Suffix Array

나정휘

https://justiceHui.github.io/

목차

- Prerequisites
- Motivation
- Suffix Array
- Algorithms
- Applications

Prerequisites

- Trie
- Radix Sort
- Binary Search
- (optional) Segment Tree, Sparse Table

Motivation

Motivation

- 부분 문자열
  - 지금까지 공부한 문자열 알고리즘을 생각해 보자.
    - KMP
      - 문자열 S, P가 주어지면 P와 동일한 S의 부분 문자열을 O(|S| + |P|)에 찾는 알고리즘
    - Manacher
      - 문자열 S가 주어지면 각 문자를 중심으로 하는 가장 긴 팰린드롬 부분 문자열을 O(|S|)에 찾는 알고리즘
    - Aho-Corasick
      - 문자열 S와 패턴 $P_1, \ldots , P_k$가 주어지면 P_i와 동일한 S의 부분 문자열을 O(|S| + sum $|P_i|$)에 찾는 알고리즘

  - 부분 문자열에 대해 더 고찰해 보자.

Motivation

- 부분 문자열
  - 부분 문자열 = 접두사의 접미사 = 접미사의 접두사
  - 접두사를 관리하는 자료구조 = Trie

  - 문자열의 모든 접미사를 Trie로 관리하면 어떤 좋은 점이 있을까?

Motivation

- 트라이(Trie)
  - Trie의 각 정점은 Trie를 구성하는 문자열들의 접두사에 대응됨

  - 중요한 성질
    - 문자열 S_1과 S_2의 최장 공통 접두사는 Trie에서 LCA를 구하면 됨
    - terminal node에만 관심이 있다면, 경로들을 압축해서 O(N)개의 정점만 남겨둘 수 있음
      - N = 문자열의 개수
    - Trie를 구성하는 문자열이 모두 미리 주어진다면, 문자열들을 정렬해서 비슷한 효과를 볼 수 있음
      - 같은 조상을 공유하는 문자열끼리 인접한 위치에 있음

Motivation

- 접미사 트리 (Suffix Tree)
  - 문자열 S의 모든 접미사를 넣은 Trie
    - O(N)에 구할 수 있지만 지금은 신경쓰지 말자.

  - 해결할 수 있는 문제
    - 문자열 매칭
      - 문자열 S와 Q개의 문자열 P_1, P_2, …, P_Q가 주어지면, O(sum $|P_i|$) 시간에 P_i가 S의 부분 문자열인지 판별

    - 최장 공통 접두사
      - 문자열 S의 부분 문자열 S[a..b], S[c..d]의 최장 공통 접두사의 길이를 O(log N) 시간에 구하는 문제

Motivation

- 접미사 트리 (Suffix Tree)
  - 문자열 S의 모든 접미사를 넣은 Trie
    - O(N)에 구할 수 있지만 지금은 신경쓰지 말자.

  - 해결할 수 있는 문제
    - 문자열 매칭
      - 문자열 S와 Q개의 문자열 P_1, P_2, …, P_Q가 주어지면, O(sum $|P_i|$) 시간에 P_i가 S의 부분 문자열인지 판별
      - 단순히 Trie를 순회하면 됨
    - 최장 공통 접두사
      - 문자열 S의 부분 문자열 S[a..b], S[c..d]의 최장 공통 접두사의 길이를 O(log N) 시간에 구하는 문제
      - S의 접미사 S[a..]와 S[c..]의 최장 공통 접두사를 구하면 되고, Trie에서의 LCA를 구하면 됨

Motivation

- 접미사 트리 (Suffix Tree)
  - 아주 강력한 자료구조지만 너무 어려움

  - Trie의 성질을 다시 한번 생각해 보면…
    - Trie를 구성하는 문자열이 모두 미리 주어진다면, 문자열들을 정렬해서 비슷한 효과를 볼 수 있음
    - 즉, 문자열 S의 모든 접미사를 정렬해도 비슷한 효과를 볼 수 있음

  - ex) 문자열 S에서 패턴 P를 찾는 문제
    - P로 시작하는 S의 접미사 sfx_1, sfx_2, … 를 모두 찾은 다음
    - sfx_i의 길이가 |P|인 접두사를 구하면 됨
    - 모든 접미사를 정렬하면 P로 시작하는 접미사는 서로 인접함

질문?

