


PART 1 알고리즘 기초

1-1 알고리즘 이란?

- 어떠한 문제를 해결하기 위해 정해놓은 일련의 절차
- 올바른 알고리즘
 - 어떠한 경우에도 실행결과가 똑같이 나오는 것

1-1-1. 조건문과 분기

· 분기

— 프로그램의 실행흐름을 다른 곳으로 변경하는 명령

1-1-2. 순서도 기호

· 데이터



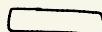
· 선



· 처리



· 단말



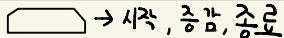
· 미리정의한 처리



· 판단



· 루프 범위



1-2 반복하는 알고리즘

· 반복구조 - 루프 (loop)

— 어떤 조건이 성립하는 동안 반복해서 처리

1-2-1 while 문

· 사전 판단 반복구조

1-2-2 for 문

· 변수가 하나만 있을 때

— 카운터용 변수

· 반복을 제어할 때 사용

/-2-3. 다중루프

- 반복문 중첩
- 변수
 - 파이썬: 객체에 연결된 이름
 - 전역 변수
 - 함수 외부에서 정의
 - 지역 변수
 - 함수 내부에서 정의

PART 2 기본 자료구조와 배열

2-1. 자료구조와 배열

- 배열
 - 물음 단위로 값을 저장
 - 원소
 - 저장된 객체
- 인덱스
 - 원소들의 순서
- 언팩
 - 배열의 원솟값들을 풀어 여러 변수에 대입

2-1-1. 인덱스

- 배열 객체 [i]
- 새로운 값 정의 불가능

2-1-2. 슬라이스

- 배열 객체 [시작:종료:증감]
- 값의 변경 유/무
 - 이터터 불 → 불가능
 - 뮤타불 → 가능
- 자료구조
 - 논리적인 관계로 이루어진 데이터 구성

2-2 배열이란?

· 스캔

- 배열 원소를 차례대로 살펴보는 방식

· 모듈 - py

- 하나의 스크립트 프로그램

- if __name__ == '__main__':

· 모듈이 직접 실행될 때 → TRUE

· 모듈이 임포트 될 때 → FALSE

· enumerate (이터레이션)

→ (인덱스, 값) 반복 → 투플

→ enumerate (이터레이션, i)

· (인덱스, 값) ⇒ 인덱스: i 부터 시작

· 이터레이션

- 반복 가능한 자료형

· 문자열, 리스트, 투플, 집합, 딕셔너리

2-2-1. 리스트 역순으로 정렬

· 정렬 값 반환

- 리스트 객체. reverse()

· 역순 정렬 리스트 생성

- = list(reversed(리스트 객체))

· 함수 사이에 인수 주고 받기

- 인수 - 유터보

· 함수 안에서 매개변수값 변경 → 객체 자체를 업데이트

- 인수 - 이유터보

· 함수 안에서 매개변수 값 변경 → 실제 인수에 영향 X

· 리스트의 원소와 복사

- 파일 → 리스트 내 각 원소들의 자료형을 다르게 할 수 있음

- 복사

· 얕은 복사 : 객체가 참조 자료형의 멤버를 포함 → 참조 값만 복사

· 깊은 복사 (=전체복사) : 참조 값과 참조하는 객체 자체를 복사 → 객체가 갖고 있는 모든 것 복사

PART 3 검색 알고리즘

3-1 검색 알고리즘 이란?

- 데이터 집합에서 원하는 값을 가진 원소를 찾아 내는 방식

3-1-1 검색과 키

- 키

- 검색조건이 주목한 어떤 항목

- 주로 데이터의 일부

3-1-2 검색의 종류

- 1) 배열 검색

- 선형검색 : 우측위로 늘어놓은 데이터 집합

- 이진검색 : 일정한 규칙으로 늘어놓은 데이터 집합

- 해시법 : 추가·삭제가 자주 발생하는 데이터집합

- 체인법, 오픈 주소법

- 2) 연결 리스트 검색

- 3) 이진 검색트리 검색

3-2 선형검색

- 순차검색

- 직선(선형) 모양으로 늘어선 배열에서 검색

- → 원하는 키값을 가진 원소를 찾을 때 까지 맨 앞부터 순서대로 검색

- 원소의 값이 정렬되지 않은 배열에서 검색에 사용되는 유일한 방법

3-2-1 종료조건

- . 검색실패

- 검색할 값을 찾지 못하고 맨 끝까지 간 경우

- . 검색성공

- 검색할 값과 같은 원소를 찾는 경우

3-2-2 보초법

- . 종료조건의 검사 이용(시간)을 높으로 줄임

- (원래 데이터) + 보초 (= 키값)

- > 인덱스 == len(원래데이터+보초) ⇒ 검색 실패

3-3 이진검색 - binary search

- 정렬된 배열 데이터 사용 - 오름차순
- 선행검색 대비 빠른 속도

3-3-1. 이진검색

- 반복

- 1) 배열의 중간값과 key 값의 크기 비교

. 중간값 : 2진수 \Rightarrow 10진수

. 중간값 < key

- 배열 \Rightarrow [중간값+1 ~ 끝값]

. 중간값 > key

- 배열 \Rightarrow [시작값 ~ 중간값-1]

- 종료조건

. 중간값과 key 값이 일치할 때

. 검색범위가 더 이상 없을 경우

. 비교 횟수

- 평균 : $\log_2 n$

3-3-2. 복잡도

- 알고리즘의 성능 평가 기준

시간 복잡도

- 실행하는데 걸리는 시간

공간 복잡도

- 메모리와 파일 공간의 필요한 정도

- 복잡도 표기 - Big O

- $O()$

. $| \log n \ n \ n \log n \ n^t \ n^3 \ n^k \ 2^n$

작다

크다

. $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$

3-4 해시법

- 정색

- 데이터의 추가·삭제에 효율적

3-4-1. 정렬된 배열에서 원소 추가하기

- 1) 추가할 자리를 찾기 위해 이진 정색법

- 2) 추가할 자리 이후의 모든 원소를 한 칸씩 뒤로 이동

⇒ O(n) 의 복잡도 소요

- 3) 추가할 자리에 새로운 원소 추가

3-4-2. 해시법 - hashing

- 데이터를 저장할 위치 = 인덱스를 간단한 연산으로 구함

- 해시값

- 데이터에 접근할 때 기준이 됨

- 해시 함수

- 키를 해시값으로 변환하는 과정

3-4-3. 해시충돌

- 저장 할 버킷(해시값)이 충복되는 현상

- 키 : 해시값 \Leftrightarrow n:1 관계

- 대처법

- 체인 법

- 해시값이 같은 원소를 연결 리스트로 관리

- 오픈 주소법

- 빈 버킷을 찾을 때 까지 해시를 반복

3-4-4. 체인법 (=오픈 해시법)

