다항식 계산 - 두가지 버전 시간 측정

다항식 계산하기

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

- 위의 식은 전형적인 `n-1` 차 다항식이다
- 다항식의 `n` 개의 계수(coefficient)가 리스트 `A` 에 저장되어 있다고 하자
- `evaluate_n2(A, x)`:
  - `f(x)`를 계산하고 그 값을 리턴하는 데, `O(n^2)` 시간의 계산이 필요한 함수
- `evaluate_n(A, x)`:
  - `f(x)`를 계산하고 그 값을 리턴하는 데, `O(n)` 시간의 계산이 필요한 함수

- 시간 측정법
  - `time` 모듈을 이용한다
  - `time.process_time()` 함수는 현재 프로세서 시간(초)을 리턴한다
  - `time.perf_counter()` 함수는 특정 시작 시간으로부터 경과한 현재의 시간(초)을 리턴한다
  - 예를 들어 함수 `f` 의 실행시간을 측정하고 싶다면

```
import time
s = time.process_time() # 또는 time.perf_counter()
f() # f 호출
e = time.process_time() # 또는 time.perf_counter()
print("수행시간 =", e-s)
```

다항식 계산을 위한 두 가지 버전 시간 측정해 보기

1. 입력 크기 `n`을 `input` 받은 후, `n-1` 차 다항식의 `n` 개의 **계수**를 랜덤 생성하여 리스트 `A` 에 저장함.
   1. `random` 모듈을 `import` 한 후, `randint(-1000, 1000)`을 호출하여 랜덤 정수를 `n` 개 생성하면 됨
2. `x` 값은 `randint(-1000, 1000)`을 호출하여 생성함

3. 두 함수 `evaluate_n2(A, x)`과 `evaluate_n(A, x)`을 작성하여 각각 호출함
4. 위의 실행시간 측정 방법을 이용하여 두 함수의 실행시간을 각각 측정하여 출력함
    1. `n` 값을 1000 부터 100,000 까지 다양하게 바꿔가면서 측정해봄. (제한시간은 **60 초**임)
        - 100,000 가까이 되면 `O(n^2)` 시간 알고리즘은 제한시간 안에 끝나지 않을 수 있음. 이 경우엔 가능한 크기까지만 하면 됨
    2. 두 함수의 실행시간 차이가 `n` 의 값에 따라 어떻게 변하는지 살펴보고, **두 페이지**짜리 레포트를 작성해 **PDF 형식**으로 eclass 에 업로드할 것
        - 레포트 **표지는 작성하지 말 것**
        - 두 함수의 수행시간을 분석해 Big-O 로 표기하고,
        - 다양한 `n` 값의 값에 따른 두 함수의 실제 실행 시간을 그래프 등의 시각적인 방법으로 나타내어 설명하고,
        - 느낀점을 추가해 마무리하면 됨

[배열-정렬-스캔] a + b + c = 0?제출완료

4 초

- 정수가 저장된 세 개의 리스트 `A`, `B`, `C` 에 대해, `a ∈ A, b ∈ B, c ∈ C` 를 선택해 `a + b + c = 0` 이 되는 `(a, b, c)` 쌍이 존재하는지 알고 싶다
- 이를 풀기 위해, 조금 더 간단한 문제를 생각해보자.
  - 정수가 저장된 두 개의 리스트 `X`, `Y` 와 특정 정수 `z` 에 대해, `x ∈ X, y ∈ Y` 를 선택해 `x + y = z` 가 되는 `(x, y)` 쌍을 존재하는지 검사하는 문제
  - 함수 `two_sum(X, Y, t)`는 그런 쌍이 하나라도 존재하면 `True` 를 리턴하고 그렇지 않으면 `False` 를 리턴하는 함수이다

- **입력**: 첫 번째 줄에 한 개 이상의 정수는 `A` 에 저장하고, 두 번째 줄에 주어진 한 개 이상의 정수는 `B` 에 저장하고, 세 번째 줄에 주어진 한 개 이상의 정수는 `C` 에 저장

