자료구조 (Data Structure)
7주차: 정렬

오늘의 목차

- 정렬 알고리즘의 필요성과 정의
- 정렬 알고리즘의 종류

  - heap sort와 selection sort
  - quick sort와 insertion sort

  - merge sort
  - counting sort

정렬 알고리즘의 필요성

- 데이터를 정렬하는 이유는?

  - 유저의 필요

  - 프로그램 성능 향상

정렬의 정의

- 데이터를 특정 기준에 따라 순서대로 배열하는 과정
- 오름차순과 내림차순 정렬

HEAP_SORT

- 힙 자료구조를 이용한 정렬 방법

HEAP_SORT 과정

1. 배열로부터 최대 힙 생성

2. 루트 노드를 제거하고 마지막 노드와 교체

3. 나머지 노드들에 대해 힙 속성 회복

4. 2~3 과정을 반복하여 정렬 완료

HEAP_SORT 시각화

- 예시: [5, 3, 2, 7, 1] 정렬 과정

  - 최대 힙 생성: [7, 5, 2, 3, 1]
  - 1회전: [5, 3, 2, 1] + [7]
  - 2회전: [3, 1, 2] + [5, 7]
  - 3회전: [2, 1] + [3, 5, 7]
  - 4회전: [1] + [2, 3, 5, 7]
  - 5회전: [] + [1, 2, 3, 5, 7]

HEAP_SORT

함수 HEAP_SORT(배열리스트 L):

- 힙 heap = 새 최대힙(L->data, L->size)
- 반복: i는 L의 전체 인덱스를 거꾸로

  L->data[i] = HEAP_POP(heap)
- 삭제: heap

HEAP_SORT의 시간 복잡도

최대 힙 생성시 시간복잡도 (perfect binary tree):

- 트리의 끝 층 노드(leaf): 비교 0회, 교환 0회
- 끝의 한층 위 노드(leaf 위): 비교 2회, 교환 1회
- … 루트 노드: 비교 2h 회, 교환 h 회
- 비교 총 합: $2 \times 0 \times \frac{n}{2} + 2 \times 1 \times \frac{n}{2^2} + \cdots + 2 \times h \times \frac{n}{2^{h+1}}$

$$= n \sum_{i=0}^{h} \frac{i}{2^i}$$

$$\approx 2n$$

HEAP_SORT의 시간 복잡도

HEAP_POP 수행시 시간복잡도 (perfect binary tree):

- 트리 높이가 h이면: 비교 2h 회, 교환 h 회
- … 높이가 1이면: 비교 2 회, 교환 1회
- 비교 총 합: $2 \times h \times \frac{n}{2} + 2 \times (h - 1) \times \frac{n}{2^2} + \cdots$

$$= \sum_{i=0}^{h-1} \left((h-i) \frac{n}{2^i} \right)$$

$$= hn \sum_{i=0}^{h-1} \frac{1}{2^i} - n \sum_{i=0}^{h-1} \frac{i}{2^i}$$

$$= 2hn - 2n$$

$$\approx 2n \log n$$

HEAP_SORT의 시간 복잡도

HEAP_SORT의 시간복잡도:

- 비교 횟수 $\approx 2n \log n$
- 교환 횟수 $\approx n \log n$
- $O(n \log n)$

SELECTION_SORT

- HEAP_SORT와 같은 전략

  - 정렬 전 구간의 최대값을 맨 뒤로 이동

- 최대값을 찾을 때 모든 값을 확인해서 느림

SELECTION_SORT 과정

1. 배열에서 최대값 찾기

2. 최대값을 맨 뒤의 원소와 교체

3. 두번째 최대값을 뒤에서 두번째 원소와 교체

4. ... 반복

SELECTION_SORT 시각화

- 예시: [5, 3, 2, 7, 1] 정렬 과정

  1회전: [5, 3, 2, 1, 7]
  2회전: [1, 3, 2, 5, 7]
  3회전: [1, 2, 3, 5, 7]
  4회전: [1, 2, 3, 5, 7]
  5회전: [1, 2, 3, 5, 7]