- 해시값이 같은 데이터 저장하기

- 연결 리스트에 의해 체인 모양으로 연결

- 1) Node 클래스 만들기

- 기체별 버킷을 나타냄

- Key : 키(임의의 자료형)

- Value : 값(임의의 자료형)

- Next : 뒤쪽 노드를 참조(Node형)

· 2) ChainedHash 클래스 만들기

- Capacity : 해시테이블의 크기 (배열 table 의 원소수)
 - 크기는 소수를 선호한다.
- table : 해시테이블을 저장하는 list 형 배열
- hash_value() 함수
 - 인수 key에 대응하는 해시값 반환
 - 해시값 : $key \% capacity$
 - key가 int 형인 경우와 아닌 경우를 구분

· 3) search() 함수

- 1. 해시함수를 사용하여 key \rightarrow 해시값 번환
- 2. 해시값을 인덱스로 하는 버킷에 주목
- 3. 버킷이 참조하는 연결리스트를 스캔
 - key와 같은 값 발견 \rightarrow 검색성공
 - 맨 끝까지 발견 X \rightarrow 검색실패

· 4) 원소를 추가하는 add() 함수

- 1. 해시함수를 사용하여 key \rightarrow 해시값 번환
- 2. 해시값을 인덱스로 하는 버킷에 주목
- 3. 버킷이 참조하는 연결리스트 선형검색
 - key와 같은 값 발견 \rightarrow 추가실패
 - 맨 끝까지 발견 X \rightarrow 추가 성공

· 5) 원소를 삭제하는 remove() 함수

- 1. 해시함수를 사용하여 key \rightarrow 해시값 번환
- 2. 해시값을 인덱스로 하는 버킷에 주목
- 3. 버킷이 참조하는 연결리스트 선형검색
 - key와 같은 값 발견 \rightarrow 값 삭제
 - 맨 끝까지 발견 X \rightarrow 삭제 실패

· 6) 원소를 출력하는 dump() 함수

- 해시 table의 내용을 한꺼번에 출력

3-4-5. 오픈 주소법 (= 단 히 해시법)

- 충돌 발생 → 재해시를 수행하여 빈 버킷을 찾는 방법

- 1) 원소 추가하기

- 빈 버킷을 찾을 때까지 헤시함수 재정의 후 재해시

- 2) 원소 삭제하기

- 각 버킷에 속성 부여

- 문자 → 데이터가 저장되어 있음

- - → 비어 있음

- * → 삭제 완료

- 3) 원소 검색하기

- 부여된 속성에 의해

- 4) 원소 출력하기 - dump()

PART4 스택과 큐

- 데이터를 임시 저장하는 기본 자료구조

4-1 스택이란?

4-1-1. 스택 알아보기

- LIFO 방식

- 데이터의 입력과 출력 순서

- 후입 선출

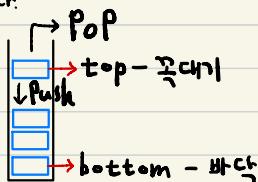
- 가장 나중에 넣은 것을 먼저 꺼낸다.

- Pop

- 스택에서 데이터를 꺼내는 작업

- Push

- 스택에 데이터를 넣는 작업



4-1-2. 스택 구현하기

- 크기가 결정된 고정길이 스택 만들기

- 스택 배열 - `stk`

- 데이터를 저장하는 스택 본체인 `list`형 배열

• 스택 크기 - Capacity

- 스택의 최대 크기를 나타내는 int형 정수
- $= \text{len}(\text{stk})$

• 스택 포인터 - ptr

- 스택에 쌓여 있는 데이터의 개수를 나타내는 정수

• 예외 처리 클래스

- Empty

- 비어 있는 스택에 pop 또는 peek(들여다보기) 할 때 예외 처리

- Full

- 가득 차 있는 스택에 push 할 때 예외 처리

• 초기화 함수 - `__init__()`

- 스택 본체 : `stk = [None] * Capacity`
- 스택의 크기 : `Capacity`
- 스택 포인터 : `ptr = 0`

• 스택 내의 데이터 개수 함수 - `__len__()`

- `return self.ptr`

• 스택이 비어 있는지 확인하는 함수 - `is_empty()`

- `return self.ptr == 0`

• 스택이 가득 차 있는지 확인하는 함수 - `is_full()`

- `return self.ptr >= self.capacity`

• 스택에 데이터를 push 하는 함수 - `push(self, value)`

- If 스택이 가득 차 있는 경우

- 예외 처리

- else

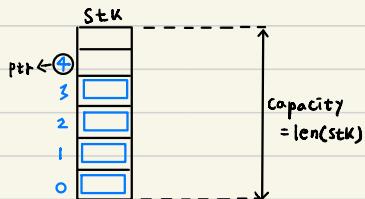
- 전달 받은 value \rightarrow 스택에 저장

- 포인터 +1

• 스택에 있는 데이터를 pop 하는 함수 - `pop(self)`

- If 스택이 비어 있는 경우

- 예외 처리



- else

- 포인터 - i
- return self.stk[self.ptr]
- 데이터를 들여다 보는 함수 - peek(self)
 - If 스택이 비어 있는 경우
 - 예외처리

- else

- return self.stk[self.ptr-1]
- 스택의 모든 데이터 삭제 함수 - clear(self)
 - self.ptr = 0
- 데이터를 검색하는 함수 - find(self, value)
 - 꼭대기 쪽부터 선형검색
 - 검색실패시 → return -1
- 데이터 개수를 세는 함수 - count(self, value)
 - 바닥쪽 부터 선형검색
 - 정색성공시 결과값 + 1
- 데이터가 포함되어 있는지 확인하는 함수 - __contains__(self, value)
 - return self.count(value)
 - 있으면 → True 반환
 - 없으면 → False 반환
- 스택의 모든 데이터를 출력하는 함수 - dump(self)
 - If self.is_empty()
 - print('스택이 비어 있습니다.')
 - else
 - print(self.stk[:self.ptr])

- 딘더(dunder) 함수

- __ 항수 이름 __ → double underscore
- __ 항수 이름 __ () → 항수 이름 (액체)
로 간단히 표현

— 데크 - deque

· 맨 앞과 맨 끝 양쪽에서 원소를 추가·삭제하는 구조

4-2 큐란?