- **출력**: `a ∈ A, b ∈ B, c ∈ C` 를 선택해 `a + b + c = 0` 이 되는 `(a, b, c)` 쌍이 존재하면 `True` 를 출력하고, 존재하지 않으면 `False` 를 출력

- **주석**: 코드의 수행 시간을 간략히 분석하고 Big-O 로 표기하시오 (주석 없으면 채점 점수도 **0 점**)

- **주의**: **O(n^3) 시간 알고리즘은 만점의 20% 점수만 부여**합니다

- **힌트**:
  - `two_sum(X, Y, t)`을 **O(n^2)** 시간에 동작하도록 하는 건 매우 쉽습니다. `Z` 의 값을 `t` 라고 하고 이 함수를 `n` 번 호출하면 문제가 풀리기에 **O(n^3)** 시간에 문제가 풀립니다

- **O(n^3)** 보다 더 빠르게 하려면 `two_sum(X, Y, t)`을 **O(n^2)**보다 더 빠르게 해야 합니다
- 우선 `two_sum(X, Y, t)`을 호출하기 전에 `X, Y`를 정렬합니다. (정렬은 전체 알고리즘을 통틀어 한 번만 하면 됩니다)
- 정렬된 `X, Y`를 가지고 `two_sum(X, Y, t)`을 호출합니다. 합이 `t`가 되는 `x, y` 원소를 찾는 것인데, `X, Y`는 정렬되어 있기에 두 리스트를 스캔하면서 선형 시간에 찾을 수 있습니다. 조금만 고민해보면 어렵지 않게 알 수 있습니다
- `two_sum(X, Y, t)`을 **O(n^2)**보다 더 빠르게 하는 다른 방법은 Hash Table 을 사용하는 것입니다. 정렬을 미리 하지 않고, 해시 테이블을 이용하는 것입니다. 이 경우에는 최악의 경우의 시간이 아닌 **평균 시간**으로 `two_sum(X, Y, t)`을 **O(n)** 시간에 수행할 수 있습니다
- 어떠한 방법이든 좋습니다. 두 가지 방법을 모두 구현해 수행시간을 별도로 비교해보면 더 좋습니다. ^^

입/출력 예시
:
공백
:
줄바꿈
:
탭

예시 1

입력

```
3123
-30-2
1-12
```

출력

```
True
```

예시 2

입력

```
4311
713
```

3-125

출력

False

[Selection+Heap] k 번째로 작은 수 찾기제출완료

1 초

- `n` 개의 값과 값 `k` 가 주어지면, `n` 개의 값 중에서 `k` 번째로 작은 수를 찾는 문제가 selection 문제라 한다
  - `n` 개의 값을 `A` 에 저장한다
- Selection 알고리즘은 여러 방향으로 설계할 수 있다
  1. `Algorithm_SORT`: `A` 를 오름차순으로 정렬한 후 `A[k-1]`를 리턴한다
     - **단점**: O(nlogn) 시간이 필요
     - **장점**: 단순 알고리즘
  2. `Algorithm_QUICK`: `quick_select(A, k)`를 호출한다
     - **단점**: 최악의 경우의 시간 O(n^2)
     - **장점**: 평균적인 시간 O(n)
  3. `Algorithm_MoM`: `MoM(A, k)`를 호출한다
     - **장점**: 최악의 경우의 시간은 O(n) < 최적의 시간
     - **단점**: 숨겨진 상수가 커서 실제 수행시간이 클 수 있다

  - 위의 세 알고리즘과는 조금 다른 방식을 생각해보자

  - `Algorithm_HEAP`: 힙을 이용한 selection 알고리즘
    1. `A` 의 값을 min 힙으로 만든 후: O(n) 시간
    2. 힙 성질을 이용해 `k` 번째로 작은 수를 찾는다: `O(?)` 시간
    3. 힙 성질 (heap property)은 부모 노드의 값이 자식 노드의 값보다 크지 않다는 것이다
- **입력**: 힙 성질을 만족하는 배열 `A` 의 값을 입력받고 `k` 값도 입력받는다
  - `1 <= k <= n`
  - `1 <= n <= 100,000`
- **출력**: `A` 에서 `k` 번째로 작은 값을 출력한다
- **할일**: `Algorithm_HEAP` 알고리즘의 두 번째 단계를 설계하는 것이다. 되도록 수행시간이 작아야 한다
- **주석**: 본인의 알고리즘을 설명하고 수행시간을 분석하는 주석을 반드시 포함해야 한다

- 다른 세 가지 알고리즘과 비교해 본인의 알고리즘의 장점과 단점도 설명해야 한다 --> 힙과 힙 정렬 부분 공부할 것!

