String Algorithm

14004 구재현

Table of Contents

Why String?

- 1. 스트링 문제가 ioi 빼고 다나오는듯 ㅎㅎ
- 2. 스트링 문제는 모르면 접근도 못함
- 3. 재밌음 ㅎ
- 4. 돈이 많이 됨

1. Hashing

- 해싱은 “알아둬야 합니다.” 없이 못 푸는 문제들이 존재
- 해싱을 통해서 문자열 s의 “Substring”을 빠르게 비교할 수 있음
- 약간 부분합같은 느낌이라고 생각하면 좋음
- 일단, 해싱이 뭔지부터 알아보자

Rabin-Karp Hash

- asdfghjkl -> Hashing?
- a * 26^8 + s * 26^7 + d * 26^6 ……
- 26진수! 무슨 느낌인지 알겠죠?
- 하지만, 저장할 수 있는 수에 한계가 있으니 적당히 모듈러를 때립니다.
- Hash[i] = Hash[i-1] * 26 + (Character) 라는 배열을 생각

Rabin-Karp Hash 2

- Hash[i] = Hash[i-1] * 26 + (Character) 라는 배열을 생각
- 구간 [s, e]에 있는 Substring의 Hash값은?
- Hash[e] - (Hash[s-1] * 26 ^ (e - s + 1))
- Substring Hash is O(1).

ex) Longest Palindrome in nlgn.

- 길이는 홀수라고 맘대로 가정 (짝수도 잘 하면 됨…)
- 임의의 위치 i에 대해서, S[i - p, i] == Reverse(S[i, i+p]) 를 만족하는 가장 큰 수 p를 모두 찾을 수 있다면 풀 수 있음.
- Reverse한 문자열을 가지고 있으면, Rabin-Karp 해시를 O(1)에 비교 가능.
- p를 Binary Search (Parametric Search)로 찾으면 문제 해결 가능.
- 해싱을 한 규칙이 똑같다면 문자열이 달라져도 상관없음. 당연하죠?

ex) 문자열에서 두번 등장하는 가장 긴 부분문자열

- sagasagasaga -> sagasaga = [0, 7] and [4, 11]
- assume the length is p. (parametric search)
- p에서 가능 -> p-1에서는 항상 가능. p에서 불가능 -> p+1에서 항상 불가능하니. 이진 탐색.
- 그러면, 문자열에서 두번 등장하는 길이 p의 부분 문자열이 존재하는가는?
- O(n)의 경우의 수가 있으니, Rabin Karp Hash를 그때그때 구한 후, 중복 원소가 존재하면 빼버리면 됨.
- O(lgn) * O(nlgn) = O(nlg^2n)

Technical Difficulties

- 일단 해싱을 사용하면 답이 틀릴 수 있음. 해싱은 대충 스트링을 우겨넣는 방식이기 때문에, 충돌이 있다면? (google for birthday problem)
- 하지만 요즘 해싱을 쓰는 건 그냥 취향 차이라고 생각하는 경우가 많은듯. 대회 정해도 해시로 나오는 경우가 많음 (ex : USACO Censoring)
- 해싱을 사용하면 2^32 / 2^64의 modular를 쓰고 싶은 경우가 많을 거임 (귀찮으니까..) 이러한 해싱 방법에 무조건 WA를 박을 수 있는 데이터가 존재. 모듈러는 무조건 소수로!!

KMP Algorithm

- Knuth-Morris-Pratt Algorithm.
- KMP Algorithm은 문자열 S가, 문자열 T의 부분문자열인지를 판별하는 것이고, O(|S| + |T|) linear time에 작동.
- 문제 자체는 해싱으로도 풀 수 있음.
- KMP Algorithm의 핵심은 Failure function. 이게 제일 중요함. Failure function의 개념은 Aho-corasick 등에도 그대로 응용.

