

#1. PS 맛보기

나정휘

<https://justicehui.github.io/>

목차

빠르게 계산하는 방법

잘 생각하는 방법

알고리즘의 성능

시간 복잡도

점근 표기법

빠르게 계산하는 방법

빠르게 계산하는 방법

어떻게 하면 빠르게 계산할 수 있을까?

- 고려해야 하는 경우의 수 자체를 줄이는 것
- 편하게 계산할 수 있는 형태로 변형하는 것
- 똑같은 값을 더 효율적으로 계산하는 것
- ...

빠르게 계산하는 방법

BOJ 2003. 수들의 합 2

- 양의 정수로 구성된 배열 A가 주어짐
- $A[i] + A[i+1] + \dots + A[j-1] + A[j] = M$ 이 되는 경우의 수를 구하는 문제
- 배열 크기 꿀팁
 - N개의 수가 주어진다 → N의 최댓값보다 큰 배열 생성
 - 10000, 100000 처럼 하면 배열 크기 실수하기 쉬움
 - 10101, 101010 처럼 하면 끝자리 보고 확인할 수 있음
 - 10'000, 100'000 처럼 중간에 작은 따옴표 찍어도 됨

```
#include <stdio.h>

int N, M, A[10101], R;

int f(int i, int j){
    int res = 0;
    for(int k=i; k<=j; k++) res += A[k];
    return res;
}

int main(){
    scanf("%d %d", &N, &M);
    for(int i=1; i<=N; i++) scanf("%d", &A[i]);
    for(int i=1; i<=N; i++){
        for(int j=i; j<=N; j++){
            if(f(i, j) == M) R += 1;
        }
    }
    printf("%d", R);
}
```

빠르게 계산하는 방법

BOJ 2003. 수들의 합 2

- 양의 정수로 구성된 배열 A가 주어짐
- $A[i] + A[i+1] + \dots + A[j-1] + A[j] = M$ 이 되는 경우의 수를 구하는 문제

- 얼마나 많은 연산이 필요할까?

$$\begin{aligned} - T(N) &= \sum_{i=1}^N \sum_{j=i}^N \sum_{k=i}^j 1 = \sum_{i=1}^N \sum_{j=i}^N (j - i + 1) \\ - &= \sum_{i=1}^N \sum_{j=i}^N j + \sum_{i=1}^N \sum_{j=i}^N (1 - i) \\ - &= \sum_{i=1}^N \frac{N(N+1) - i(i-1)}{2} + \sum_{i=1}^N (N - i + 1)(1 - i) \\ - &= \frac{1}{2} \sum_{i=1}^N (N^2 + N - i^2 + i) + \sum_{i=1}^N (N - Ni - i + i^2 + 1 - i) \\ - &= \frac{N^3}{2} + \frac{N^2}{2} - \frac{N(N+1)(2N+1)}{12} + \frac{N(N+1)}{4} \\ &\quad + N^2 - \frac{N^2(N+1)}{2} - \frac{N(N+1)}{2} + \frac{N(N+1)(2N+1)}{6} + N - \frac{N(N+1)}{2} \\ - &\approx \frac{N^3}{2} - \frac{N^3}{6} - \frac{N^3}{2} + \frac{N^3}{3} = \frac{N^3}{6} \end{aligned}$$

```
#include <stdio.h>

int N, M, A[1010], R;

int f(int i, int j){
    int res = 0;
    for(int k=i; k<=j; k++) res += A[k];
    return res;
}

int main(){
    scanf("%d %d", &N, &M);
    for(int i=1; i<=N; i++) scanf("%d", &A[i]);
    for(int i=1; i<=N; i++){
        for(int j=i; j<=N; j++){
            if(f(i, j) == M) R += 1;
        }
    }
    printf("%d", R);
}
```

빠르게 계산하는 방법

BOJ 2003. 수들의 합 2

- 양의 정수로 구성된 배열 A가 주어짐
- $A[i] + A[i+1] + \dots + A[j-1] + A[j] = M$ 이 되는 경우의 수를 구하는 문제

- 조금 더 빠르게!

$$\begin{aligned} - T(N) &= \sum_{i=1}^N \sum_{j=i}^N 1 = \sum_{i=1}^N (N - i + 1) \\ - &= N^2 - \frac{N(N+1)}{2} + N \approx \frac{N^2}{2} < \frac{N^3}{6} \end{aligned}$$

```
#include <stdio.h>

int N, M, A[1010], R;

int main(){
    scanf("%d %d", &N, &M);
    for(int i=1; i<=N; i++) scanf("%d", &A[i]);
    for(int i=1; i<=N; i++){
        int sum = 0;
        for(int j=i; j<=N; j++){
            sum += A[j];
            if(sum == M) R += 1;
        }
    }
    printf("%d", R);
}
```

빠르게 계산하는 방법

BOJ 2003. 수들의 합 2

- 양의 정수로 구성된 배열 A가 주어짐
- $A[i] + A[i+1] + \dots + A[j-1] + A[j] = M$ 이 되는 경우의 수를 구하는 문제
- 더 빠르게!
 - 각각의 i마다 합이 M 이상이 되는 가장 작은 j인 J(i)를 생각해 보자.
 - A의 원소는 모두 양의 정수이기 때문에 $J(i) \leq J(i+1)$
 - A에 0이 없으므로 $i \sim J(i)$ 의 합이 M이지만 확인해도 충분함
 - 따라서 실제로 고려해야 하는 구간의 개수는 N개

빠르게 계산하는 방법

BOJ 2003. 수들의 합 2

- 양의 정수로 구성된 배열 A가 주어짐
- $A[i] + A[i+1] + \dots + A[j-1] + A[j] = M$ 이 되는 경우의 수를 구하는 문제
- 더 빠르게!
 - 각각의 i마다 합이 M 이상이 되는 가장 작은 j인 J(i)를 생각해 보자.
 - A의 원소는 모두 양의 정수이기 때문에 $J(i) \leq J(i+1)$
 - A에 0이 없으므로 i ~ J(i)의 합이 M이지만 확인해도 충분함
 - 따라서 실제로 고려해야 하는 구간의 개수는 N개
 - i와 j는 감소하지 않고 증가만 하므로
 - 반복문은 최대 2N번 돌아감
 - 투 포인터 기법이라고 부름

```
#include <stdio.h>

int N, M, A[10101], R;

int main(){
    scanf("%d %d", &N, &M);
    for(int i=1; i<=N; i++) scanf("%d", &A[i]);
    for(int i=1, j=1, sum=0; i<=N; sum+=A[i]){
        while(j <= N && sum < M) sum += A[j++];
        if(sum == M) R += 1;
    }
    printf("%d", R);
}
```