- Python 의 `heapq` 모듈에서 min 힙을 제공한다. 필요하면 min 힙을 추가로 사용해도 된다

```
>>> import heapq
>>> A = [5, 4, 2, 7] # 리스트를 힙 자료구조로 사용한다
>>> heapq.heapify(A) # A 의 값들이 힙 성질 만족하게 됨
>>> A
[2, 4, 5, 7]
>>> heapq.heappush(A, 1) # insert 1 into A
>>> A
[1, 2, 5, 7, 4]
>>> A[0] # just print min of A (not deleted)
1
>>> heapq.heappop(A) # delete min key and return
1
>>> A
[2, 4, 5, 7]
```

입/출력 예시
:
공백
:
줄바꿈
:
탭
예시 1
입력

```
3
654321
```

출력

```
3
```

[AL] quick, merge, heap 정렬 비교해보기 제출완료

10 초

Quick, merge, heap 정렬을 모두 구현해보고, 시간과 비교/교환-이동 횟수를 다양한 `n` 에 대해 실행하여 비교해봅니다

- 세 가지 정렬 알고리즘을 작성합니다. 대부분의 코드는 교재와 동영상에 나와 있어요.
  - `quick_sort(A, first, last): A[first] ... A[last]` 까지 quick sort 하는 함수
  - `merge_sort(A, first, last): A[first] ... A[last]` 까지 merge sort 하는 함수
  - `heap_sort(A)`: `A` 의 값들을 heap sort 하는 함수 (heap 클래스를 정의할 필요 없이 함수로)

- 추가로 고려할 사항:
  - quick sort 와 merge sort 는 분할정복 알고리즘으로 하나의 값이 남을 때까지 분할하는 것이 기본 전략입니다. 그런데 굳이 하나가 남을때까지 분할할 필요는 없습니다. 경우에 따라서는 10 과 40 사이의 상수 K 에 대해서, K 개 이하가 되면 분할을 멈추고 insertion sort 로 정렬을 하면 더 빠르게 전체 정렬이 가능할 수도 있습니다
  - **[추가 점수]** 하나의 값이 남을 때까지 분할하는 quick, merge 정렬과 적당한 상수 K 개 이하가 될 때까지만 분할하는 quick, merge 정렬을 함께 구현해 비교해보세요

- 램덤한 수를 `n` 개 생성하여, 리스트에에 저장하여, 세 개의 정렬 함수를 호출해 정렬하고 다음 세 값을 각각 기록합니다.
  - 수행시간: `timeit` 모듈을 이용합니다 (샘플 코드에 나온대로 하면 됩니다)
  - 비교횟수: 두 수를 비교하는 횟수를 기록합니다.
  - 교환횟수: 두 수가 교환(**swap**) 또는 이동(**move**) 횟수를 기록합니다. `merge` 정렬에서의 **이동 횟수를 포함**해야 합니다

    - **hint**: 비교와 교환+이동횟수를 저장하는 변수를 `global` 변수로 선언해 기록하는 게 편함!

- `n` 의 값을 `n = 100, 500, 1000, 5000, 10000, 50000, 100000, 500000` 정도까지 다양하게 변화하면서 위의 세 가지 값이 어떻게 변하는지 기록하고 그 결과를 분석한 후 개인적인 느낌을 자유롭게 작성해봅니다. (도표나 그래프 등을 이용하면 더 효과적이겠죠?)