Suffix Array

Suffix Array

- 접미사 배열
  - 문자열의 모든 접미사를 사전순으로 정렬한 자료구조
    - ex) banana: a / ana / anana / banana / na / nana
  - $O(N^2 \log N)$, $O(N^2)$, $O(N \log^2 N)$, $O(N \log N)$, $O(N)$에 구할 수 있음
    - 대회에서는 주로 $O(N \log^2 N)$이나 $O(N \log N)$ 알고리즘 사용
    - 이건 뒤에서 알아보자.

Suffix Array

- 접미사 배열
  - 문자열 S에서 패턴 P를 찾는 문제
    - S#P의 접미사 배열을 구한 다음
    - 정확히 P만 포함하는 접미사를 포함하는 적당한 구간이 정답의 전부
    - ex) S = “abcabcbc”, P = “bc”, S#P = “abcabcbc#bc”
      - #bc / abcabcbc#bc / abcbc#bc / bc / bc#bc / bcabcbc#bc / bcbc#bc / c / …
    - 각각의 접미사가 P로 시작하는지 판단하는 방법은?

Suffix Array

- 접미사 배열
  - 문자열 S에서 패턴 P를 찾는 문제
    - S#P의 접미사 배열을 구한 다음
    - 정확히 P만 포함하는 접미사를 포함하는 적당한 구간이 정답의 전부
    - ex) S = “abcabcbc”, P = “bc”, S#P = “abcabcbc#bc”
      - #bc / abcabcbc#bc / abcbc#bc / bc / bc#bc / bcabcbc#bc / bcbc#bc / c / …
    - 각각의 접미사가 P로 시작하는지 판단하는 방법은?

- LCP 배열 (Longest Common Prefix Array)
  - 접미사 배열 상에서 인접한 두 접미사의 최장 공통 접두사의 길이를 저장한 자료구조

Suffix Array

- LCP 배열 (Longest Common Prefix Array)
  - 접미사 배열 상에서 인접한 두 접미사의 최장 공통 접두사의 길이를 저장한 자료구조
    - ex) banana: a / ana / anana / banana / na / nana / 1 3 0 0 2
  - 접미사 배열을 알고 있으면 O(N)에 구할 수 있음
    - 이건 뒤에서 알아보자.

Suffix Array

- LCP 배열
  - 각각의 접미사가 P로 시작하는지 판단하는 방법
    - X = 접미사 배열에서 정확히 P만 포함하는 접미사의 위치
    - L = R = X로 시작, LCP가 |P| 이상이면 L R 확장
      - 투 포인터와 비슷한 느낌
      - ex. bc / bc#bc / bcabcbc#bc / bcbc#bc
    - 패턴이 여러 개 주어졌을 때 더 빨리 처리할 수 있을까?

Suffix Array

- LCP 배열
  - 두 접미사의 최장 공통 접두사 = LCP 배열의 구간 최솟값
  - Segment Tree 사용하면 O(log N), Sparse Table 사용하면 O(1)에 구할 수 있음

  - 문자열 S와 패턴 P_1, P_2, … , P_k가 주어졌을 때 S에서 P_i가 등장하는 위치를 구하는 문제
    - S#P_1#P_2#…#P_k의 접미사 배열과 LCP 배열을 구하자.
    - 각각의 패턴 P_i에 대해, P_i를 접두사로 갖는 접미사의 구간을 구함
      - 접미사 P_i#P_{i+1}#…#P_k 의 위치 X를 구하고
      - 파라메트릭 서치와 구간 최솟값 쿼리를 이용해
      - X와 LCP가 $|P_i|$ 이상인 구간 [L, R]을 구함
    - L = |S| + sum |Pi|라고 하면
    - 전처리 O(L log L), 쿼리 O(log L)