SELECTION_SORT

함수 SELECTION_SORT(리스트 L, 인덱스 lo, 인덱스 hi):

- 반복: pos는 hi부터 lo까지

    max_i = pos

    반복: i는 pos부터 lo까지

        만약: L->data[i] > L->data[max_i]면

            max_i = i

    SWAP(L, max_i, pos)

SELECTION_SORT

함수 SWAP(리스트 L, 인덱스 i, 인덱스 j):

- tmp = L->data[i]
- L->data[i] = L->data[j]
- L->data[j] = tmp

SELECTION_SORT의 시간 복잡도

비교 횟수:

- 최대값을 찾을 때: $n-1$
- 두번째 최대값을 찾을 때: $n-2$
- …
- 총합: $(n-1)+(n-2)+\cdots=\frac{n(n-1)}{2}=O(n^2)$

교환 횟수:

- n - 1번

HEAP_SORT와 SELECTION_SORT 비교

- 시간 복잡도
  - HEAP_SORT: O(n log n)
  - SELECTION_SORT: $O(n^2)$
- 공간 복잡도
  - 둘 다: O(1)
- HEAP_SORT는 빅데이터에서 더 나은 성능을 제공

QUICK_SORT

- 가장 빠른 정렬 알고리즘
- 기준값을 정해서 작은 값들과 큰 값들을 분리, 재배치

QUICK_SORT 과정

1. 기준값 선택 (제일 끝 값)

2. 기준값을 기준으로 작은 값들과 큰 값들로 분할

3. 작은 값들과 큰 값들에 대해 1~2 과정을 반복

QUICK_SORT 시각화

- 예시: [5, 3, 2, 7, 1] 정렬 과정

기준값 1 선택: [1], [5, 3, 2, 7]
기준값 7 선택: [1], [5, 3, 2], [7]
기준값 2 선택: [1], [2], [5, 3], [7]
기준값 5 선택: [1], [2], [3], [5], [7]

최종 정렬 결과: [1, 2, 3, 5, 7]

QUICK_SORT

함수 QSORT(리스트 L, 인덱스 lo, 인덱스 hi):

- 만약: lo < hi이면

  pivot = PARTITION(L, lo, hi)

  QSORT(L, lo, pivot - 1)

  QSORT(L, pivot + 1, hi)

QUICK_SORT

함수 PARTITION(리스트 L, 인덱스 lo, 인덱스 hi):

- pivot_data = L->data[hi]
- left_rear = lo // pivot이하 그룹 끝 + 1
- 반복: i는 lo부터 hi - 1까지
  - 만약: L->data[i] <= pivot_data이면
    - SWAP(L, left_rear, i)
    - left_rear += 1
- SWAP(L, left_rear, hi) // 기준점 위치
- 반환: left_rear

QUICK_SORT의 시간 복잡도

QUICK_SORT의 평균 비교 횟수 (왼쪽이 k개일 때):

- $C(n) = n - 1 + C(k) + C(n - k - 1)$

  평균 $= n - 1 + \frac{2}{n}\sum C(k)$

  $nC(n) = n(n-1) + 2\sum C(k)$

  $(n-1)C(n-1) = (n-1)(n-2) + 2\sum C(k)$

  $nC(n) - (n+1)C(n-1) = 2(n-1)$

  $\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \approx \frac{C(n-1)}{n} + \frac{2}{n} - \frac{4}{n(n+1)} \approx 2\ln n$

  $C(n) \approx 2n \ln n = O(n \log n)$

- 평균 교환 횟수는 $C(n)/2 \approx n \ln n$

QUICK_SORT의 시간 복잡도

QUICK_SORT의 최악 비교 횟수 (왼쪽이 0개로 나뉠 때):

