**#시간, 공간복잡도**

시간복잡도 : 입력의 크기와 문제를 해결하는데 걸리는 시간의 상관관계

텍스트, 스크린샷, 라인, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

컴퓨터는 1초동안 3억~5억개의 연산을 수행할 수 있다.

문제의 제한시간이 1초 => 당신은 3억~5억개의 연산 안에 결과를 내야 한다.

1~100의 범위 안에서 어떤 수를 찾을 때, mid의 값과 비교하여 절반씩 줄여가며 찾아내는 연산 -> O(log2N)의 시간복잡도

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

공간복잡도 : 입력의 크기와 문제를 해결하는데 필요한 공간의 상관관계

메모리 제한 512MB = 1.2억개의 int 변수

메모리 제한이 512MB인데 크기가 5억인 int 배열이 필요하다 = 불가능

**#컴퓨터의 소수 표현 (IEEE 754)**

텍스트, 폰트, 스크린샷, 디자인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

1. 실수의 저장/연산 과정에서 반드시 오차가 발생할 수 밖에 없다.

- float은 6자리, double은 15자리의 유효숫자 오차가 발생한다.

- float이 메모리를 적게 사용하긴 하지만 오차가 double보다 크기 때문에 소수 자료형이 필요한 경우 double을 사용하는 것이 좋다.

- 소수 자료형을 사용하는 문제의 경우엔 보통 10-6까지의 오차를 허용한다. 같은 문구가 존재한다. 없다면 대부분 정수만으로 풀이가 가능하다.

1. Double 자료형에 long long 범위의 정수를 함부로 담으면 안된다.

* double은 유효숫자가 15자리이고 long long은 19자리여서 오류

1. ==를 사용하여 비교하면 안된다. 오차가 항상 존재하기 때문에 double 자료형인 두 값이 동일한지 비교하고 싶다면 1e-12만큼의 오차를 따로 계산해줘야한다.

* Abs(a-b) < 1e-12

**#함수 인자**

함수 인자로 변수, 배열, 구조체를 넘기는 경우

변수와 구조체는 값이 복사되어 넘어가서 함수 안에서 값을 변경해도 변수, 구조체의 값이 변하지는 않는다. 값을 변경하고 싶은 경우 포인터나 참조자를 사용하면 된다.

배열을 넘기는 경우 주소가 넘어가기 때문에 함수 안에서 값을 변경하면 배열의 값이 변경된다.

STL (벡터, 큐, 스택 등의 자료구조) 도 함수 인자로 넘길 때 값을 복사하여 넘겨준다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

함수 안에 비교연산 한번밖에 없지만 함수 인자로 넘겨줄 벡터를 복사하는 과정이 필요하기 때문에 시간복잡도가 O(N)이 된다.

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

STL을 참조자를 사용하여 함수 인자로 보내면 주소 정보만 전달하기 때문에 의도한대로 O(1)에 실행을 끝낼 수 있다.

**#시간복잡도를 위한 출력 관련**

ios::sync\_with\_stdio(false) -> cin과 printf의 동기화를 끊기.

* printf와 cin을 섞어서 사용하면 안된다.

cin.tie(nullptr) -> 입력을 받기 전에 cout 버퍼를 비우는 행위를 무시하는 명령어

* 입력과 출력의 순서가 섞일 수 있지만 채점 시스템은 출력값만 확인하기 때문에 상관 없다.

출력의 맨 마지막에 공백 or 줄바꿈이 추가로 있어도 상관 없다.

find(vec.begin(),vec.end(), a) -> 반복자 반환

idx < vec.size() – idx -> 앞에서 가까운지 뒤에서 가까운지 판별

string str; getline(cin, str) -> \n 나오기 전까지 한 줄을 모두 str에 저장. 전에 있는 \n을 cin.ignore()로 무시해야 할 수 있음.

**#1 배열 : 메모리 상에 원소를 연속하게 배치한 자료구조**

O(1)에 k번째 원소 확인 가능 -> 메모리에 연속적으로 저장되기 때문

메모리의 오버헤드가 적음, 캐시 힛 비율이 높음.

메모리 상에 연속한 구간을 잡아야 하기 때문에 할당에 제약이 있음.

임의의 위치 값 확인, 마지막 원소 추가/제거는 O(1)

임의의 위치 추가/제거는 O(N) => 뒤에 있는 원소들을 앞/뒤로 한칸씩 밀어줘야 하기 때문

algorithm 헤더의 std :: copy

std::copy(A + 3, A + 6, B); // A[3]~A[5]를 B로 복사

std::copy(A + 3, A + 6, B + 2); // A[3]~A[5]를 B[2]~B[4]로 복사

std::fill(arr1, arr1 + n, 1); // n-1 인덱스까지 n개의 요소를 1로 채움

memset은 0이 아닌 값으로 초기화하는 경우 오류 발생.

전역에 선언하면 모든 값이 0으로 초기화되지만

지역에 선언할 경우 fill로 초기화 or arr[10] = {}; 로 초기화 해줘야 한다.

algorithm헤더의 fill(a, a+(a의 크기), 0); 을 하면 a배열이 0으로 채워진다.

int k = \*max\_element(begin(arr), end(arr)); (포인터 반환)

**#2 벡터**

배열과 마찬가지로 값을 O(1)에 가지고 올 수 있음.

vec.push\_back() , vec.pop\_back()= O(1)

vec.insert, vec.erase, vec.push\_front, vec.pop\_front = O(N)

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

range based for loop를 사용할 수 있음. for(int e : v1). 참조자 int& e : v1을 하면 e를 변경했을 때 v1이 같이 변경됨.

