83

! * 에 몇글자가 대응되는지 알아본다. 의 의미.

↳ 주어진 패턴 : t*l?*gr?ng*s 주어진 문자열 : thelord.frings

쪼갠 조각 수 : {t*, l?*, o*, r?ng*, s}

- t* 에 대응되는 글자 수 찾기

=> t, th, the

l이 나오면 조각 끝나짐.

- match(w, s) 해석.

w : 와일드카드 s : 원문. pos : 현재 위치.

while문 종료조건 (매칭 불가능)

1. w[pos] 와 s[pos]가 대응되지 않는다.

와일드카드 w가 찾는 문장에 s는 포함되지 않는단 소리.

2. w 끝에 도달했다.

w 내에 '*'이 하나도 들어 있지 않다 = w와 s의 길이가 정확히 같아야 함.

3. s 끝에 도달했다.

w는 남았는데 s가 끝나버림. w의 남은 패턴이 전부 *이지 않은 이상 거짓.

4. w[pos]가 *인 경우.

*에 몇글자가 대응될지 알 수 없으므로 아무것도 안들어가는 것부터 남은 문자열 끝까지 완전 탐색. => 그래서 이거 어떻게 하는건데 !!! 101 ↳ 완전탐색 후 스킵할 개수인듯.

이때 w[pos+1] 이후를 w으로 하고 s의 [pos+skip] 이후를 s로 해서 match(w,s) 에 재귀호출했을 때 답이 하나라도 참이면 답은 참이 됨.

↳ 다음들은 똑같이 검사한단 소리잖아..?

↳ skip을 하나씩 늘려 가며 일일이 재귀합니다 ^^!

=> 너의 단점은.. ☆

w가 ******a 고 s가 aaaaaaaaab 같을 경우. 극한으로 오래 걸릴 수 있음.

*이다가 저 a들 다 넣어 볼텐데 * 짱많음.

=> 캐시를 이용해서 중복 계산을 제거하자!!

+ 재귀를 더 갖다박아서 시간복잡도를 줄이자! (????)

O(n!) 이 O(n²) 이 됨!! (224p.)

8.4 이 하게시리 적어가는

아. 애는 입력부터 어떻게 받을지 심난하다 진짜..

삼각형모양 수의 집합에서, 아래 혹은 오른쪽 아래로 내려갈 수 있다고 할 때 최대합을 만드는 경로 찾기.

일단 가로로 짤라서 아래로 갈지 오른으로 갈지 정하면서 완전탐색.

그럼 나오는 점화식.

아래로 내려감

$$path1(y,x,sum) = \max \begin{cases} path(y+1, x, sum + triangle[y][x]) \\ path(y+1, x+1, sum + triangle[y][x]) \end{cases}$$

오른 아래로 내려감.

=> 이러면 한 줄 생길때마다 걸리는 시간 2배씩 늘어남..

그냥 캐싱하는 메모이제이션 또한 경로마다 합이 다 다를테니 완전탐색과 동일해짐..

↳ 매개변수로 y, x, sum을 받기 때문에 sum도 확인해서인듯.

그래서!! 관점을 바꿔 만든 Path2 = (y,x)에서 시작해서 맨 아래까지 내려가는 부분 경로의 최대 합.

↳ 매개변수 sum을 제거하기 위한 관점의 변화.

→ Path1과의 차이점 : Path1은 여태까지 전체 경로의 최대 합을 구한다면,

Path2는 앞으로 남은 놈들 가지고만 계산함. + 메모이제이션.

$$path2(y,x) = triangle[y][x] + \max \begin{cases} path2(y+1, x) \\ path2(y+1, x+1) \end{cases}$$

지금 나 내 다음 쉑

시간복잡도 $O(2^n) \longrightarrow O(n^2)$!!

=> 이렇게 푸는 방법을 "최적 부분 구조"라고 합니다.

결과가 어떤 경로로 이루어졌든 남은 부분 문제를 전부 최적으로 풀어도 된다.

최단 경로 문제도 "최적 부분 구조"가 성립하는 문제 중 하나임.

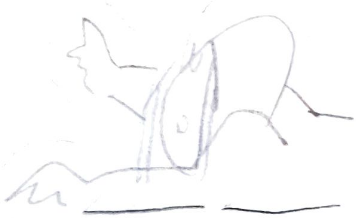
단, 부가조건이 붙는다면 그건 만족하지 않을 수도 있음 (ex) 비용

#3 최대증가 부분 수열 찾기 : 유명한 동적 계획법 문제 중 하나.

응용이! STL의 map은 레드블랙트리.

이러면 STL 애들 자체가 전체적으로 느릴 가능성 ↑.

나중에.. 코드 해석하기 싫어 ㅠ0ㅠ.



push_back

이거 뭐더라.

! push_back() 함수. (vector 꺼.)

ex) B.push_back(A[i]);

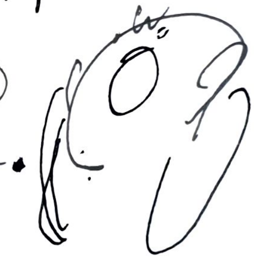
B의 맨 마지막 원소 뒤에 A[i]를 추가함.

아 그렇구나.. 신기한데 모르겠어.. ㅠ0ㅠ

? S[-1]이 -∞이면 왜 모든 시작 위치를 자동으로 시작해..?

아 시작지점을 -1로 아예 정해뒀네

-1이면 시작하라고 시키네



신박!

그래서 ~~len~~ lis3(-1)-1.

→ $O(n^2)$

좋아했어.

하지만! 더 좋은 애가 있다! $O(n\log n)$짜리.