Failure function

- Fail[i] = maximum p < i s.t S[0, p-1] == S[i-p, i-1]

- Failure function을 O(|S|^3)에 구할 수 있음. 일단은 그걸로 넘어감.

- 이제 이게 T에서 S를 찾는 데 무슨 도움이 될까?

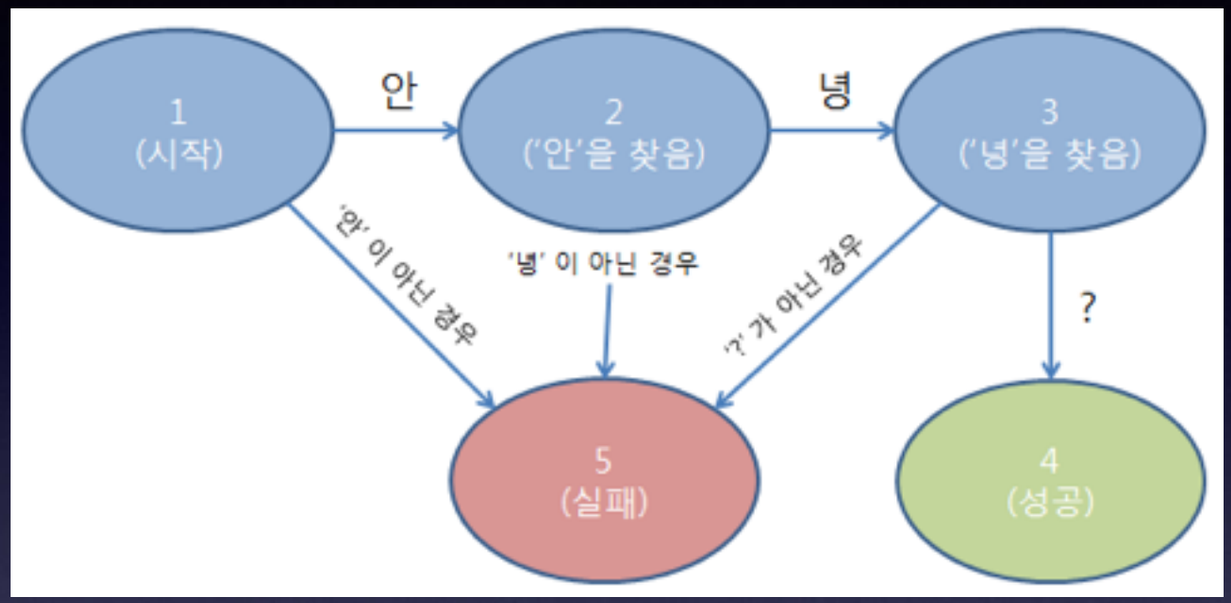
KMP Algorithm

- int matched = 0;
- for all character in T
- if(현재 character가 S[matched]와 같으면):
- -> matched++, matched == |S|일 경우 매칭 성공
- 아니면 :
- **matched = fail[matched - 1]! <<< 무슨 의미를 가지는가?**

Calculating Failure Function Faster

- O(|S|)에 하는 방법은 간단함.
- |S|[1] 과 |S|[0]의 KMP를 돌리면 됨!
- 이 과정에서 |S|의 Failure function이 없는데??
- 하지만, 지금 필요한 Failure function은 무조건 이미 구해놨다!!
- 고로 O(|S|)에 Failure function 계산 가능.

Finite State Automata



- Finite State Automata는 입력에 따라서 다음 위치가 변하는 식으로 움직이는 기계.
- 모든 입력이 주어진 후 Automata가 어디에 있는가가, 성공과 실패를 가름.
- 행렬로써의 표현도 가능함. (practice : 비결정 유한 상태 오토마타 in O(2^18 * lgN)

Trie / Aho-Corasick

- 님들도 Suffix Array 하고 싶죠?
- ㅇㅇ 나도 그럼. 시간 관계상 생략
- 종만북 ㄱㄱ

Trie / AC 연습 문제

- https://www.acmicpc.net/problem/5467
- https://www.acmicpc.net/problem/9250
- https://www.acmicpc.net/problem/10256
- http://www.koistudy.net/?mid=prob_page&NO=911
- https://www.acmicpc.net/problem/10745

Suffix Array가 왜 필요할까요?