빠르게 계산하는 방법

약수의 개수를 구하는 문제

- $f(k)$ = k 의 약수의 개수라고 정의하자.
- $f(1) + f(2) + \dots + f(n-1) + f(n)$ 을 계산하는 문제

- 가장 단순한 방법

$$- T(N) = \sum_{k=1}^N k = \frac{N(N+1)}{2}$$



```
#include <stdio.h>
```

```
int f(int k){
    int res = 0;
    for(int i=1; i<=k; i++){
        if(k % i == 0) res++;
    }
    return res;
}

int main(){
    int n, sum = 0;
    scanf("%d", &n);
    for(int i=1; i<=n; i++) sum += f(i);
    printf("%d", sum);
}
```

빠르게 계산하는 방법

약수의 개수를 구하는 문제

- $f(k)$ = k 의 약수의 개수라고 정의하자.
- $f(1) + f(2) + \dots + f(n-1) + f(n)$ 을 계산하는 문제
- 조금 더 빠른 방법
 - $T(N) = \sum_{k=1}^N \lfloor \sqrt{k} \rfloor \leq \sum_{k=1}^N \sqrt{k} = N\sqrt{N} < \frac{N(N+1)}{2}$
 - $N = pq$ ($p, q > 0$)이면 p, q 중 하나는 $\sqrt{N}$ 이하
 - 둘 모두 $\sqrt{N}$ 초과이면 $pq > N$ 이 되기 때문

```
#include <stdio.h>

int f(int k){
    int res = 0;
    for(int i=1; i*i<=k; i++){
        if(k % i == 0) res++;
        if(k % i == 0 && i * i != k) res++;
    }
    return res;
}

int main(){
    int n, sum = 0;
    scanf("%d", &n);
    for(int i=1; i<=n; i++) sum += f(i);
    printf("%d", sum);
}
```

빠르게 계산하는 방법

약수의 개수를 구하는 문제

- $f(k) = k$ 의 약수의 개수라고 정의하자.
- $f(1) + f(2) + \dots + f(n-1) + f(n)$ 을 계산하는 문제
- 더 빠른 방법
 - 1부터 n 까지의 약수의 개수를 세는 문제
 - 1부터 n 까지의 n 이하의 배수의 개수를 세는 문제
 - 둘이 똑같은 문제

빠르게 계산하는 방법

약수의 개수를 구하는 문제

- $f(k)$ = k 의 약수의 개수라고 정의하자.
- $f(1) + f(2) + \dots + f(n-1) + f(n)$ 을 계산하는 문제
- 더 빠른 방법
 - 1부터 n 까지의 약수의 개수를 세는 문제
 - 1부터 n 까지의 n 이하의 배수의 개수를 세는 문제
 - 둘이 똑같은 문제
 - $T(N) = \sum_{i=1}^N \frac{N}{i} = N \sum_{i=1}^N \frac{1}{i}$
 - $\approx N \int_1^N \frac{1}{x} dx = N[\ln x]_1^N = N \ln N < N\sqrt{N}$



```
#include <stdio.h>

int main(){
    int n, sum = 0;
    scanf("%d", &n);
    for(int i=1; i<=n; i++){
        for(int j=i; j<=n; j+=i) sum++;
    }
    printf("%d", sum);
}
```

빠르게 계산하는 방법

약수의 개수를 구하는 문제

- $f(k)$ = k 의 약수의 개수라고 정의하자.
- $f(1) + f(2) + \dots + f(n-1) + f(n)$ 을 계산하는 문제
- 더 빠른 방법
 - 1부터 n 까지의 약수의 개수를 세는 문제
 - 1부터 n 까지의 n 이하의 배수의 개수를 세는 문제
 - 둘이 똑같은 문제
 - 이렇게 하면 나눗셈 N 번으로 됨
 - 사실 $\sqrt{N}$ 번으로도 가능한데... 지금은 몰라도 됨



```
#include <stdio.h>
```

```
int main(){  
    int n, sum = 0;  
    scanf("%d", &n);  
    for(int i=1; i<=n; i++) sum += n / i;  
    printf("%d", sum);  
}
```

빠르게 계산하는 방법

BOJ 11659. 구간 합 구하기 4

- 양의 정수로 구성된 배열 A가 주어짐
- i, j 가 주어지면 $A[i] + A[i+1] + \dots + A[j-1] + A[j]$ 를 구하는 쿼리를 M번 처리해야 함
- 가장 단순한 방법
 - 매번 $a = 1, b = N$ 으로 주어지면 NM 만큼의 연산 필요
 - 더 빠르게 할 수 있을까?



```
#include <stdio.h>

int N, M, A[101010];

int main(){
    scanf("%d %d", &N, &M);
    for(int i=1; i<=N; i++) scanf("%d", &A[i]);
    for(int i=1; i<=M; i++){
        int a, b, sum = 0;
        scanf("%d %d", &a, &b);
        for(int j=a; j<=b; j++) sum += A[j];
        printf("%d\n", sum);
    }
}
```

빠르게 계산하는 방법

BOJ 11659. 구간 합 구하기 4

- 양의 정수로 구성된 배열 A가 주어짐
- i, j 가 주어지면 $A[i] + A[i+1] + \dots + A[j-1] + A[j]$ 를 구하는 쿼리를 M번 처리해야 함
- 더 빠른 방법
 - 고등학교에서 수열의 합을 구하던 것을 생각해 보자.
 - $S_n = \sum_{k=1}^n A_k = S_{n-1} + A_n$ 이라고 정의하면
 - $\sum_{k=a}^b A_k = S_b - S_{a-1}$
 - S를 미리 계산해 두면 구간의 합은 뺄셈 1번으로 가능

빠르게 계산하는 방법

BOJ 11659. 구간 합 구하기 4

- 양의 정수로 구성된 배열 A가 주어짐
- i, j 가 주어지면 $A[i] + A[i+1] + \dots + A[j-1] + A[j]$ 를 구하는 쿼리를 M번 처리해야 함
- 더 빠른 방법
 - 고등학교에서 수열의 합을 구하던 것을 생각해 보자.
 - $S_n = \sum_{k=1}^n A_k = S_{n-1} + A_n$ 이라고 정의하면
 - $\sum_{k=a}^b A_k = S_b - S_{a-1}$
 - S를 미리 계산해 두면 구간의 합은 뺄셈 1번으로 가능
 - S를 보통 누적 합 배열이라고 부름
 - 인덱스 0부터 사용하면 $S[a-1]$ 에서 오류 발생할 수 있음



```
#include <stdio.h>

int N, M, A[101010], S[101010];

int main(){
    scanf("%d %d", &N, &M);
    for(int i=1; i<=N; i++) scanf("%d", &A[i]);
    for(int i=1; i<=N; i++) S[i] = S[i-1] + A[i];
    for(int i=1; i<=M; i++){
        int a, b;
        scanf("%d %d", &a, &b);
        printf("%d\n", S[b] - S[a-1]);
    }
}
```

잘 생각하는 방법

잘 생각하는 방법

잘 생각하는 것이란?