ㄱ.. C[i]가.. 지금까지 만든 부분 배열이 갖는 길이 i인 증가 부분 수열 중 최소의 마지막 값이라는데..

예시라도 줘라 개짧은거 ㅠ

(8.5) JLIS 문제. 8.6도 네

2개의 정수 수열 A와 B에서 각각 최대 증가 부분수열을 찾아 오름차순 정렬 한 거

= 합친 증가 부분 수열.

이중 가장 긴 수열을 합친 LIS의 길이를 구하는 프로그램 만들기.

2sec / 64MB 제한

- ∞ 표현 위해.. numeric_limits<long long> :: min()

이게 쉬워..

Wow.. * numeric_limits 타입 : (자료형) 변수의 최대 혹은 최소 표현가능 숫자를 리턴.

numeric_limits (int) :: max (); => int 형 최대 표현가능 숫자 리턴.

좀 많이 유용해 보인다..

야 코드 비교시킬 거면 적어도 옆에 붙이는 거라!!! 1이

- 앞의 LIS 문제를 이용해 풀겁니다 라고합니다..

①. lis3(start) = S[start]에서 시작하는 최대증가 부분 수열의 길이.

②. jlis (indexA, indexB) = A[indexA]와 B[indexB]에서 시작하는 합친 부분 수열의 최대 길이.

여기서부터가 문젠데.. 진짜 자살하고싶다 (˘・˘)

(여기서!) 이 증가 부분 수열의 다음 숫자는 A[indexA+1] 이후 OR B[indexB+1] 이후의 수열 중 max (A[indexA], B[indexB])보다 큰 수 중 하나가 됨!

// 네 이해했습니다

어.. 그 다음에 A에 있는 놈을 다음 숫자로 선택했을 경우 합친 증가 부분 수열의 최대 길이는 1+ jlis (next A, index B)가 됨.. 다음 A 놈보다 큰놈 찾아야 하니까.. B는 선택 안했으니 그대로 그리고 next A가 수열에 추가 되었으니 +1 하는듯.

=> 그래서 나온 점화식.

$$jlis(indexA, indexB) = \max \begin{cases} \max_{nextA \in NEXT(indexA)} jlis(nextA, indexB) + 1 \\ \max_{nextB \in NEXT(indexB)} jlis(indexA, nextB) + 1 \end{cases}$$

얘네 뭔데.. 1이

걍 index A, B의 다음 index 범위 내에 포함된다는거로 이해하..

· jlis (int indexA, int indexB) 정리.

① indexA의 A 원소와 indexB의 B원소 중 큰 놈을 MaxElement로 지정.

② MaxElement 보다
- nextA가 크면
  좌표를 (nextA, indexB)로 바꿔 재귀하고 ret의 길이를 1 늘림,
  그리고 지금의 ret과 비교 후 큰놈 선택.
- nextB가 크면
  좌표를 (indexA, nextB)로 바꿔 재귀하고 ret 길이를 1 늘림,
  그리고 지금의 ret과 비교 후 큰놈 선택.

(nextA/B의 몫이다.)

이거 아님! ~~* cache 되어있던 ret이 호출되었을 수 있기 때문에 반드시 필요.~~

아닌거같은데..? ret과 jlis(어쩌구)+1 은 왜 비교하지?

③ return ret.

초기값을 (-1, -1)로 주기 때문이다! 함수 초반에 ret을 2로 주기 때문에 cache[0][0]이자 ret의 초기값은 2.

(x) 1부터 ... 0부터 ... 길이 비교

반복문 한 번 돌 때마다 ret이 바뀌잖아!

나중에 만날 때 보다 그 서로 다른 줄에 있던 경우가

indexA 와 indexB의 처음 수열 ...

+ 메모이제이션의 원리.

① ret의 주소를 cache[indexA+1][indexB+1]의 주소로 둠. (값이 같이 변경된다.)

② ret을 계산하여 ret에 값이 저장될 때마다 각 cache[indexA+1][index B+1]에도 값이 저장됨.

③ 만약 호출된 적 있는 index의 재귀가 이루어지면 그 index의 Cache 값은 -1이 아님. ①을 통해 ret에 Cache[indexA+1][indexB+1] 값이 들어있기 때문에 검사 가능.

④ 그걸 재귀함수 초반에 물어봄으로써 -1이 아닌 Cache 값은 그대로 반환 후 함수 종료. (뒤에꺼 안하고)

· C++의 STL에 대해서..!

- STL : 일종의 데이터 다루는 종류의 라이브러리들, 데이터 정리에 유용하게 쓰는 애들.
  · STL 컨테이너 : 클래스 형식으로 정의되는 애들, 일종의 컬렉션 클래스,
  · STL 알고리즘 : 자주 사용하는 유용한 알고리즘들을 함수 템플릿으로 정의해 둔 것.

STL 컨테이너
- 순차 컨테이너 : 데이터의 순차적 저장.
  vector, list, deque ...
- 연관 컨테이너 : 데이터의 저장 뿐만 아니라 일정 규칙에 따라 정리.
  set, map ...
  ↳ 얘네는 multi 붙으면 중복값을 허용하게 된다.
- 어댑터 컨테이너 : 순차 컨테이너를 변형, 데이터를 미리 정해진 방식에 따라 관리.
  stack, queue, priority_queue ...

~~* itera~~ * Iterator (반복자!) : 포인터처럼 STL 컨테이너의 특정 위치를 나타냄.

STL의 장점 : 컨테이너의 종류가 다르더라도 컨테이너가 제공하는 인터페이스가 같다.

↳ STL 이라면 STL 알고리즘을 쓰면 된다는 것. .push_back, .begin, .end, .back 등

STL 알고리즘 : .binary_search, .find, .for_each, .max, .min, .remove, .replace, .sort, ...