- 강의동영상과 교재에서 설명한 내용만으로 충분히 코드를 작성할 수 있기에 외부 코드를 참조하거나 사용하는 것은 금지합니다
- 코드는 컴파일 에러가 없어야 하고 올바르게 동작해야 합니다
- 코드는 구름에 제출하면 됩니다 (코드를 자동채점하는 게 아님에 유의. 제출만 가능)

- **레포트는 3 장을 넘지 않게 해서 eclass 에 제출**합니다
    - PDF 파일 형식으로 업로드해주길 바랍니다
    - 측정한 값들을 그래프 등의 시각화를 통해 분석해주기 바랍니다
    - 코드 자체를 레포트에 포함할 필요 없습니다
    - 표지는 만들지 말고요~

무조건 오름차순으로 만들어보자~제출완료

25 점
6 초

- **입력**: `n` 개의 정수 값이 주어진다. 이 값들을 리스트에 `A` 에 저장한다고 하자
  - `1 <= n <= 1,000`
  - `1 <= 정수 값 <= 200`
- 다음과 같은 연산을 생각해보자
  - `A[i] = A[i] + 1` 또는 `A[i] = A[i] - 1`
- `A` 의 값들에 연산을 원하는 만큼 적용해서 오름차순이 되도록 변경할 수 있다
  - 즉, `A[0] <= A[1] <= ... <= A[n-1]` 이 되도록 변경할 수 있다
- **출력**: `A` 의 값들이 **오름차순이 되기 위해 필요한** 연산의 **최소 횟수**
- **주석 1**: 본인이 작성한 알고리즘을 간단 명료하게 설명한다
- **주석 2**: 알고리즘의 수행시간을 간단히 분석한 후, 수행 시간을 Big-O 로 표기한다

입/출력 예시
:
공백
:
줄바꿈
:
탭
예시 1
입력

```
12321
```

출력

```
2
```

예시 2
입력

```
54321
```

출력

```
6
```

예시 3

입력

```
12244
```

출력

```
0
```

[주석필요] 작은 것들을 위한 시제출완료

40 점

- **입력**: `n` 개의 **서로 다른** 정수 값이 주어지면 리스트 `A` 에 저장
  - `1 <= n < = 10,000`
  - `m(i, j) = min(A[i], A[i+1], ..., A[j-1], A[j]) (i <= j)`라고 정의하자
- **출력**: 모든 인덱스 쌍 `(i, j)`에 대해 (`0 <= i <= j < n`) `m(i, j)`의 **합**을 계산해 출력

- **예**:
  - `A = [1, 2 -1, 4]`
  - `m(0,0) = 1, m(1,1) = 2, m(2,2) = -1, m(3,3) = 4`
  - `m(0,1) = min(1,2) = 1, m(1,2) = -1, m(2,3) = -1`
  - `m(0,2) = -1, m(1,3) = -1`
  - `m(0,3) = -1`
  - 따라서 전체 합은 `(1+2-1+4) + (1-1-1) + (-1-1) + -1` = `2`가 정답

- **주석 필요**: `O(n^2)`, `O(nlogn)`, `O(n)` 시간 등 여러 알고리즘이 존재한다. 본인이 작성한 알고리즘을 간략히 설명하고, 수행시간을 분석하시오

입/출력 예시
:
공백
:
줄바꿈
:
탭
예시 1
입력

```
12-14
```

출력

```
2
```

예시 2
입력

```
3124
```

출력

```
17
```

[DP] 히스토그램 (Histogram) 구하기제출완료

70 점
12 초

- 데이터 `i` 의 빈도수 `F[i]`가 입력으로 주어진다. 예를 들어, `F = [4, 2, 3, 6, 5, 6, 12, 16]`이라면 데이터 0 번은 4 번 등장하고, 데이터 1 번은 2 번 등장하는 식이다
- 우리는 빈도수를 최대 `B` 개의 연속된 빈도수의 그룹으로 묶으려고 한다. 예를 들어, `F[0] - F[3]` 그룹과 `F[4] - F[7]` 그룹, 두 그룹으로 묶는다고 하자. 이왕이면, 빈도수가 비슷한 값들을 하나의 그룹으로 묶어 빈도수의 원래 분포를 더 잘 반영하고 싶다
- 한 **그룹의 오차**(error)는 그룹의 빈도수의 평균 값과 개별 값의 차이의 제곱한 값으로 정의한다. 즉, (개별 빈도수 - 그룹의 평균 값)^2 의 합으로 정의한다
  - 예를 들어, 위의 두 그룹 `[4, 2, 3, 6]`과 `[5, 6, 12, 16]`에 대해서, 첫 번째 그룹의 평균 `3.75`, 두 번째 그룹의 평균 `9.75` 이다.
  - 첫 번째 그룹의 오차는 `(4-3.75)^2 + (2-3.75)^2 + (3-3.75)^2 + (6-3.75)^2 = 8.75` 이 된다
  - 두 번째 그룹의 오차는 같은 방법으로 계산해 보면, `80.75` 가 된다
- 한 **묶음의 오차**(error)은 각 그룹의 오차의 합으로 정의한다.
  - 위의 예에서 설명한 묶음의 오차는 `8.75 + 80.75 = 89.5` 이다