4-2-1. 큐 알아보기

· FIFO 방식

— 데이터의 입력과 출력순서

— 선입선출

• 가장 먼저 넣은 데이터를 가장 먼저 꺼내는 방식

· enqueue - 인큐

— 큐에 데이터를 추가하는 작업

· dequeue - 디큐

— 큐에 있는 데이터를 꺼내는 작업

• front - 프런트

— 데이터를 꺼내는 쪽

• rear - 리어

— 데이터를 넣는 쪽

4-2-2. 배열로 큐 구현하기

• enqueue

— 처리의 복잡도 - $\bigcirc(1)$

• 비고적적은 비용

• dequeue

— 처리의 복잡도 - $\bigcirc(n)$

• 맨 뒤 남아있는 원소들을 한 칸씩 앞으로 옮기는 과정

• \rightarrow 프로그램의 효율성 기대 X

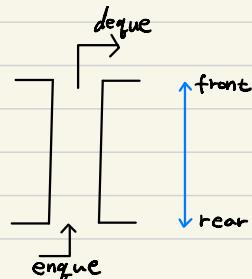
4-2-3. 링 버퍼 와 큐

• dequeue 할 때 배열안의 원소를 옮기지 않음

• front, rear \rightarrow 논리적인 순서

\rightarrow 물리적인 순서 X

• 처리의 복잡도 - $\bigcirc(1)$



4-2-4. 링 버퍼로 큐 구현하기

- 예외 처리 클래스

- Empty (Exception)

- 비어 있는 큐에 `dequeue()`, `peek()` 함수 호출 시

- Full (Exception)

- 가득 차 있는 큐에 `enqueue()` 함수 호출 시

- 초기화 함수 - `__init__(self, capacity)`

- `que` : 큐의 `list` 형 배열

- `Capacity` : 큐의 최대 크기, `int` 형

- `front, rear` : 맨 앞, 뒤 원소를 나타내는 인덱스

- `no` : 현재 큐에 쌓여 있는 데이터 개수

- 큐에 있는 데이터 개수 반환 - `__len__(self)`

- `return self.no`

- 큐가 비어 있는지 확인 - `is_empty(self)`

- `return self.no <= 0`

- 큐가 가득 차 있는지 확인 - `is_full(self)`

- `return self.no >= self.capacity`

- 데이터를 큐에 넣는 함수 - `enqueue(self, x)`

- if 큐가 가득 차 있는 경우

- 예외 처리

- else

- `que[rear]`에 데이터 넣음

- `rear, no +`

- if `rear == capacity`

- `rear = 0`

- 데이터를 큐에서 꺼내는 함수 - `dequeue(self)`

- if 큐가 비어 있을 경우

- 예외 처리

- else

- `que[front]`에 저장된 값을 꺼내어 `x`에 저장

- front + 1, no - 1
- if front == capacity
 - front = 0
- return X
 - X 값 반환
- 데이터를 들여다 보는 함수 - peek(self)
 - if 큐가 비어 있는 경우
 - 예외처리
 - else
 - return self.que[self.front]
 - 맨 앞의 데이터 반환
- 검색하는 함수 - find (self, value)
 - 맨 앞부터 선형검색
 - 맨 앞 = front
 - 인덱스 i → (i + front) % Capacity
 - 검색 실패시 → return -1
 - 데이터 개수를 세는 함수 - count (self, value)
 - 맨 앞부터 선형검색
 - 맨 앞 = front
 - 인덱스 i → (i + front) % Capacity
 - 검색성공시 결과값 + 1
 - 데이터가 큐에 있는지 확인하는 함수 - __contains (self, value)
 - return self.count (value)
 - 큐의 전체 원소를 삭제하는 함수 - clear (self)
 - self.no = self.front = self.rear = 0
 - enqueue, dequeue는 no, front, rear의 값을 바탕으로 수행
 - ⇒ 실제 que의 원소값 변경 X
 - 큐의 전체 데이터를 출력하는 함수 - dump (self)
 - if self.is_empty()
 - print('큐가 비어있습니다.')
 - else
 - print (self.que[:self.ptr])

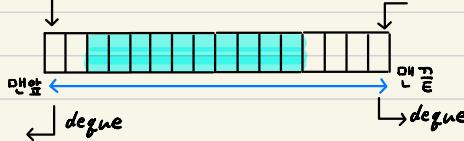
- else

. `print(self.que[(i+self.front)%self.capacity], end=' ')`

- deque - 양방향 대기열 덱

- 맨 앞과 맨 끝 양쪽에서 원소를 추가·삭제 하는 구조

- . `enqueue`



- 링 버퍼의 활용

- . 가장 최근에 들어온 n 개 만 저장하고 나머지는 버림.

