
   ② 힙에서 원소 꺼내기: O(log n) * n 번 = O(n log n)
   ③ 최선/평균/최대 모두 O(n log n)
4. **특징**
   ① 추가 메모리가 거의 필요 없음
   ② 안정 정렬이 아님(같은 값의 순서가 바뀔 수 있음).
   ③ 실행 시간은 병합 정렬보다 조금 길 수 있지만, 메모리 효율이 좋아 실무에서 종종 사용
5. **정리**
   ① 힙 정렬은 시간 복잡도를 보장하면서도 추가 메모리가 거의 필요 없는 효율적인 정렬 알고리즘
   ② 안정 정렬이 아니므로, 데이터의 상대적 순서를 유지해야 하는 경우에는 적합하지 않음

2.7 비교

1. 정렬 알고리즘 비교

① 시간 복잡도 비교

알고리즘	최선	평균	최악	공간복잡도	안정성
선택 정렬	O(n^2_	O(n^2)	O(n^2)	O(1)	x
버블 정렬	O(n)	O(n^2)	O(n^2)	O(1_	o
삽입 정렬	O(n_	O(n^2)	O(n^2)	O(1)	o
퀵 정렬	O(n log n)	O(n log n)	O(n^2)	O(n log n)	x
병합 정렬	O(n log n)	O(n log n)	O(n log n)	O(n)	o
힙 정렬	O(n log n)	O(n log n)	O(n log n)	O(1)	x

2. 특징

① 선택 정렬: 단순, 교환 횟수 적음. 하지만 전체적으로 느림.

② 버블 정렬: 구현 직관적, 거의 정렬된 데이터에만 쓸 만함.

③ 삽입 정렬: 소규모 데이터, 거의 정렬된 데이터에서 효과적

④ 퀵 정렬: 평균적으로 가장 바름. 피벗 선택 중요. 불안정 정렬.

⑤ 병합 정렬: 항상 O(n log n), 안정 정렬, 추가 메모리 필요.

⑥ 힙 정렬: O(n log n)보장, 메모리 효율적, 하지만 실제 성능은 퀵보다 약간 느림

3. 상황별 사용 정렬

① 데이터가 작거나 거의 정렬됨-> 삽입 정렬, 버블 정렬

② 일반적인 경우->퀵 정렬(실무에서 가장 많이 사용)

③ 안정 정렬 필요(동일 값 순서 유지 등)-> 병합 정렬

④ 메모리 제한이 있는 경우-> 힙 정렬

4. 정리

① 정렬 알고리즘은 상황에 다라 장단점이 있음

② 실무에서는 보통 퀵 정렬과 병합 정렬이 많이 쓰임

③ 버블/선택/삽입 정렬은 교육용 또는 특수한 상황에서 활용