- 문제를 잘 만져서 내가 아는 개념을 적용할 수 있는 형태로 변형하는 것
- 똑같은 것을 똑같다고 생각하는 것
- 문제를 수식으로 잘 표현한 다음 성질을 관찰하는 것
- ...

잘 생각하는 방법

BOJ 8298. Coins

- R과 O로 구성되어 있는 문자열이 주어짐
- O의 개수가 정확히 R의 개수의 K배인 구간의 최대 길이를 구하는 문제
- ex. RORR**OO**R**OO**R**OO**ORO, K=3

잘 생각하는 방법

BOJ 8298. Coins

- R과 O로 구성되어 있는 문자열이 주어짐
- O의 개수가 정확히 R의 개수의 K배인 구간의 최대 길이를 구하는 문제
- ex. RORR**OO**R**OO**R**OO**ORO, K=3
- K = 1이면 R의 개수와 O의 개수가 같은 구간을 찾는 문제
 - R을 +1, O를 -1이라고 생각하면 합이 0인 구간을 찾는 문제
 - $S[j] = S[i-1]$ 인 구간을 찾는 문제

잘 생각하는 방법

BOJ 8298. Coins

- R과 O로 구성되어 있는 문자열이 주어짐
- O의 개수가 정확히 R의 개수의 K배인 구간의 최대 길이를 구하는 문제
- ex. RORR**OOROOORO**O, K=3
- K = 1이면 R의 개수와 O의 개수가 같은 구간을 찾는 문제
 - R을 +1, O를 -1이라고 생각하면 합이 0인 구간을 찾는 문제
 - $S[j] = S[i-1]$ 인 구간을 찾는 문제
- K > 1이면?
 - R을 +K, O를 -1이라고 생각하면 똑같이 풀 수 있음

잘 생각하는 방법

BOJ 4307. 개미

- 길이가 L 인 막대 위에 N 마리의 개미가 있음
- 각 개미는 왼쪽 또는 오른쪽으로 1 단위 속도로 이동하고 있음
- 개미가 막대의 끝에 도달하면 떨어짐
- 두 개미가 만나면 방향을 반대로 바꾸어 이동함
- 개미의 초기 위치와 방향이 주어지면 가장 먼저/마지막으로 개미가 떨어지는 시간 구하는 문제

잘 생각하는 방법

BOJ 4307. 개미

- 길이가 L 인 막대 위에 N 마리의 개미가 있음
- 각 개미는 왼쪽 또는 오른쪽으로 1 단위 속도로 이동하고 있음
- 개미가 막대의 끝에 도달하면 떨어짐
- 두 개미가 만나면 방향을 반대로 바꾸어 이동함
- 개미의 초기 위치와 방향이 주어지면 가장 먼저/마지막으로 개미가 떨어지는 시간 구하는 문제
- 두 개미가 튕겨져 나가는 것 = 서로를 스쳐 지나가는 것
 - 각 개미가 초기 상태에서 끝까지 이동하는데 걸리는 최소/최대 시간을 구하면 됨

잘 생각하는 방법

BOJ 9012. 괄호

- 괄호 문자열이 주어지면 올바른 괄호 문자열인지 판별하는 문제
- 예시
 - `((()))` 는 올바른 괄호 문자열
 - `()()()` 는 올바르지 않은 괄호 문자열
 - `()))()` 는 올바르지 않은 괄호 문자열

잘 생각하는 방법

BOJ 9012. 괄호

- 괄호 문자열이 주어지면 올바른 괄호 문자열인지 판별하는 문제
- 예시
 - `((()))` 는 올바른 괄호 문자열
 - `()()()` 는 올바르지 않은 괄호 문자열
 - `()))()` 는 올바르지 않은 괄호 문자열
- 올바른 괄호 문자열의 조건
 - 여는 괄호의 개수 = 닫는 괄호의 개수
 - 앞에서부터 봤을 때 항상 (여는 괄호의 개수 $\geq$ 닫는 괄호의 개수)를 만족해야 함