- . 봉법

- 크기가 n 인 링 버퍼 생성

- $Cnt = 0$

- . 정수를 입력 받은 개수

- 반복

- . 링 버퍼 [$Cnt \% n$] = 입력값

- . $Cnt + 1$ → $Cnt \geq n \Rightarrow$ 더이 쓰임

- . If 입력 == N

- break

- 링버퍼에 남아있는 수 출력

- . $i = Cnt - 1$

- . If $i < 0$

- $i = 0$

- . 반복 $i < Cnt$

- `print(i+1 번째 = 링버퍼[i%N])`

- $i + 1$

입력 순서대로

0부터 차례대로

PART 5 재귀 알고리즘

5-1 재귀 알고리즘의 기본

5-1-1. 재귀

- 어떠한 이벤트에서 자기자신을 포함
- 다시 자기 자신을 사용하여 정의되는 경우

5-1-2. 팩토리얼

- $n!$ ($n =$ 양의 정수)
 - $0! = 1$
 - $\text{if } n > 0$
 - $n! = n \times (n-1)!$

• 재귀 호출

- '자기 자신과 똑같은 함수'를 호출

• 직접재귀와 간접재귀

- **직접재귀**
 - '자기 자신과 똑같은 함수'를 호출
- **간접재귀**
 - 다른 함수를 통해 자신과 똑같은 함수를 호출

5-1-3. 유clidean 호제법

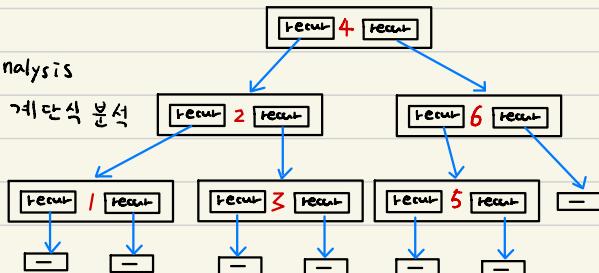
- 최대 공약수 - $\text{gcd}(x, y)$
 - $\text{if } y == 0$
 - $\text{return } x$
 - else
 - $\text{return } (\text{gcd}(y, } x \% y))$

5-2 재귀 알고리즘 분석

5-2-1. 두 가지 분석 방법

- 하향식 분석 - **top-down analysis**

- 가장 위쪽부터 아래로 계단식 분석



· 상향식 분석 - bottom-up analysis

- 아래쪽 부터 쌓아 올리며 분석하는 방법

· ex) recur(n)

if $n > 0$:

 recur(n-1)

 print(n)

 recur(n-2)



$\Rightarrow \text{recur}(4) = ?$

$\text{recur}(1) : \text{아무것도 하지않음}$

$\text{recur}(0) : \text{아무것도 하지않음}$

$\text{recur}(1) : \text{recur}(0) 1 \text{ recur}(0) \rightarrow 1$

$\text{recur}(2) : \text{recur}(1) 2 \text{ recur}(0) \rightarrow 1$

$\text{recur}(3) : \text{recur}(2) 3 \text{ recur}(1) \rightarrow 123 1$

$\therefore \text{recur}(4) : \text{recur}(3) 4 \text{ recur}(2) \rightarrow 1231412$

5-2-2 재귀 알고리즘의 비재귀적 표현

- 단점: 성능 저하가 심하고 메모리를 많이 소모한다

· 꼬리 재귀 제거

- 재귀 호출이 끝난 이후 현재 함수 블록 내에서 추가 연산
을 할 필요가 없도록 구현하는 형태

· 재귀 제거

5-3 하노이의 탑

- 쌓아 놓은 원반을 최소 횟수로 옮기는 알고리즘

5-3-1. 하노이의 탑

- 작은 원반이 위, 큰원 반이 아래에 위치하는 규칙을 유지하면서

기둥 3개를 이용하여 원반을 옮기는 문제



1) 바닥에 있는 원반을 제외한 나머지 원반을
2번 기둥으로 옮김



2) 바닥에 있는 원반을 1기둥 \rightarrow 3기둥으로 옮김



3) 2기둥에 있는 나머지 원반을 3기둥으로 옮김



5-4 8퀸 문제

5-4-1. 8퀸 문제

- 8개의 퀸이 서로 공격하여 잡을 수 없도록 8×8 체스판에 배치하세요.

⇒ 92 가지의 해

5-4-2. 퀸 배치하기

· 규칙 1

- 각 열에 퀸을 1개만 배치

· 규칙 2

- 각 행에 퀸을 1개만 배치

5-4-3. 분기 작업으로 규칙 1 해결하기

· 분기

- 나누는 작업

· 분기 작업

- 차례대로 가지가 뻗어 나가듯이 배치 작업을 열거하는 방법

· 분할정복법 (=분할 해결법)

- 1) 큰 문제를 작은 문제로 분할

- 2) 작은 문제 풀이법을 결합하여 전체 풀이법을 얻는 방법

- 주의 할점

- 2) 단계를 쉽게 할 수 있도록 문제를 분할

- ex)

· 하노이 탑 8퀸 문제

5-4-4. 한정작업과 분기 한정법 → 규칙 2 해결

· 한정작업

- 필요하지 않은 분기를 없애서 필요한 조합을 열거하지 않는 방법

- ex)

- 이전 열에 퀸이 있는 행은 제외

· 분기 한정법

- 분기 작업과 한정작업을 조합하여 문제를 풀이하는 방법

5-4-5. 8퀸 문제 해결하기

· 대각선 배치문제 ($/$, \backslash) 까지 고려하여 분기 한정법

- 92가지 조합

PART 6 정렬 알고리즘

6-1 정렬 알고리즘

6-1-1 정렬 이란?

- 항목간의 대소관계에 따라 데이터 집합을 일정한 순서로 바꾸어 늘어놓는 작업
- 오름차순 - *ascending*
 - 작은 \rightarrow 큰
- 내림차순 - *descending*
 - 큰 \rightarrow 작은
- 정렬 \leftarrow 안정적인 - 정렬후 값이 같은 원소의 순서유지
불안정적인 - 정렬후 값이 같은 원소의 순서유지 보장X
- 정렬 \leftarrow 내부정렬 - 하나의 배열에서 작업 가능
외부정렬 - 하나의 배열에서 작업 불가능
 - 내부정렬 응용

6-2 버블 정렬

- 단순 교환 정렬
- 이웃한 두 원소 비교후 필요에 따라 교환을 반복

6-2-1. 버블 정렬 알아보기

- 파스 (pass)
 - 일련의 비교교환 하는 과정
 - 원소가 n개일때, n-1번 수행
 - 원소가 n개일때, n-1번 수행

6-2-2. 알고리즘 개선

- 1) 파스에서 더 이상 교환이 이루어지지 않을 시
 \rightarrow 정렬을 완료 \rightarrow 중단 (break)
- 2) 이미 정렬된 원소를 제외한 나머지에만 파스 수행
 \rightarrow 이전 파스에서 마지막 교환한 원소들 중 오른쪽 원소까지 새로운 범위