Suffix Array

- 정리
  - 접미사 배열 모든 접미사를 사전 순으로 정렬한 자료구조
  - LCP 배열 접미사 배열에서 인접한 두 접미사의 최장 공통 접두사
  - 두 접미사의 LCP LCP에서 구간 최솟값 쿼리

  - 문자열 S와 Pi가 주어졌을 때 S에서 Pi가 등장하는 위치를 구하는 문제
    - $S\#P_1\#P_2\#...\#P_k$ 의 SA를 구하고
    - 접미사 $P_i\#P_{i+1}\#...\#P_k$ 의 위치에서 시작해서
    - $LCP \geq |P_i|$ 인 구간을 찾으면 됨

질문?

Algorithms

Algorithms

- $O(N^2 \log N)$
  - Quick Sort, Merge Sort, Heap Sort, …
  - 비교 O(N log N)번, 각 비교마다 O(N)
- $O(N^2)$
  - Radix Sort
  - 접미사들의 첫 번째 글자만 사용해서 정렬
    - banana: anana, ana, a / banana / nana, na
  - 첫 번째 글자가 같은 그룹 안에서 두 번째 글자로 정렬
    - a / anana, ana / banana / nana, na
  - 두 번째 글자까지 같은 그룹 안에서 세 번째 글자로 정렬
  - N번째 글자까지 반복

Algorithms

- $O(N \log^2 N)$
  - Manber-Myers Algorithm
  - Radix Sort를 이용한 $O(N^2)$ 알고리즘을 개선
  - 첫 k글자만 사용해 정렬한 SA를 이용해서
  - 2k글자를 사용해 정렬한 SA를 바로 만들 수 있음

  - S[i:i+2k]과 S[j:j+2k]를 비교하는 상황을 생각해 보자.
    - 이미 첫 k글자만 사용해 정렬되어 있는 상태
    - 먼저 S[i:i+k]와 S[j:j+k]를 비교
      - 만약 두 문자열이 다른 그룹에 속한다면 그 결과를 그대로 반환
    - S[i+k:i+2k]와 S[j+k:j+2k]를 비교
      - 두 문자열 모두 길이가 k이므로 k글자만 사용해 정렬되어 있는 배열을 그대로 사용

Algorithms

- $O(N \log^2 N)$
  - S = “abaabcab”

  - k = 1
    - a baabcab
    - a abcab
    - a bcab
    - a b
    - b aabcab
    - b cab
    - b
    - c ab

    Group A

    Group B

    Group C

  - k = 2
    - a a bcab
    - a b aabcab
    - a b cab
    - a b
    - b
    - b a abcab
    - b c ab
    - c a b

    Group (A, A)

    Group (A, B)

    Group (B, X)

    Group (B, A)

    Group (B, C)

    Group (C, A)

Algorithms

- $O(N \log^2 N)$
  - S = “abaabcab”
  - k = 2
    - aa bcab — Group 1
    - ab aabcab
    - ab cab — Group 2
    - ab
    - b — Group 3
    - ba abcab — Group 4
    - bc ab — Group 5
    - ca b — Group 6
- k = 4
  - aa bc ab — Group (1, 5)
  - ab — Group (2, X)
  - ab aa bcab — Group (2, 1)
  - ab ca b — Group (2, 6)
  - b — Group (3, X)
  - ba ab cab — Group (4, 2)
  - bc ab — Group (5, 2)
  - ca b — Group (6, 3)

Algorithms

- $O(N \log^2 N)$ 구현
  - sa[i] = 앞 k 글자만 이용해 정렬했을 때 i번째로 오는 접미사의 번호
  - pos[i] = sa 배열에서 i번 접미사의 위치
    - s[i..]의 등수, sa 배열의 역 함수
    - pos[sa[i]] = i