- 모든 부분문자열은 Suffix의 Prefix.
- 어떠한 문자열이 부분 문자열임을 빠르게 판별하려면??
- Suffix 상에서 이진탐색을 하고 Prefix랑 얼마나 매치되는지를 빠르게 하는게 된다면…
- 이라는 입풀이를 해봅니다 ㅠㅠ
- -> 입풀이를 현실화하기 위해서 만들어진게 Suffix Array!

Suffix Array

- banana 의 Suffix 6개를 모으자!
- banana (0)
- anana (1)
- nana (2)
- ana (3)
- na (4)
- a (5)

Suffix Array

- a (5)
- ana (3)
- anana (1)
- banana (0)
- na (4)
- nana (2)
- 사전순 정렬. 이제 Suffix Array는 [5, 3, 1, 0, 4, 2].
- O(n^2lgn)에 구현 가능.

ex) 문자열에서 두번 등장하는 가장 긴 부분문자열 2.0

- 문자열에서 두번 등장하는 가장 긴 부분문자열?
- Suffix의 Prefix가 부분문자열이라는 말을 다시 상기
- Suffix 두개를 잡아서. 가장 Prefix가 많이 매치되는 걸 찾으면 그게 가장 긴 부분 문자열임.
- 하지만 Suffix Array가 있다면, Suffix Array에서 인접한 것만 보는게 무조건 이득!

Longest Common Prefix

- 두 부분문자열 [p1, p1 + t-1] == [q1, q1 + t-1] 를 만족할때 가장 긴 t의 길이는?

- = Longest Common Prefix.

- Suffix Array가 있으면 인접한 두 원소의 Longest Common Prefix만 알아도 도움이 됨.

- 나머지 원소들의 LCP는 그것에 의해서 결정되기 때문.

- 앞 문제는 SA 상에서 인접한 LCP만 봐서, 그 중 Maximum만 취해도 풀림.

ex) 문자열에서 두번 등장하는 가장 긴 부분문자열 2.0

- Suffix Array가 주어지면, 인접한 원소끼리 LCP를 계산하는 것이 O(n)에 가능함.
- 설명하기 싫음
- http://blog.naver.com/dark__nebula/220419358547
- LCP 중 Maximum을 구하면 문제를 풀수 있음
- Suffix Array를 구하는 속도가 문제를 푸는 속도를 결정

Calculate Suffix array faster

- O(n^2lgn)에 SA를 계산하는 방법은 설명함.
- O(nlg^2n)에 쉽게 SA를 계산할 수 있음. (코딩이 쉽지 알고리즘이 쉬운지는…)
- http://blog.naver.com/dark__nebula/220419318870
- O(nlgn)도 가능함!!
- http://blog.myungwoo.kr/57
- O(n)에 할 수 있음 (Ukkonen’s Algorithm)

Longest Common Prefix. Again

- 인접한 두 원소간의 LCP를 알았는데… 인접하지 않은 두 원소간의 LCP를 구하는 방법은?

- LCP[i, j] = Min(LCP[i, i+1] , LCP[i+1, i+2]… LCP[j-1, j])

- 유명한 Range Minimum Query 문제로. 인덱스 트리를 사용해서 O(QlgN)이나 O(Q) 에 문제를 해결 가능.

- Range Minimum Query == Longest Common Prefix == Lowest Common Ancestor. Why? (hint : Suffix들로 Trie를 만들면?)

Manacher’s Algorithm

- O(n) algorithm for finding palindrome
- http://blog.myungwoo.kr/56