6-2-3. 셰이커 정렬

- 홀수파스 : 가장 작은 원소 \rightarrow 맨 앞 \rightarrow 파스의 스캔방향을 앞·뒤로 바꿈
- 짝수파스 : 가장 큰 원소 \rightarrow 맨 뒤

6-3 단순 선택정렬

- 가장 작은 원소 부터 선택해 알맞은 위치로 옮기는 작업 반복

6-3-1. 단순 선택정렬 알아보기

교환 과정

- 1) 아직 정렬하지 않은 부분에서 가장 작은 원소 선택

2) 선택한 원소 \leftrightarrow 아직 정렬하지 않은 부분에서 가장 앞의 원소

비교횟수

$$-(n^2-n)/2$$

불안정적인 정렬

- 서로 이웃하지 않는 떨어져 있는 원소를 교환

6-4 단순 삽입정렬

- 주목한 원소보다 더 앞쪽에서 알맞은 위치로 삽입하여 정렬

6-4-1. 단순 삽입정렬 알아보기

정렬되지 않은 부분의 맨 앞원소를 정렬된 부분의 알맞은 위치에 삽입

6-4-2. 선택한 원소를 알맞은 위치에 삽입하는 과정

반복제어

- 종료조건 1. 정렬된 배열의 맨앞에 도달했을 경우

or (종료조건 2. 선택한 원소보다 작거나 같은 한칸앞의 원소를 발견할 경우

→ 계속조건 1. 정렬된 배열의 현재 인덱스 > 0

and (계속조건 2. 한칸앞의 원소 $>$ 선택한 원소

6-4-3. 단순정렬 알고리즘의 시간 복잡도

- 버블, 선택, 삽입 $\rightarrow O(n^2)$

- 효율이 좋지 않음

이진 삽입정렬

- 이미 정렬을 마친 배열을 제외하고 원소를 삽입해야 할 위치를 검사

6-5 셀 정렬

- 단순 삽입 정렬 보완

6-5-1. 단순 삽입정렬의 문제점

- 장점

- 정렬 완료된 또는 거의 끝나가는 상태에서 속도가 빠름

- 단점

- 삽입할 위치가 멀리 있으면 이동 횟수가 많아짐

6-5-2. 셀 정렬 알아보기

- 1) 배열의 원소들을 그룹으로 나눠 각 그룹별로 정렬을 수행

- 그룹 안의 원소의 개수는 단계별로 서로 배수가 되지 않도록 결정

- 2) 정렬된 그룹을 합침

→ 1, 2) 작업을 반복

- 전체적으로 원소의 이동 횟수 감소

- 정렬 횟수는 늘어남

6-6 퀵 정렬

- 가장 빠른 정렬 알고리즘

- 널리 사용됨

6-6-1. 퀵 정렬 알아보기

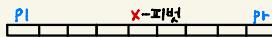
- 기준(피벗)에 의해서 그룹을 나눔

- 피벗 - 중심축

- 임의 선택 가능

- 선택된 피벗은 2개로 나눈 그룹 어디에 넣어도 상관X

6-6-2. 배열을 두 그룹으로 나누기



- $p_i \geq X$ 인 원소를 찾을 때 까지 오른쪽으로 스캔

- $p_i < X$ 인 원소를 찾을 때 까지 왼쪽으로 스캔

- 조건을 만족하는 원소 발견 시 정지후 p_i 과 p_r 값 교환

- p_i 과 p_r 이 교차

- 그룹을 나누는 과정 끝



- 피벗 이하인 그룹 : $0, 1, \dots, p_i - 1$

- 피벗 이상인 그룹 : $p_i + 1, \dots, n-1$

- if 배열 나누기후 $p_i > p_{i+1}$ 일때,

- 피벗과 일치하는 그룹

6-6-3. 퀵정렬 만들기

- 분할정복 알고리즘
→ 재귀 호출 사용

- `qSort (배열, left, right)`
 - 배열을 두개의 그룹으로 나누는 함수
- `quick-sort (배열)`
 - `qSort (배열, 0, len(배열)-1)`

6-6-4. 비재귀적인 퀵 정렬 만들기

- 스택을 사용하여 구현
 - `qSort (배열, left, right)`
- 스택 생성
 - 크기: 배열의 크기

- 스택 푸시
 - 나눌 범위의 맨 앞, 맨 끝 인덱스를 조합한 튜플 스택
 - 반복으로 배열 나누기 - 스택이 비어있지 않은 동안
 - 기존 스택에 있는 범위로 배열을 나눔
 - If 나누기 전 맨 앞 < 나누기 후 맨 끝
 - 푸시 (, )
 - If 나누기 후 맨 앞 < 나누기 전 맨 앞
 - 푸시 (, )

- 스택의 크기
 - 배열을 스택에 푸시하는 순서에 따라서 다름

• 규칙 1

- 원소수가 많은 쪽의 그룹을 먼저 푸시

• 스택에 쌓이는 데이터의 최대 개수 < $10^2 n$

- 원소수가 적은 쪽의 그룹을 먼저 푸시

스택에 넣고, 꺼내는 횟수는 같지만, 동시에 쌓이는 최대 데이터 개수는 다르다.

스택의 크기

• 규칙 2 > 규칙 1

6-6-5. 피벗 선택하기

- 전체에서 중앙값으로 하는게 이상적

— But, 중앙값을 구하기 위해 많은 계산시간이 걸림

→ 피벗을 선택하는 의미 X → 다른 방법 필요

방법 1.

발견

- I^f 원소수 ≥ 3

· 임의의 원소 3개를 꺼내 중앙값 → 피벗

방법 2.