  - S[i:i+k] < S[j:j+k]는 pos[i] < pos[j]와 동일함
  - 각 k마다 std::sort를 이용해 접미사들을 O(N log N)에 정렬하면
  - $O(N \log^2 N)$에 접미사 배열을 구할 수 있음

Algorithms

- $O(N \log^2 N)$ 구현

```cpp
vector<int> SuffixArray(const string &s){
    int n = s.size();
    vector<int> sa(n), pos(n), tmp(n);
    for(int i=0; i<n; i++) sa[i] = i, pos[i] = s[i];
    for(int k=1; ; k<<=1){
        auto cmp = [&](int a, int b){
            if(pos[a] != pos[b]) return pos[a] < pos[b];
            if(a + k < n && b + k < n) return pos[a+k] < pos[b+k];
            return a > b; // short first
        };
        sort(sa.begin(), sa.end(), cmp);
        for(int i=1; i<n; i++) tmp[i] = tmp[i-1] + cmp(sa[i-1], sa[i]);
        for(int i=0; i<n; i++) pos[sa[i]] = tmp[i];
        if(tmp.back() + 1 == n) break;
    }
    return sa;
}
```

Algorithms

- O(N log N)
  - std::sort를 counting sort로 바꾸면 됨
  - k마다 정렬을 O(N)에 할 수 있으므로 O(N log N)

- O(N)
  - https://www.secmem.org/blog/2021/11/21/linear-suffix-array/
  - |S| ≤ 50만에서는 O(N log N)과 O(N) 거의 차이 없음

Algorithms

- O(N log N)

```cpp
vector<int> SuffixArray(const string &s){
    int n = s.size();
    vector<int> sa(n), pos(n), tmp(n), cnt(max(n, 256));
    auto counting_sort = [&](){
        fill(cnt.begin(), cnt.end(), 0);
        for(int i=0; i<n; i++) cnt[pos[i]]++;
        partial_sum(cnt.begin(), cnt.end(), cnt.begin());
        for(int i=n-1; i>=0; i--) sa[--cnt[pos[tmp[i]]]] = tmp[i];
    };
    for(int i=0; i<n; i++) sa[i] = tmp[i] = i, pos[i] = s[i];
    counting_sort();
    for(int k=1; ; k<<=1){
        int idx = 0;
        for(int i=n-k; i<n; i++) tmp[idx++] = i;
        for(int i=0; i<n; i++) if(sa[i] >= k) tmp[idx++] = sa[i] - k;
        counting_sort();
        tmp[sa[0]] = 0;
        for(int i=1; i<n; i++){
            tmp[sa[i]] = tmp[sa[i-1]];
            if(sa[i-1] + k < n && sa[i] + k < n && pos[sa[i-1]] == pos[sa[i]] && pos[sa[i-1]+k] ==
pos[sa[i]+k]) continue;
            tmp[sa[i]]++;
        }
        swap(pos, tmp);
        if(pos[sa.back()] + 1 == n) break;
    }
    return sa;
}
```

질문?

Algorithms

- O(N) LCP Array
  - Kasai’s Algorithm
  - Lemma 1. lcp(sa[i-1], sa[i]) $>$ 0이면 pos[sa[i-1]+1] $<$ pos[sa[i]+1]
    - S[sa[i-1]..]과 sa[sa[i]..]의 첫 글자가 일치하면
    - 첫 글자를 제거해도 두 접미사의 대소 관계가 유지됨
  - Lemma 2. lcp(sa[i-1], sa[i]) = k($>$ 0)이면 lcp(sa[i-1]+1, sa[i]+1) ≥ k-1
    - 첫 글자가 동일하기 때문에, 첫 글자를 지우면 lcp는 최대 1 감소