- 묶음을 다르게 해보면 묶음의 오차도 달라진다
  - 이번에도 두 그룹으로 나누는데, 첫 번째 그룹은 `[4, 2, 3, 6, 5, 6]`, 두 번째 그룹은 `[12, 16]`으로 해보자
  - 같은 방식으로 묶음의 오차를 계산해보면 `21.3333` 으로 앞에서 설명한 묶음의 오차보다 작다!

- 여러분은 오차가 최소인 묶음을 찾아야 한다
  - 이 문제를 **히스토그램 (묶음) 문제**라 한다

- **입력**: 첫 줄에는 최대 그룹 수 `B` 와 `n` 이 주어진다. 다음 `n` 개의 줄에는 빈도수 `F[i]`가 주어진다
  - `1 <= B <= 30,  1 <= n <= 1,000`

- **출력**: 최대 B 개의 그룹으로 F 의 연속된 값으로 그룹을 만드는데, 오차가 최소인 묶음을 찾아 최소 오차 값을 출력
    - 단, 소수점 아래 4 째 자리에서 반올림해서 3 째 자리까지만 출력한다! (`round` 함수 이용)
    - `round(x, 3)`이라고 하면 x 의 소수점 아래 4 째자리에서 반올림해서 3 째 자리까지만 계산한다

- **힌트:** 한 그룹의 오차를 식으로 표현해 간단한 형식으로 정리해보면, 빠른 시간에 계산할 수 있는 방법을 찾을 수 있다

- **주석**: 알고리즘을 간략히 설명하고 수행시간 분석을 하세요. (필수)

입/출력 예시
:
공백
:
줄바꿈
:
탭
예시 1
입력

```
28
4
2
3
6
5
6
12
16
```

출력

```
21.333
```

예시 2
입력

```
38
1
2
3
4
5
6
7
8
```

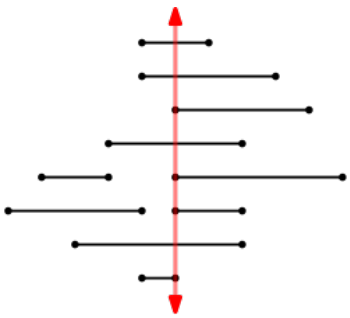
출력

```
4.5
```

못 하나로 관통할 수 있는 막대의 최대 개수 구하기제출완료

30 점
5 초

- `n` 개의 막대가 입력으로 주어질 때, 오직 **하나**의 못(pin)만을 사용해 가능하면 많은 개수의 막대를 꽂으려고 한다. 아래 그림에서는 붉은색 위치에 핀을 꽂으면 8 개의 막대까지 꽂을 수 있다
- 여러분은 하나의 핀만으로 꽂을 수 있는 최대 막대 수를 계산해 출력해야 한다
    - [**주의**] 못이 막대의 끝을 통과하더라도 꽂은 것으로 간주한다.
    - 아래 그림은 두 번째 샘플 데이터 케이스를 그린 것이다
    - 못이 막대의 여러 끝 점을 지날 수도 있기에 이런 예외적인 경우를 주의해서 구현해야 한다



- **입력**:
    - 첫 줄에는 값 `n` 이 주어진다.
        - `n` 은 1 이상 100,000 이다
    - 둘째 줄부터 `n` 개의 구간의 왼쪽 끝 점 `a` 와 오른쪽 끝 점 `b` 의 좌표 값이 차례대로 주어진다
        - 이 두 값의 범위는 0 이상 200,000 이하이며, 항상 `a < b` 이다
    - 동일한 끝 점을 갖는 구간이 두 개 이상 나타날 수 있다

- **출력**: **한 개의 못으로 꽂을 수 있는 최대 막대 개수**

- **주석:** 자신의 알고리즘을 간략히 설명하고, 수행 시간을 분석하세요. 수행 시간이 빠를 수록 좋다!