잘 생각하는 방법

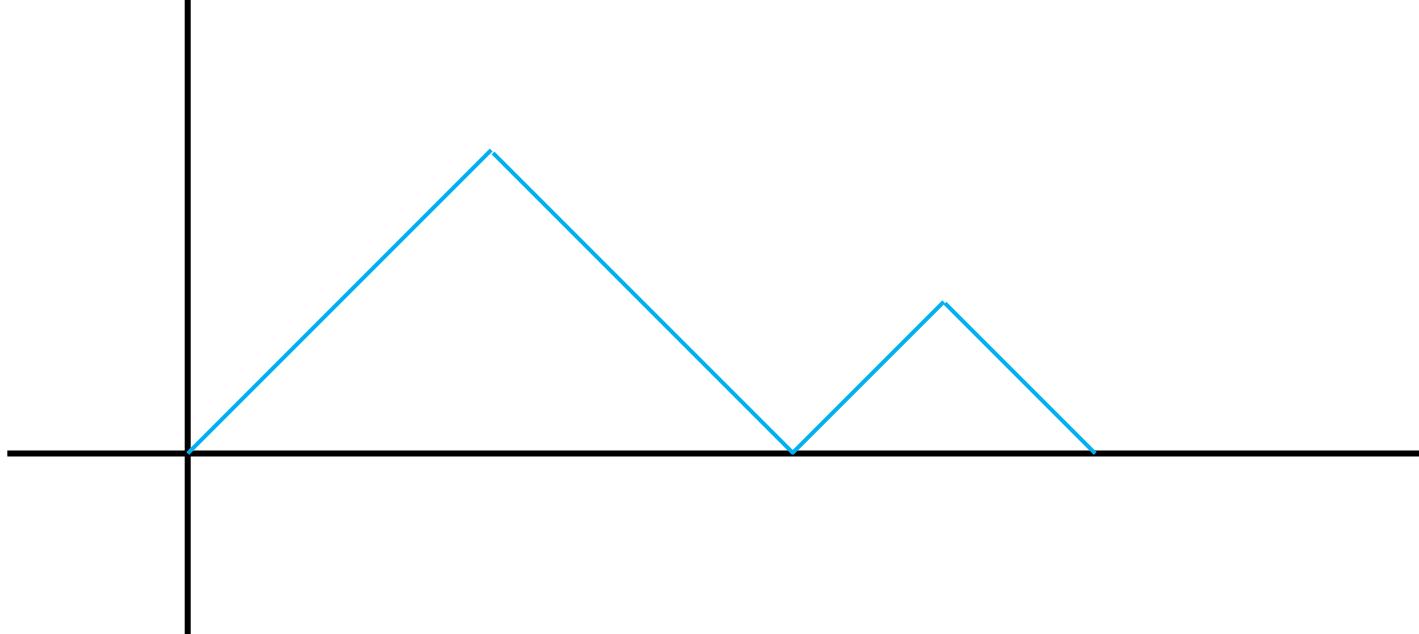
BOJ 9012. 괄호

- 괄호 문자열이 주어지면 올바른 괄호 문자열인지 판별하는 문제
- 예시
 - `((()))`는 올바른 괄호 문자열
 - `()()()`는 올바르지 않은 괄호 문자열
 - `()))()`는 올바르지 않은 괄호 문자열
- 올바른 괄호 문자열의 조건
 - 여는 괄호의 개수 = 닫는 괄호의 개수
 - 앞에서부터 봤을 때 항상 (여는 괄호의 개수 $\geq$ 닫는 괄호의 개수)를 만족해야 함
 - 여는 괄호를 +1, 닫는 괄호를 -1로 취급하는 수열 A와 누적 합 배열 S를 만들면
 - $S[N] = 0$ and $\forall i, S[i] \geq 0$ 을 만족하는지 확인하면 됨

잘 생각하는 방법

BOJ 9012. 괄호

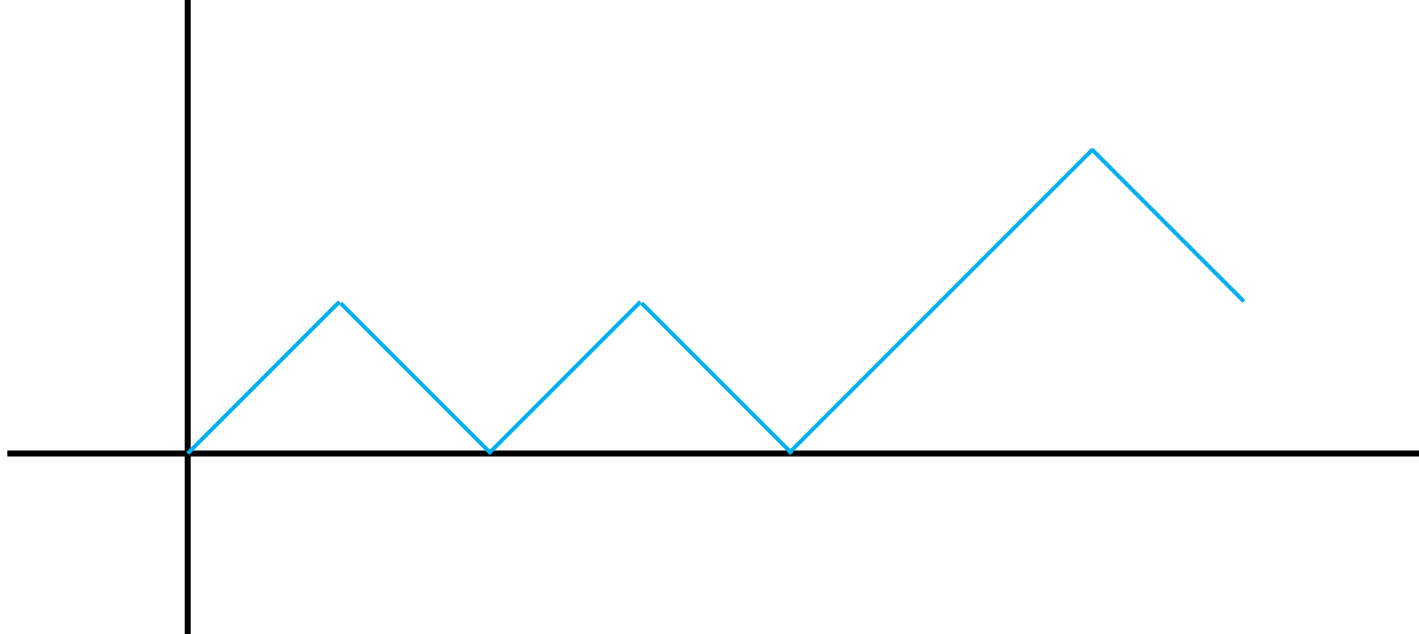
- 괄호 문자열이 주어지면 올바른 괄호 문자열인지 판별하는 문제
- 예시
 - `((()))` 는 올바른 괄호 문자열
 - `()()()` 는 올바르지 않은 괄호 문자열
 - `()))()` 는 올바르지 않은 괄호 문자열