- $C(n) = (n-1) + C(n-1)$

$= (n-1) + (n-2) + C(n-2)$

$= (n-1) + (n-2) + \cdots$

$= \frac{(n-1)n}{2}$

$= O(n^2)$

INSERTION_SORT

- 정렬된 부분의 적절한 위치로 데이터를 이동해간다

INSERTION_SORT 과정

왼쪽부터 정렬해나가는 경우:

1. 정렬되지 않은 부분의 왼쪽 원소에 대해서,
   정렬된 부분에서 적절한 위치 찾기

2. 찾은 위치에 추가하여 정렬된 부분 확장

3. ... 반복

INSERTION_SORT 시각화

- 예시: [5, 3, 2, 7, 1] 정렬 과정

  1회전: [3, 5, 2, 1, 7]
  2회전: [2, 3, 5, 1, 7]
  3회전: [1, 2, 3, 5, 7]
  4회전: [1, 2, 3, 5, 7]

INSERTION_SORT

함수 INSERTION_SORT(리스트 L, 인덱스 lo, 인덱스 hi):

- 반복: rear는 lo부터 hi까지

```
value = L->data[rear]     // 이동할 데이터
i = rear                  // 데이터를 이동할 위치
반복: i > lo이고, L->data[i - 1] > value인 동안
        L->data[i] = L->data[i - 1]
        i -= 1
L->data[i] = value
```

INSERTION_SORT의 시간 복잡도

반복 조건 확인 횟수 (최악):

- $C_{worst}(n) = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} = O(n^2)$

반복 조건 확인 횟수 (평균):

- $C_{avg}(n) = \frac{C_{worst}(n)}{2} = \frac{n(n+1)}{4} = O(n^2)$

반복 조건 확인 횟수 (최선, 거의 정렬된 경우):

- $C_{best}(n) = 1 + 1 + \cdots + 1 = n = O(n)$

QUICK_SORT vs HEAP_SORT

- 퀵 정렬은 단순해서 평균 실행 시간이 더 빠르다
- 힙 정렬은 최악의 경우에도 O(n log n)을 보장한다

하이브리드 정렬 (INTRO_SORT)

- 퀵 정렬의 평균 성능과 힙 정렬의 최악 성능 결합
- n이 작아지면 INSERTION_SORT로 전환
- C++의 표준 정렬 함수

INTRO_SORT 과정

1. 퀵 정렬로 분할 정복 수행

2. 재귀 깊이가 일정 수준 이상일 경우 힙 정렬로 전환

3. 데이터 수가 적으면 INSERTION_SORT

INTRO_SORT 시각화

- 예시: [5, 3, 2, 7, 1] 정렬 과정
- 삽입 정렬이 실행됨
- 최종 정렬 결과: [1, 2, 3, 5, 7]

INTRO_SORT

함수 INTRO_SORT(리스트 L, 인덱스 lo, 인덱스 hi,
정수 limit):

- 만약: hi - lo < 15이면

  INSERTION_SORT(L, lo, hi)
- 아니면 만약: limit == 0이면

  HEAP_SORT(L, lo, hi)
- 아니면:

  pivot = PARTITION(L, lo, hi)

  INTRO_SORT(L, lo, pivot - 1, limit - 1)

  INTRO_SORT(L, pivot + 1, hi, limit - 1)

MERGE SORT

- 반으로 나눠서 각각 정렬한 후, 정렬하며 합치는 방법

MERGE SORT 과정

1. 배열을 반으로 분할

2. 각 부분에 대해 재귀적으로 정렬

3. 정렬된 두 부분을 합침

MERGE SORT 시각화

- 예시: [5, 3, 2, 7, 1] 정렬 과정

  분할: [5,3] , [2,7,1]

  정렬: [3,5] , [1,2,7]

  합치기: [1,2,3,5,7]

MERGE_SORT

함수 MERGE_SORT(리스트 L, 인덱스 lo, 인덱스 hi):