입/출력 예시

:
공백
:
줄바꿈
:
탭

예시 1

입력

```
10
23
24
35
46
56
67
79
910
1011
1012
```

출력

```
3
```

예시 2

입력

```
10
79
49
26
712
67
59
35
711
610
68
```

출력

```
8
```

예시 3

입력

```
10
1123
820
1327
510
1225
914
1422
313
210
916
```

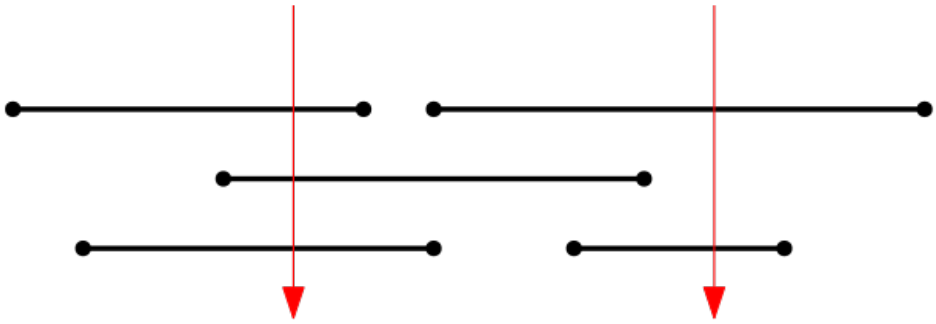
출력

7

[Greedy] 못 박기 (Pinning)제출완료

30 점

- 아래 그림처럼 `n` 개의 막대가 입력으로 주어질 때, `n` 개의 막대를 최소 개수의 못(pin)으로 꽂으려 한다. 모든 막대는 최소 하나 이상의 못이 박혀야 한다. 이 때, 필요한 핀의 최소 개수는 몇 개일까?
  - 아래 그림에서는 2 개의 못으로 모두 꽂을 수 있다
  - 못이 막대의 끝을 통과하더라도 꽂은 것으로 한다



- **입력**:
  - 첫 줄에는 값 `n` 이 주어진다.
    - `n` 은 1 이상 100,000 이다
  - 둘째 줄부터 `n` 개의 구간의 왼쪽 끝 점 `a` 와 오른쪽 끝 점 `b` 의 좌표 값이 차례대로 주어진다
    - 이 두 값의 범위는 0 이상 200,000 이하이며, 항상 `a < b` 이다
  - 동일한 끝 점을 갖는 구간이 두 개 이상 나타날 수 있다

- **출력**: **최소 못의 개수**

- **힌트: 생각보다 어렵지 않아요~**
  - 위의 그림에서는 두 개의 못이 필요하다. 왼쪽 못을 현재의 위치에서 약간 오른쪽으로 이동해도 여전히 같은 막대를 관통한다. 이 논리를 계속 적용하면 못의 위치를 몇 군데 중 하나로 제한해도 된다는 사실을 알게 된다. 이 사실을 이용해보자!
  - 강의실 배정 문제와 얼마나 유사할까?

- **주석:** 자신의 알고리즘을 간략히 설명하고, 수행 시간을 분석하세요.