잘 생각하는 방법

BOJ 9012. 괄호

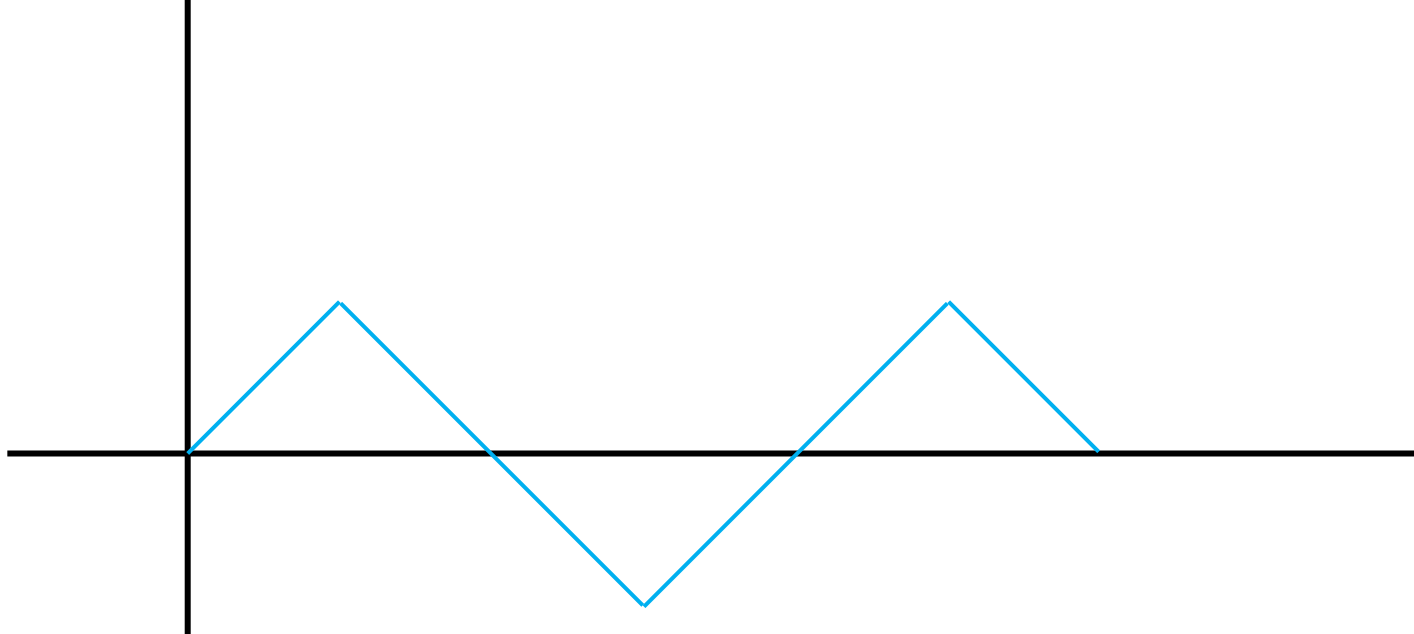
- 괄호 문자열이 주어지면 올바른 괄호 문자열인지 판별하는 문제
- 예시
 - `((()))` 는 올바른 괄호 문자열
 - `()()()` 는 올바르지 않은 괄호 문자열
 - `))((` 는 올바르지 않은 괄호 문자열



잘 생각하는 방법

BOJ 9012. 괄호

- 괄호 문자열이 주어지면 올바른 괄호 문자열인지 판별하는 문제
- 예시
 - `((()))` 는 올바른 괄호 문자열
 - `()()()` 는 올바르지 않은 괄호 문자열
 - `()))()` 는 올바르지 않은 괄호 문자열



잘 생각하는 방법

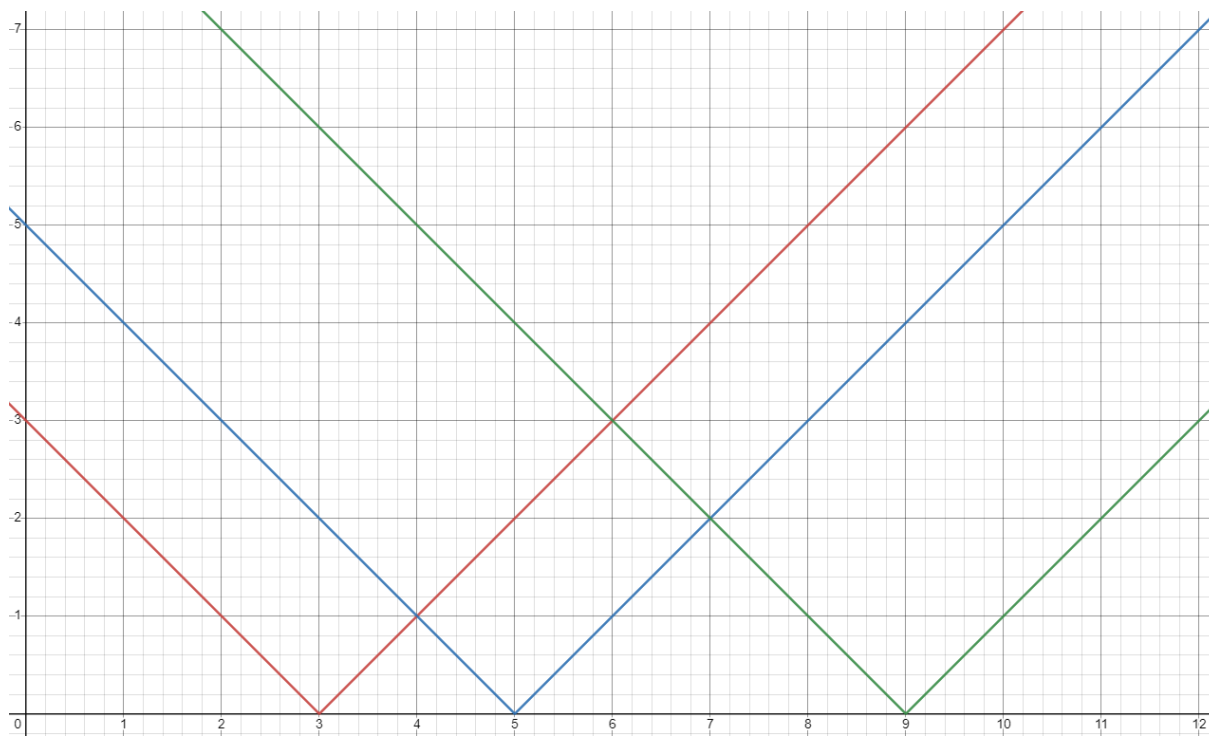
택시 거리 최소화

- 두 점 $(x_1, y_1), (x_2, y_2)$ 사이의 거리를 $|x_1 - x_2| + |y_1 - y_2|$ 라고 정의하자.
- N 개의 점이 주어졌을 때 $\sum(|x_i - x| + |y_i - y|)$ 를 최소화하는 점 (x, y) 를 찾는 문제
- 필요한 관찰
 - x 좌표와 y 좌표는 분리해서 생각해도 됨
 - $\sum |x_i - x|$ 를 최소화하는 x 와 $\sum |y_i - y|$ 를 최소화하는 y 를 각각 구하는 문제