- 만약: lo < hi이면

  mid = (lo + hi) / 2

  MERGE_SORT(L, lo, mid)

  MERGE_SORT(L, mid + 1, hi)

  MERGE(L, lo, mid, hi)

MERGE 1 / 4

함수 MERGE(리스트 L, 인덱스 lo, 인덱스 mid,
인덱스 hi):

- left_size = mid - lo + 1
- right_size = hi - mid

- left_q = 새로운 큐(left_size)
- right_q = 새로운 큐(right_size)

MERGE 2 / 4

- 반복: i는 lo부터 mid까지

  ENQUEUE(left_q, list->data[i])

- 반복: i는 mid + 1부터 hi까지

  ENQUEUE(right_q, list->data[i])

MERGE 3 / 4

- merged = 새로운 리스트(list->data, 0)
- 반복: !EMPTY(left_q) 그리고 !EMPTY(right_q)일 동안

    만약: FRONT(left_q) <= FRONT(right_q)이면

        ALIST_PUSH(merged, DEQUEUE(left_q))

    아니면:

        ALIST_PUSH(merged, DEQUEUE(right_q))

MERGE 4 / 4

- 반복: !EMPTY(left_q)일 동안

  ALIST_PUSH(merged, DEQUEUE(left_q))

- 반복: !EMPTY(right_q)일 동안

  ALIST_PUSH(merged, DEQUEUE(right_q))

- 삭제: left_q, right_q, merged

MERGE_SORT의 시간 복잡도

MERGE시 시간복잡도:

- 총 길이가 k인 두 배열을 합칠 때, 비교 k회, 대입 2k회
- MERGE 트리 한 층에서: 비교 n회, 대입 2n회
- MERGE 트리의 높이인 log n번 반복

총 비교 n log n회, 대입 2n log n회: O(n log n)

MERGE_ vs QUICK_ and HEAP_SORT

- MERGE_SORT는 값이 같은 데이터 순서가 보존된다
- 이미 정렬된 부분을 보존하며 합칠 수 있다
- MERGE시 추가 메모리 공간이 필요하다

하이브리드 정렬(POWER_SORT)

- MERGE 트리의 왼쪽 끝에서 MERGE 해가는 형태
- 앞에서부터 정렬된 구간을 찾고, MERGE 해가는 방법
- 정렬된 구간이 짧으면 INSERTION SORT로 늘림
- 정렬된 구간들은 스택에 보관
- 파이썬 정렬 함수에서 쓰임

COUNTING_SORT

- 데이터의 개수를 세어 정렬하는 방법

COUNTING_SORT 과정

1. 데이터 개수를 세는 배열 생성 및 초기화

2. 각 원소의 개수 세기

3. 출력 배열에 정렬된 원소 배치

COUNTING_SORT 시각화

- 예시: [5, 5, 2, 2, 1] 정렬 과정
- 최대값: 5
- 카운트 배열: [0, 1, 2, 0, 0, 2]
- 최종 정렬 결과: [1, 2, 2, 5, 5]

COUNTING_SORT

함수 COUNTING_SORT(리스트 L, 정수 max_value):

- count = 새로운 배열(max_value + 1) // 0으로 초기화
- 반복: value는 L의 모든 데이터

        count[value] += 1

- sorted = 새로운 리스트(L->data, 0)
- 반복: value는 0부터 max_value까지

        반복: count[value]번

                ALIST_PUSH(sorted, value)

- 삭제: count, sorted

COUNTING_SORT의 시간 복잡도

개수 배열 초기화의 시간복잡도:

- O(k)

데이터 개수 세기:

- O(n)

총 O(n + k)

요약

- heap sort: O(n log n)
- quick sort: 평균 O(n log n), 최악 O(n^2)
- insertion sort: 최선 O(n), 평균 및 최악 O(n^2)
- merge sort: O(n log n)
- counting sort: O(n + k)