벡터의 벡터.size()는 unsigned int를 반환하기 때문에 벡터가 비어있는 경우에 쓰면 unsigned int 0 을 반환하고 거기서 어떤 값을 빼서 0 밑으로 값이 작아지는 순간 오버플로우가 발생한다.

**#3 연결 리스트 (텍스트 에디터에 많이 사용)**

임의의 위치에 원소를 확인/변경하기 위해 O(N)가 필요하다.

임의의 위치에 원소를 추가/제거를 위해 O(1)이 필요하다. (다른 원소들의 이동이 필요 없음)

* 전 원소의 주소를 알고 있어야 함

캐시 힛 비율이 낮지만 메모리 할당이 편하다.

단일 연결 리스트 : 원소가 다음 원소의 위치를 기억

이중 연결 리스트 : 원소가 이전 원소의 위치까지 기억 (STL)

원형 연결 리스트 : 맨 끝 원소가 맨 처음 원소의 위치를 기억

배열과 연결 리스트는 원소들의 순서가 정해져 있는 선형 자료구조이다.

push\_front(), push\_back(), pop\_front(), pop\_back()은 모두 O(1)

list<int>::iterator t = L.begin()을 통해 list의 시작 주소를 가리키는 이터레이터 생성.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

L.insert(t,6) 을 통해 t가 가리키는 곳 앞에 6을 삽입

L.erase(t)는 t가 가리키는 값을 제거. 다음 원소의 위치를 반환

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

**#4 스택**

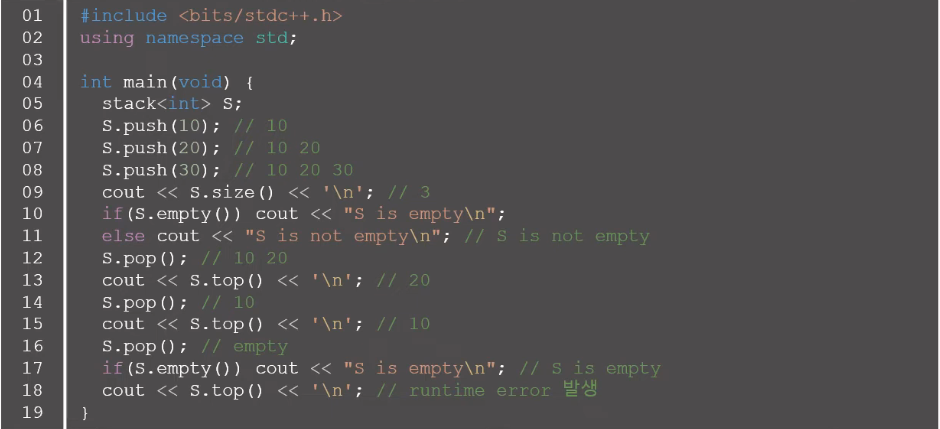
한쪽 끝에서만 추가, 제거, 확인이 가능한 자료구조. FILO이다.

원소의 추가, 제거가 O(1)이며 원소의 확인은 TOP의 원소만 확인 가능하다. 이것도 O(1)

제일 상단이 아닌 원소들의 확인/변경은 원칙적으로 불가능하다.

배열로 스택을 구현할 때 pop에선 그냥 idx를 1 줄이기만 하면 된다. 덮어씌워질테니까.

스택이 비어있을 때 top과 pop을 하면 런타임 에러가 발생한다.



**#5 큐**

뒤에서 추가, 앞에서 제거하는 자료구조형. FIFO이다.

원소의 추가, 제거, 맨 앞/뒤 원소 확인이 O(1)이며, 맨 앞, 맨 뒤가 아닌 나머지 원소들의 확인/변경이 원칙적으로 불가능하다.

배열로 큐를 구현할 땐 배열을 원형으로 만드는 것이 좋다.

큐가 비어있을 때 front, back, pop 등을 호출하면 에러가 발생한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**#6 덱**

DoubleEndedQueue로 자료구조의 맨 앞과 맨 뒤에서 모두 추가와 삭제가 가능하다. 스택과 큐는 특수한 덱의 일종이라고 생각해도 된다.

원소의 추가, 제거, (앞,뒤) 원소 확인이 O(1)이다. 또한 스택, 큐와 다르게 STL에선 인덱스로 원소 확인이 가능하다.

벡터의 상위호환인가? 싶은 생각이 들 수도 있겠지만 벡터는 메모리에 연속되어 저장되어 있고 덱은 연속되어 저장되지 않는다. -> 벡터는 인덱스로 원소 찾을 때 O(1)이다.

앞쪽과 뒤쪽의 추가, 제거가 모두 필요한 경우 덱. 뒤쪽만 필요하면 벡터.

텍스트, 스크린샷, 폰트, 디스플레이이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI 생성 콘텐츠는 정확하지 않을 수 있습니다.

**#스택의 활용(수식의 괄호 쌍)**

어떤 문자열에 들어있는 괄호들의 쌍이 올바르게 매칭되는지 확인하는 문제.

내 생각 : { , [ , ( 이 들어오면 push하고, ), ], } 이 들어오면 top에 있는 괄호와 같은 괄호인지 확인하는 식으로 진행하면 된다.

**#아스키 코드**

소문자 a의 아스키 코드는 97. a – 97을 하면 0이 출력된다. z의 아스키 코드는 122

대문자 A의 아스키 코드는 65. A- 65를 하면 0이 출력된다. Z의 아스키 코드는 90

-97과 -65를 하지 않고 그냥 ‘a’, ‘A’를 빼줘도 동일하게 작동한다.

문자형 숫자를 int 형으로 바꾸고 싶을 땐 ‘0’을 빼주면 된다.

문자열을 숫자로 바꾸고 싶을 땐 atoi(str)을 사용하면 된다.