잘 생각하는 방법

택시 거리 최소화

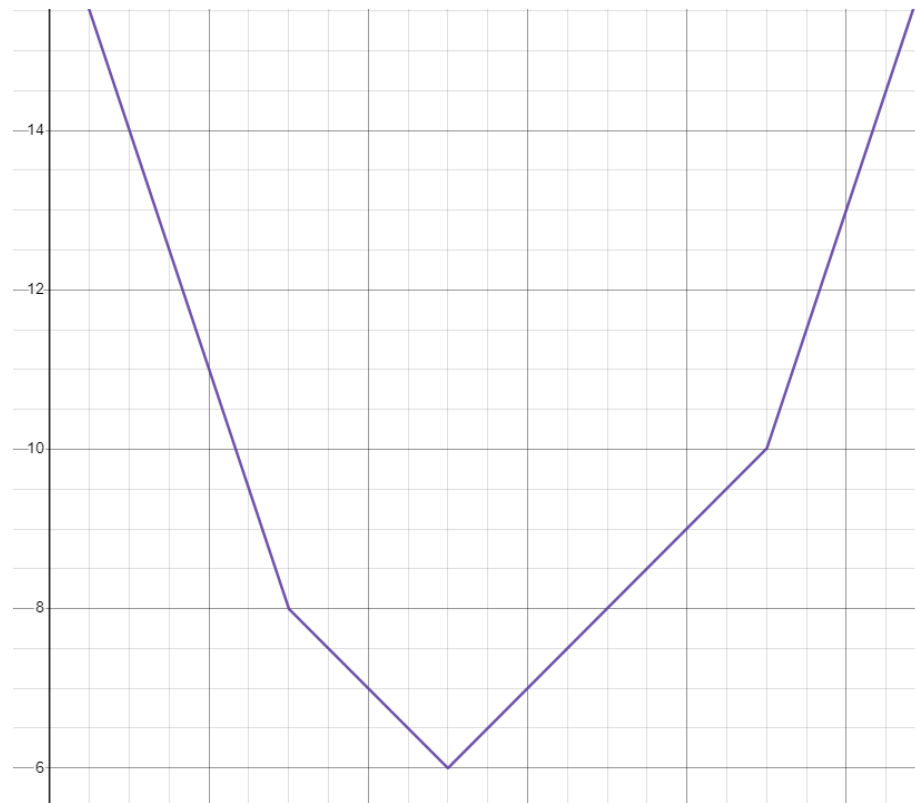
- 필요한 관찰
 - x좌표와 y좌표는 분리해서 생각해도 됨
 - $\sum |x_i - x|$ 를 최소화하는 x 와 $\sum |y_i - y|$ 를 최소화하는 y 를 각각 구하는 문제
 - $f_i(x) = |x_i - x|$ 라고 정의하면 f_i 는 x_i 를 중심으로 V자 형태로 뻗어나가는 형태
 - $x_1 = 3, x_2 = 5, x_3 = 9$ 일 때의 그래프



잘 생각하는 방법

택시 거리 최소화

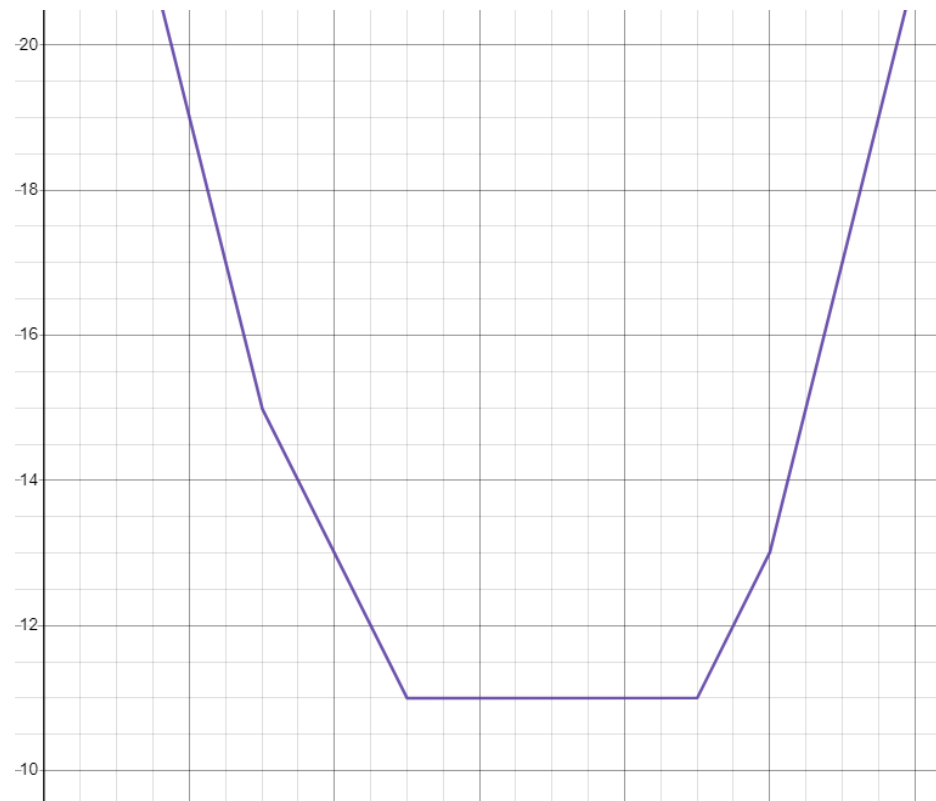
- 필요한 관찰
 - $f_i(x) = |x_i - x|$ 라고 정의하면 f_i 는 x_i 를 중심으로 V자 형태로 뻗어나가는 형태
 - $\sum f_i(x)$ 가 최소가 되는 x 를 구하는 문제
 - $x_1 = 3, x_2 = 5, x_3 = 9$ 일 때의 그래프



잘 생각하는 방법

택시 거리 최소화

- 필요한 관찰
 - $f_i(x) = |x_i - x|$ 라고 정의하면 f_i 는 x_i 를 중심으로 V자 형태로 뻗어나가는 형태
 - $\sum f_i(x)$ 가 최소가 되는 x 를 구하는 문제
 - $x_1 = 3, x_2 = 5, x_3 = 9, x_4 = 10$ 일 때의 그래프



잘 생각하는 방법

택시 거리 최소화

- 필요한 관찰
 - $f_i(x) = |x_i - x|$ 라고 정의하면 f_i 는 x_i 를 중심으로 V자 형태로 뻗어나가는 형태
 - $\sum f_i(x)$ 가 최소가 되는 x 를 구하는 문제
 - 점이 홀수 개면 기울기가 음수에서 양수로 바뀌는 지점
 - 점이 짝수 개면 기울기가 0인 지점
 - $f_i(x)$ 는 $x < x_i$ 인 지점의 기울기에 -1, $x > x_i$ 인 지점의 기울기에 +1 만큼 기여함
 - 점이 홀수 개면 중앙값에서 기울기가 음수 → 양수로 바뀜
 - 점이 짝수 개면 가운데 있는 두 점 사이에서 기울기 0임
 - 중앙값을 구하면 해결!