— 1) 배열의 맨앞, 가운데, 맨끝 원소를 정렬

2) 가운데 원소와 맨끝-i 원소를 교환

3) 맨끝-i ⇒ 피벗, 맨앞+i ~ 맨끝-2 ⇒ 나눌 대상

6-6-6. 쿠적정렬의 시간복잡도

- $\bigcirc(n \log n)$

— 배열의 초기값 아 피벗을 선택하는 방법에 따라 달라짐

— 원소수가 적은 경우에는 빠른 알고리즘이 아님

6-7

병합정렬

- 배열을 앞 뒷 부분의 두 그룹으로 나누어 정렬 후 병합

6-7-1. 정렬을 마친 배열의 병합

- 각 배열에서 주목하는 원소의 값을 비교하여 작은쪽의 원소를 꺼내 새로운 배열에 저장

— 저장 후 두 배열 중 하나에 남은 원소를 새로운 배열에 저장

- 시간복잡도 — 병합

— $\bigcirc(n)$

6-7-2. 병합정렬 만들기

· 병합정렬

— 정렬을 마친 배열의 병합을 이용하여 분할정복법에 따라 정렬하는 알고리즘

- 시간복잡도 — 병합정렬

— $\bigcirc(n \log n)$

6-8 힙 정렬

- 선택정렬 응용

6-8-1. 힙 정렬 알아보기

- 힙

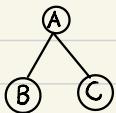
- 완전 이진트리

- 부모의 값 \geq 자식의 값
 - 부모의 값 \leq 자식의 값
- 두 조건 중 하나를 항상 만족

- 부분 순서트리

- 형제간의 대소관계 일정 X

- 트리



- (A) - 루트, (B), (C)의 부모노드
- (B), (C) - (A)의 자식노드
- (B), (C)는 서로 형제노드

- 완전 이진 트리

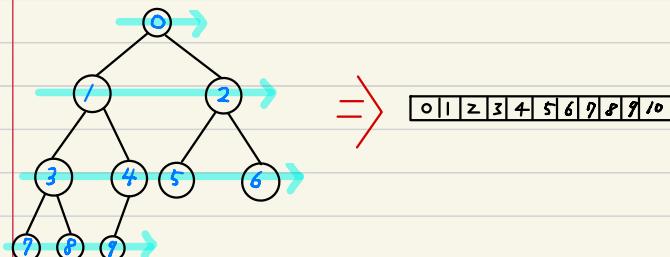
- 완전

- 부모는 왼쪽 자식부터 추가하여 모양유지

- 이진

- 자식의 최대개수 = 2

- 힙 \rightarrow 배열



- 자기 자신 $\alpha[i]$ 에 대해서

- 부모노드 - $\alpha[(i-1)//2]$

- 자식노드 \swarrow 왼쪽 - $\alpha[i*2 + i]$

오른쪽 - $\alpha[i*2 + 2]$

6-8-2. 힙 정렬의 특징

- '힙에서 최댓값 - 루트' 특성을 이용

— 1) 힙에서 최댓값인 루트를 꺼냄 → 새로운 정렬 배열의 맨 끝에 위치

— 2) 남은 부분을 힙으로 만듬

— 위의 과정을 반복

- 루트를 삭제한 힙의 재구성

— 1) 마지막 원소를 루트로 이동

— 2) 루트의 자식 노드 중 큰 값과 위치를 바꿈

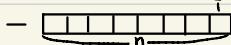
• 2) 과정을 반복

— 자식의 값이 각을 대까지

— 자식이 없는 리프가 될 때 까지

6-8-3. 힙 정렬 알아보기

- 힙으로 만든 배열 α



- 방법

— 1) $i = n-1$ 로 초기화

2) $\alpha[i] = \text{루트}$ 와 $\alpha[i]$ 교환 → 힙의 최댓값 $\Rightarrow \alpha[i]$

3) $\alpha[0] \sim \alpha[i-1]$ 로 힙 재구성

4) i 값을 1 감소시켜 0이 되면 종료

• 그렇지 않으면 2)로 돌아감

6-8-4. 배열 \rightarrow 힙

- 가장 아래 부분의 오른쪽 서브트리부터 상향식으로 힙 변환을 진행하여

전체 배열 \rightarrow 힙으로 변경

6-8-5. 힙 정렬의 시간 복잡도

- $\mathcal{O}(n \log n)$

— $\mathcal{O}(n)$ - 최댓값인 원소를 선택

$\mathcal{O}(\log n)$ - $\alpha[0] \sim \alpha[i-1] \rightarrow$ 힙 재구성

6-9 도수 정렬

- 분포수 세기 정렬
- 원소의 대소 관계 판단 X

6-9-1. 도수 정렬 알아보기

- 단계

- 1) 도수 분포표 만들기

- 배열 =

--	--	--	--	--	--

 $\rightarrow \min, \max$

→ 도수분포표

- 배열

- 크기: $\max - \min + 1$

- 도수분포표 [원소] + 1

2) 누적 도수분포표 만들기

- 누적 도수분포표 [i] = 누적 도수분포표 [i] + 도수분포표 [i-1]

3) 작업용 배열 만들기

- 배열 [i] = 작업용 배열 [누적 도수분포표 [배열[i]-1]]

4) 배열 복사하기

- 배열 [i] = 작업용 배열 [i]

PART 7 문자열 검색

- 어떤 문자열 안에 찾고자 하는 문자열이 포함되어 있는지 검사, 만약 포함되어 있다면 위치 반환

- 텍스트

- 검색되는 쪽의 문자열

- 파악

- 찾아내는 문자열

7-1 브루트 포스법

7-1-1. 브루트 포스법 알아보기

- 단순법

- 선형검색을 단순하게 확장

- 효율이 좋지 않음
 - 이미 검사한 위치를 기억하지 못함

- 방법

- 텍스트 :

A	B	A	B	C	D	E
---	---	---	---	---	---	---

- 패턴 :

A	B	C
---	---	---

→ a 1.

A	B	A	B	C	D	E
---	---	---	---	---	---	---

A	B	C
---	---	---

 → 1번 째 문자 일치

2.

A	B	A	B	C	D	E
---	---	---	---	---	---	---

A	B	C
---	---	---

 → 2번 째 문자 일치

3.

A	B	A	B	C	D	E
---	---	---	---	---	---	---

A	B	C
---	---	---

 → 3번 째 문자 불일치

b 4.

A	B	A	B	C	D	E
---	---	---	---	---	---	---

A	B	C
---	---	---

 → 1번 째 문자 불일치

c 5.

A	B	A	B	C	D	E
---	---	---	---	---	---	---

A	B	C
---	---	---

 → 1번 째 문자 일치

6.