입/출력 예시
:
공백
:
줄바꿈
:
탭
예시 1
입력

```
10
23
24
35
46
56
67
79
910
1011
1012
```

출력

```
4
```

예시 2
입력

```
10
79
49
26
712
67
59
35
711
610
68
```

출력

```
2
```

예시 3
입력

```
10
1123
820
1327
510
1225
914
1422
313
210
916
```

출력

2

[AL] 0/1 Knapsack 문제제출완료

10 초

1. **입력**:
    1. 첫 줄에 `K` (배낭의 크기)
    2. 둘째 줄에 `n` (아이템 갯수로 `300` 을 넘지 않는다)
    3. 셋째 줄에 `n` 개의 크기 (양의 정수)
    4. 넷째 줄에 `n` 개의 가치 (양의 정수)
2. **출력**:
    1. 최대 가치(`MaxProfit`)를 정수로 출력
3. 주의:
    1. **DP** 가 아닌 백트래킹 방법으로 구현해야 함
    2. 교재, 동영상 강의, 구름 강의에서 설명한 알고리즘과 pseudo 코드를 참조해 작성함
    3. 한계함수를 위해선 fractional knapsack 알고리즘의 결과를 이용해야 함

입/출력 예시
:
공백
:
줄바꿈
:
탭

예시 1

입력

```
16
4
25105
40305010
```

출력

```
90
```

예시 2

입력

```
10
5
721024
4619304911
```

출력

```
95
```

[AL] 미로 탈출 (4 방향)제출완료

- Backtracking 방법으로 입구에서 출구까지 탈출이 가능한지 알아보고, 가능하다면 탈출로를 찾아보자!
- **입력**
  - 첫 줄: `n  # n x n` 미로라는 의미
  - 두 번째 줄: `sx sy ex ey  # 입구 칸 (sx, sy), 출구 칸 (ex, ey)`
  - 세 번째 줄부터 `n` 개의 줄은 미로를 나타낸다 (`1` 은 장애물, `0` 은 빈칸을 의미한다)
    - 각 줄은 길이가 `n` 인 `0` 과 `1` 로 구성된 문자열임
    - 미로는 이차원 문자열 리스트 `M` 에 저장함
- 여러분은 `find_way_from_maze` 함수를 호출하여, `M` 에 탈출 경로를 찾아
  - 입구 칸에는 '`s`'를, 출구 칸에는 '`e`', 그 외 탈출 경로에 해당하는 칸에는 `trace` 를 저장한다
  - `trace = '\u00B7'` 으로 dot 을 나타내는 문자로 선언되어 있으며, 탈출 경로를 그리기 위한 marker 역할을 한다
  - 탈출 경로가 있으면 `True`, 없으면 `False` 를 리턴한다
  - 주의: 현재 칸에서 동, 서, 남, 북 네 방향으로 인접한 빈 칸으로 이동가능하며, 이동할 때에는 반드시 **동쪽 -> 남쪽 -> 서쪽 -> 북쪽** 순서로 검사하여 이동해야 한다
- **출력**
  - 샘플 코드의 입출력 부분은 수정하지 말고 그대로 사용할 것!

Pseudo Code:

```
find_way_from_maze(r, c) # 현재 칸 (r, c) 방문 중
  visited[r][c] = True
  if (r, c) == exit: return True
     if 동쪽 이웃 칸이 빈 칸이고 미방문이라면:
          if find_way_from_maze(r, c+1):
                 M[r][c+1] = trace  # trace 는 dot 으로 탈출경로를
표시하기 위함
                 return True
  if 남쪽 이웃 칸이 빈 칸이고, 미방문이라면:
          ...
  if 서쪽 이웃 칸이 빈 칸이고, 미방문이라면:
```

```
        ...
if 북쪽 이웃 칸이 빈 칸이고, 미방문이라면:
        ...
return False
```

입/출력 예시

:

공백

:

줄바꿈

:

탭

예시 1

입력

```
7
1155
1111111
1000001
1111101
1000101
1011101
1000001
1111111
```

출력

```
#######
#s····#
#####·#
##·#
####·#
#e#
#######
```

예시 2

입력

```
7
3355
1111111
1000001
1111101
1000101
1011101
1000001
1111111
```

출력

```
#######
##
######
#··s##
#·####
#····e#
#######
```

예시 3

입력

```
11
3399
11111111111
10100000101
10101110101
10101010101
10111010101
10001000101
10111011101
10001010001
10101011101
10100000001
11111111111
```

출력

```
###########
##·····##
##·###·##
##s##·##
#####·##
##···##
####·####
##·##
###·####
##····e#
###########
```