알고리즘의 성능

알고리즘의 성능

알고리즘의 성능을 판단하는 척도

- 작업량: 얼마나 적은 연산을 필요로 하는가?
- 메모리 사용량: 얼마나 적은 공간을 사용하는가?
- 정확성: 얼마나 정확한 답을 구할 수 있는가?
- 최적성: 더 이상 개선할 여지가 없을 만큼 최적화가 잘 되었나?
- 단순성: 알고리즘의 작동 과정이 얼마나 단순한가?

알고리즘의 성능

작업량

- 알고리즘 문제를 풀 때는 작업(연산)의 횟수가 실행 시간에 가장 큰 영향을 미침
- 대충 어림 잡아서 1초에 연산 1억 번 정도
 - 메모리 접근 횟수와 방법도 중요하지만 지금은 생략...
- 실제 프로그램을 개발할 때는 연산 횟수 말고 다른 부분에서 시간이 더 오래 걸릴 수 있음
 - 네트워크
 - 광섬유에서의 빛의 속도는 진공에 비해 약 2/3 정도 (약 200'000'000m/s)
 - 서울과 뉴욕은 약 11000km 떨어져 있으므로 아무리 빨라도 55ms 이상의 딜레이가 발생할 수밖에 없음
 - 프로세스 스케줄링
 - 컴퓨터에서는 하나의 프로그램만 실행되는 것이 아님
 - 운영체제는 매 순간 어떤 프로세스를 처리할지 결정
 - 병목 현상
 - 프로그램이 여러 코어에서 병렬로 실행되는 경우
 - ...

알고리즘의 성능

메모리 사용량

- 요즘 나오는 문제는 대부분 메모리 제한을 여유롭게 주기 때문에 별로 신경 쓰지 않아도 됨
- 하지만 가끔씩 메모리 최적화를 요구하는 문제가 나옴
 - IOI '13 Day 1. Wombats (BOJ 8874)
 - IOI '13 Day 2. Game (BOJ 8876)
 - CEOI '15 Day 1. Pipes (BOJ 10957)
 - KOI '18 고등부. 조화로운 행렬 (BOJ 15977)
 - PA 2018 Round 3. Palindrom (BOJ 26640)
 - PtzCamp Winter '23 Day 7. Classical Data Structure Problem (BOJ 28204)
- 메모리 제한이 128MB보다 작으면 의심해 보자.
 - 요즘은 대부분 512MB ~ 1024MB 정도로 줌

알고리즘의 성능

정확성

- 이 세상에는 빠르게 풀 수 없음이 증명된 문제들이 있음
 - $P \neq NP$ 이면 Satisfiability, Chromatic Number, Traveling Salesman Problem 등을 다항 시간에 풀 수 없음
 - 관심이 있다면... [\(링크\)](#)
 - 3-SUM Conjecture이 참이면 Collinear, Separator, Convolution3SUM 등을 $O(N^{2-\epsilon})$ 보다 빠르게 풀 수 없음
 - 관심이 있다면... [\(링크\)](#)
 - APSP Conjecture이 참이면 2D maximum subarray 등을 $O(N^{3-\epsilon})$ 보다 빠르게 풀 수 없음
 - 관심이 있다면... [\(링크\)](#)
- 이런 문제는 정확한 값을 구하는 것이 어렵기 때문에 **적당히 좋은 답**을 빠르게 찾는 시도를 많이 함

알고리즘의 성능

정확성

- Traveling Salesman Problem
 - N개의 도시와 도시 간의 거리가 주어짐
 - N개의 도시를 모두 방문하고 다시 시작점으로 돌아오는 최소 거리를 구하는 문제
 - 최소 가중치 해밀턴 회로를 찾는 문제
 - $P \neq NP$ 이면 다항 시간에 해결할 수 없음
 - 단, $D(i, j) \leq D(i, k) + D(k, j)$ 를 만족하면 다항 시간에 최적해의 1.5배 이하인 해를 구할 수 있음
 - Christofides' algorithm
- Set Cover
 - 집합 X의 부분 집합 N개가 주어졌을 때, 합집합이 X가 되도록 최소 개수의 부분 집합을 선택하는 문제
 - $P \neq NP$ 이면 다항 시간에 해결할 수 없음
 - 하지만 아주 간단한 전략을 이용해 최적해의 $\ln N$ 배 이하인 해를 구할 수 있음
- TSP에서 1.5를 더 줄일 수 있을까?
- Set Cover에서 $\ln N$ 대신에 상수 배의 해를 구할 수 있을까?
- ...

알고리즘의 성능

최적성

- 문자열 S 안에 문자열 P 가 등장하는지 확인하는 문제
 - 일단 S 와 P 를 한 번씩은 훑어야 하기 때문에 최소한 $|S| + |P|$ 만큼의 연산이 필요함
 - 단순히 구현하면 $|S| * |P|$ 만큼의 연산이 필요함 → 최적이지 않음
 - KMP 알고리즘을 사용하면 $|S| + |P|$ 만큼의 연산이 필요함 → 더 좋아질 수 없으므로 이게 최적
- N 개의 원소를 정렬하는 문제
 - 최소한 약 $N \log N$ 번의 연산이 필요함을 증명할 수 있음
 - 간단한 알고리즘을 사용하면 N^2 만큼의 연산이 필요함 → 최적이지 않음
 - 조금 더 복잡한 알고리즘(합병 정렬 등)을 사용하면 $N \log N$ 만큼의 연산이 필요함 → 더 빠르게 만들 수 없음

시간 복잡도

시간 복잡도

시간 복잡도

- 문제를 해결하는데 **걸리는 시간**과 **입력 크기**의 관계를 나타내는 함수
 - 연산이 많아질수록 오래 걸림
 - PS에서는 네트워크 통신, CPU 점유 등은 신경 안 씀
 - 연산을 몇 번 수행하는지 확인하면 수행 시간을 대략적으로 유추할 수 있음
 - **최악의 경우**가 중요함
-
- 문제를 해결하는데 필수적인 **기본 연산**을 정한 다음
 - 기본 연산의 **수행 횟수**를 계산