A	B	A	B	C	D	E
---	---	---	---	---	---	---

A	B	C
---	---	---

 → 2번 째 문자 일치

7.

A	B	A	B	C	D	E
---	---	---	---	---	---	---

A	B	C
---	---	---

 → 3번 째 문자 일치

⇒ return (4-2) = 모두 일치시, 텍스트 검색 꽂 인덱스 - 패턴 검색 꽂 인덱스

- 멤버십 연산자 - 파이썬

- 패턴 in 텍스트

- 존재 → TRUE, 없음 → FALSE

- 패턴 not in 텍스트

- in 기호 과 와 반대

II-2 KMP 법

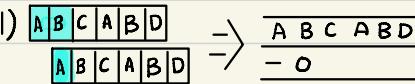
- 이전에 검사한 결과를 효율적으로 사용

- 브루트 포스법 단점 개선

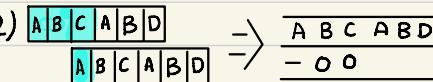
7-2-1. KMP법 알아보기

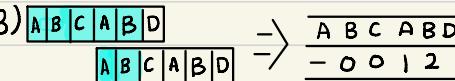
- 텍스트와 패턴안에서 겹치는 문자열을 찾아내 검사를 다시 시작할 위치를 구하여 패턴의 이동을 되도록이면 크거나하는 알고리즘
- '몇 번째 문자부터 다시 검색할지' 값을 표로 만들기
- 사용할 표 만들기
 - 건너뛰기 표

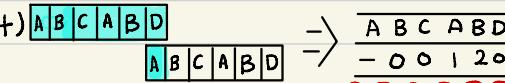
• 패턴과 패턴을 서로 겹치도록 두고 검사를 시작할 곳을 계산

• ex) 1)  $\rightarrow \begin{array}{c} \overline{A B C A B D} \\ \underline{- O} \end{array}$

- 파란색 부분이 일치하는지 검사

2)  $\rightarrow \begin{array}{c} \overline{A B C A B D} \\ \underline{- O O} \end{array}$

3)  $\rightarrow \begin{array}{c} \overline{A B C A B D} \\ \underline{- O O 1 2} \end{array}$

4)  $\rightarrow \begin{array}{c} \overline{A B C A B D} \\ \underline{- O O 1 2 0} \end{array}$

- 텍스트를 스캔하는 커서는 앞으로 갈 땐 뒤로 돌아오지 않음

- 브루트 포스법에 없는 특징

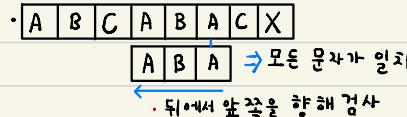
7-3 보이어·무어 법

- 실제 문자열 검색 알고리즘에서 널리 사용하는 알고리즘

7-3-1. 보이어·무어 법 알아보기

• 방법

- 패턴의 끝 문자에서 시작하여 앞쪽을 향해 검사 수행



- 일치하지 않는 문자 발견하면 미리 준비한 표를 바탕으로 패턴이 이동하는 값 결정

• 표 만들기

— 패턴에 포함되지 않는 문자를 만날 경우

• 패턴 이동량 = 패턴의 길이

— 패턴에 포함되는 문자를 만날 경우

• 패턴 이동량

= 패턴의 길이 - 마지막에 나오는 문자의 인덱스 - 1

. 패턴 안에 중복되지 않는 몇 글자 일경우

— 패턴 이동량 = 패턴의 길이

- ex) 패턴 → 'ABAC'

A	B	C	나머지 문자
1	2	4	4
↑	↑	↓	4-1-1
↓			4-2-1

• 보이어 우어법 예시

— 0 1 2 3 4 5 6 7 8
텍스트: A D E C C A B A C

패턴: A B A C → 일치

→ 불일치 = 패턴 이동량 = E → 4

. 현재 주목중인 문자 인덱스 2 $\xrightarrow{+4}$ 6 이동

0 1 2 3 4 5 6 7 8
A D E C C A B A C
A B A C → 불일치 : 패턴 이동량 = B → 2

0 1 2 3 4 5 6 7 8
A D E C C A B A C

A B A C →
} 일치 \Rightarrow index = 5에서 return:

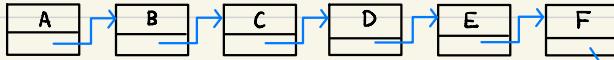
PART 8 리스트

- 데이터에 순서를 매겨 늘어놓은 자료구조

8-1 연결리스트

8-1-1. 연결리스트 알아보기

- 머리노드



— 데이터를 사슬처럼 연결

꼬리노드

뒤쪽노드를 가리키는 포인터

- 노드

— 연결리스트에서 각각의 원소

— 구성

- 데이터

- 포인터

— 뒤쪽 노드를 참조하는

— 종류

- 머리노드

— 맨 앞의 노드

- 꼬리노드

— 맨 끝의 노드

- 앞쪽 노드, 뒤쪽노드

— 특정 노드의 앞, 뒤

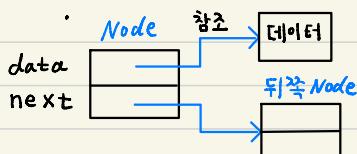
8-1-2. 배열로 연결리스트 만들기

- 단점

— 데이터를 삽입·삭제함에 따라 데이터를 옮겨야 하므로 효율적이지 않음

8-2 포인터를 이용한 연결리스트

8-2-1. 포인터로 연결리스트 만들기



- 자기참조형

— 자신과 같은 클래스 형의 인스턴스를 참조하기

위한 참조용 필드 next가 있는 구조

8-3 커서를 이용한 연결리스트

8-3-1. 커서로 연결리스트 만들기

· 커서

- int형 정수값인 인덱스로 나타낸 포인터
- 머리노드를 나타내는 head
- 머리노드의 인덱스값 소유
- 꼬리노드의 뒤쪽 커서
- - |

8-3-2. 배열안에 비어 있는 원소 처리하기

· 노드 삽입

· 저장장소

- 배열 안에서 가장 끝쪽에 있는 인덱스의 위치

· 노드 삭제

- 배열 안에 빈 공간이 생김

· 삭제 여러번 반복

→ 배열안에 빈 공간이 많이 생김 → 관리 필요

8-3-3. 프리 리스트

- 삭제된 레코드 그룹을 관리할 때 사용하는 자료구조
- 연결리스트 + 프리리스트

8-4 원형 이중 연결리스트

8-4-1. 원형리스트 알아보기

- 연결리스트의 꼬리노드가 다시 머리노드를 가리키는 모양

8-4-2. 이중 연결리스트

· 연결리스트의 단점

- 앞쪽 노드를 찾기 어렵다는 것

- → 이중 연결리스트에서 단점 극복

· 노드

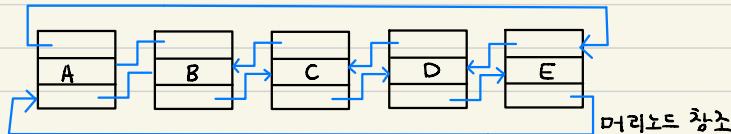
· data

- 뒤쪽노드, 앞쪽노드에 대한 각각의 포인터

8-4-3. 원형 이중 연결 리스트

- 원형리스트 + 이중 연결 리스트

head 꼬리노드참조



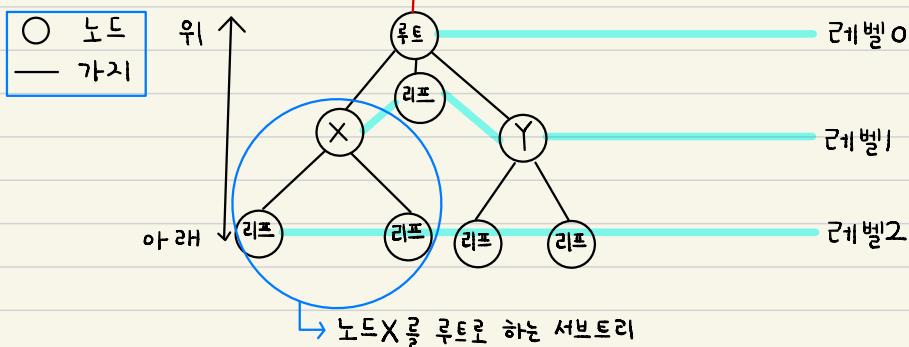
PART9 트리

9-1 트리구조

- 데이터 사이의 계층 관계를 표현하는 구조

9-1-1. 트리구조 와 관련 용어

- 트리 구조



- 루트

- 트리의 가장 위쪽에 있는 노드

- 루트는 트리에 1개만 존재

- 리프

- 트리의 가장 아래쪽에 있는 노드

- 단말노드, 외부노드

- 가지가 더 이상 뻗어나갈 수 없는 마지막노드

- 비 단말노드

- 내부노드

- 리프를 제외한 모든 노드

- 자식
 - 어떤 노드와 가지로 연결되었을 때 아래쪽 노드

- 부모
 - 어떤 노드와 가지로 연결되었을 때 가장 위쪽 노드

- 형제
 - 부모가 같은 노드

- 조상
 - 어떤 노드에서 위쪽으로 가지를 따라가면 만나는 모든 노드

- 자손
 - 어떤 노드에서 아래쪽으로 가지를 따라가면 만나는 모든 노드

- 레벨
 - 루트에서 얼마나 멀리 떨어져 있는지를 나타내는 것

- 차수
 - 각 노드가 갖고 있는 자식의 수
 - n진 트리
 - 모든 노드의 차수가 n이하인 트리

- 높이
 - 루트에서 가장 멀리 있는 리프 까지의 거리
 - = 리프 레벨의 최댓값

- 서브트리
 - 어떤 노드를 루트로하고 그 자손으로 구성된 트리

- 빙트리
 - = 널트리
 - 노드와 가지가 전혀 없는 트리

9-1-2. 순서트리와 무순서트리

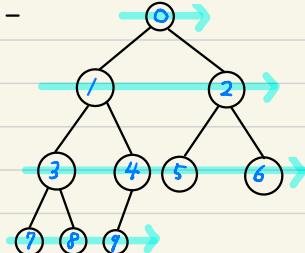
- 순서트리
 - 형제노드의 순서관계 있음

- 무순서트리
 - 형제노드의 순서관계 없음

9-1-3. 순서트리의 검색

• 너비 우선 검색

- 둘 꼭 우선 검색, 가로검색, 수평검색



• 깊이 우선 검색

- 세로검색, 수직검색

- 리프에 도달할 때까지 아래쪽으로 내려가면서 검색하는 것을 우선
- 리프에 도달 후, 다시 부모노드로 돌아가고 그 뒤 다시 자식노드로 돌아감
- 전위 순회

• 노드방문 → 왼쪽노드 → 오른쪽노드

– 중위순회

• 왼쪽자식 → 노드방문 → 오른쪽노드

– 후위순회

• 왼쪽자식 → 오른쪽자식 → 노드방문

9-2 이진트리와 이진검색트리

9-2-1. 이진트리 알아보기

- 노드가 왼쪽자식과 오른쪽자식만을 갖는 트리

• 특징

- 왼쪽자식과 오른쪽자식을 구분

• 왼쪽 서브트리, 오른쪽 서브트리

9-2-2. 완전이진트리 알아보기

- 마지막 레벨을 제외하고 모든 레벨에 노드가 가득 차 있음

- 마지막 레벨 → 왼쪽부터 오른쪽으로 노드를 채움

→ 반드시 끝까지 안 채워도 됨

- 높이 $K \rightarrow$ 노드의 최대수 = $2^{K+1} - 1$

- n 개의 노드 \rightarrow 높이 : $\log n$

9-2-3. 이진 검색 트리 알아보기

- 조건

- 왼쪽 서브트리 노드의 키값은 자신의 노드 키값보다 작아야 합니다.
- 오른쪽 서브트리 노드의 키값은 자신의 노드 키값보다 커야 합니다.
- 키값이 같은 노드는 복수로 존재하지 않습니다.

- 특징

- 구조가 단순
- 중위 순회의 깊이 우선 검색을 통하여 노드값을 오름차순으로 얻을 수 있다.
- 이진검색과 비슷한 방식으로 아주 빠르게 검색할 수 있다.
- 노드를 삽입하기 쉬움