시간 복잡도

길이가 N인 배열에서 값이 K인 원소가 존재하는지 확인

- 기본 연산: 배열의 한 원소와 K의 값을 비교
- 기본 연산의 수행 횟수: 최대 N회
- 시간 복잡도: $T(N) = N$

시간 복잡도

길이 N인 배열에서 최댓값 찾기

- 기본 연산: 배열에서 두 원소의 값 비교
- 기본 연산의 수행 횟수: 최대 N-1회
- 시간 복잡도 $T(N) = N-1$

시간 복잡도

길이가 각각 N , M 인 정렬된 두 배열을 정렬된 하나의 배열로 합치는 과정

- 기본 연산: 두 배열의 원소를 서로 비교
- 기본 연산의 수행 횟수: 최대 $N+M-1$ 번
- 시간 복잡도 $T(N, M) = N+M-1$

시간 복잡도

정수 N이 소수인지 판별하는 것

- 기본 연산: N이 어떤 정수 k로 나누어 떨어지는지 확인하는 것
- 기본 연산의 수행 횟수: 최대 $\sqrt{N}$ 번
- 시간 복잡도 $T(N) = \sqrt{N}$
 - 입력의 크기에 대해 나타내면 $T(L) = B^{L/2}$, B는 진법

점근 표기법

점근 표기법

함수에서 가장 중요한 정보

- $f(x) = x^2 + 5x, g(x) = 27x$
 - $x = 1: f(x) = 6, g(x) = 27$
 - $x = 10: f(x) = 150, g(x) = 270$
 - $x = 100: f(x) = 10500, g(x) = 2700$
 - $x = 1000: f(x) = 1005000, g(x) = 27000$
- x 가 적당히 큰 수면 $f(x) > g(x)$ 를 만족함
- $x > x_0$ 이면 항상 $f(x) > g(x)$ 를 만족함

점근 표기법

함수에서 가장 중요한 정보

- 다항 함수: 최고차항의 차수가 중요
 - ex) $f(x) = x^2 + 5x, g(x) = 27x$
 - $x > 22$ 이면 항상 $f(x) > g(x)$
- 지수 함수는 밑수의 최댓값이 중요
 - $f(x) = 3^x + x - 10, g(x) = 2^x + x^3 + 5x$
 - $x > 4.6$ 이면 항상 $f(x) > g(x)$
- x 가 충분히 크면 최고차항의 차수 or 밑수의 최댓값이 큰 함수가 더 큼

점근 표기법

Big-O Notation

- $f(x) \in O(g(x))$
 - 어떤 실수 x_0 과 양의 실수 c 가 있어서
 - $x > x_0$ 을 만족하는 모든 x 에 대해
 - $f(x) \leq c \times g(x)$ 를 만족한다.
- 어떤 실수 x_0 보다 큰 모든 x
 - x 가 한 없이 커지면 항상 부등호가 성립
- 어떤 양의 실수 c
 - 최고차항의 계수 무시
 - 최고차항의 차수와 지수 항의 밑수가 중요함

점근 표기법

Big-O Notation

- Big-O Notation의 의미
 - $f(x)$ 가 아무리 빨리 증가해도
 - 최대 $g(x)$ 에 비례하는 수준으로 증가한다
 - $f(x)$ 의 상계를 나타냄
- $f(x) = 5x, g(x) = x^2$
 - $x_0 = 5, c = 1$ 이면 $5x \leq 1 \times x^2 : 5x \in O(x^2)$
- $f(x) = 2^x + 10x + 5, g(x) = 2^x$
 - $x_0 = 7, c = 2$ 이면 $2^x + 10x + 5 \leq 2 \times 2^x : 2^x + 10x + 5 \in O(2^x)$

점근 표기법

Big-Omega Notation

- Big-O Notation과 반대로 함수의 하계를 나타냄
- $f(x) \in \Omega(g(x))$
 - 어떤 실수 x_0 과 양의 실수 c 가 있어서
 - $x > x_0$ 을 만족하는 모든 x 에 대해
 - $f(x) \geq c \times g(x)$ 를 만족한다.

Big-Theta Notation

- 상계와 하계의 교집합(Tight Bound)
- $f(x) \in \Theta(g(x)) \Leftrightarrow f(x) \in O(g(x)) \wedge f(x) \in \Omega(g(x))$
 - $f(x)$ is the order of $g(x)$, $f(x)$ and $g(x)$ are the same order
 - 시간 복잡도가 order of n 이다 = $\Theta(n)$ 이다

점근 표기법

극한을 이용한 표현

- $f(x) \in O(g(x)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(x)}{g(x)} = c \geq 0$ 로 수렴
- $f(x) \in \Omega(g(x)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(x)}{g(x)} > 0$
- $f(x) \in \Theta(g(x)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(x)}{g(x)} = c > 0$ 로 수렴

점근 표기법

다양한 시간 복잡도

- 복잡도의 위계: $1 \ll \log \log n \ll \log^k n \ll \sqrt{n} \ll n \ll n \log^k n \ll n\sqrt{n} \ll n^2 \ll 2^n \ll 3^n \ll n! \ll n^n$
- 헛갈리는 복잡도
 - 양의 상수 c 에 대해 $c^{\log n} = n^{\log c}$ 이므로 n 에 대한 다항 시간
 - $\log(n!)$
 - $\log(n!) = \log 1 + \log 2 + \dots + \log n \leq \log n + \log n + \dots + \log n = n \log n$
 - $\log(n!) = \log 1 + \log 2 + \dots + \log n \geq \log \frac{n}{2} + \log \left(\frac{n}{2} + 1\right) + \dots + \log n \geq \log \frac{n}{2} + \log \frac{n}{2} + \dots + \log \frac{n}{2} = \frac{n}{2} \log \frac{n}{2}$
 - $\frac{n}{2} \log \frac{n}{2} \leq \log(n!) \leq n \log n$ 이므로 $\log(n!) \in \Theta(n \log n)$
 - 또는 Stirling's Approximation에 의해 $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$ 이므로 $\log(n!) \approx n(\log n - 1) + \frac{1}{2} \log(2\pi n)$ 임을 알 수 있음
 - $(\log n)!$
 - Stirling's Approximation에 의해 $(\log n)! \approx \sqrt{2\pi \log n} \left(\frac{\log n}{e}\right)^{\log n}$
 - $(\log n)^{\log n} = (e^{\log \log n})^{\log n} = e^{\log n \log \log n} = (e^{\log \log n})^{\log n} = n^{\log \log n}$ 이므로 모든 다항 함수보다 빠르게 증가함