Algorithms

- O(N) LCP Array
  - Kasai's Algorithm
  - Lemma 1. lcp(sa[i-1], sa[i]) > 0이면 pos[sa[i-1]+1] < pos[sa[i]+1]
  - Lemma 2. lcp(sa[i-1], sa[i]) = k(> 0)이면 lcp(sa[i-1]+1, sa[i]+1) ≥ k-1

  - let p = pos[i-1], q = pos[i], j-1 = sa[p-1], k = sa[q-1]
    - k는 접미사 배열에서 s[i..]의 바로 앞에 오는 접미사
    - j-1은 접미사 배열에서 s[i-1..]의 바로 앞에 오는 접미사

  - Lemma 3. lcp(j-1, i-1) > 1이면 lcp(k, i) ≥ lcp(j, i)
    - Lemma 2에 의해 pos[j] < pos[i]
    - pos[j] ≤ pos[i]-1 = pos[k]이므로 접미사 배열 상에서 j보다 k가 i에 가까움
      - lcp(k, i) ≥ lcp(j, i)

Algorithms

- O(N) LCP Array
  - Kasai's Algorithm
  - Lemma 1. lcp(sa[i-1], sa[i]) > 0이면 pos[sa[i-1]+1] < pos[sa[i]+1]
  - Lemma 2. lcp(sa[i-1], sa[i]) = k(> 0)이면 lcp(sa[i-1]+1, sa[i]+1) ≥ k-1

  - let p = pos[i-1], q = pos[i], j-1 = sa[p-1], k = sa[q-1]
  - Lemma 3. lcp(j-1, i-1) > 1이면 lcp(k, i) ≥ lcp(j, i)

  - Theorem 1. lcp(j-1, i-1) > 1이면 lcp(k, i) ≥ lcp(j-1, i-1) – 1
    - Lemma 3에 의해 lcp(k, i) ≥ lcp(j, i)
    - Lemma 2에 의해 lcp(j, i) ≥ lcp(j-1, i-1) - 1

Algorithms

- O(N) LCP Array
  - let p = pos[i-1], q = pos[i], j-1 = sa[p-1], k = sa[q-1]
  - Theorem 1. lcp(j-1, i-1) > 1이면 lcp(k, i) ≥ lcp(j-1, i-1) – 1

  - 길이가 긴 접미사부터, 즉 s[i..]를 i가 증가하는 순서대로 보자.
    - 바로 직전에 lcp(j-1, i-1) = x를 계산했고, 이제 lcp(k, i)를 계산할 차례
    - lcp(k, i) ≥ x-1 이므로 x를 1 감소시킨 다음
    - s[sa[pos[i]-1]+x] == s[sa[pos[i]]+x] 이면 x 증가

    - x의 최댓값은 N이고, 각 반복문의 단계마다 x는 최대 1 감소하므로 전체 시간 복잡도는 O(N)

Algorithms

- O(N) LCP Array

```
for(int i=0, j=0; i<n; i++, j=max(j-1,0)){
    if(pos[i] == 0) continue;
    while(sa[pos[i]-1]+j < n && sa[pos[i]]+j < n && s[sa[pos[i]-1]+j] == s[sa[pos[i]]+j]) j++;
    lcp[pos[i]] = j;
}
```

질문?

Applications

Applications

- BOJ 3789 Hidden Password
  - 문자열 S가 주어지면, S의 cyclic shift 중 사전 순으로 가장 앞선 문자열을 구하는 문제

  - 원형 구조는 배열을 2번 이어 붙인 다음, 길이가 N인 부분 배열을 보면 편함
  - SS의 접미사 배열을 구한 다음 길이가 |S| 이상이면서 가장 앞에 오는 접미사를 구하면 됨

  - TMI
    - Lyndon factorization을 O(N)에 구하는 Duval algorithm을 응용하면 O(N)에 해결할 수 있음
    - https://github.com/justiceHui/icpc-teamnote/blob/master/code/String/Lyndon.cpp의 min_rotation 함수 참고

Applications

- BOJ 1605 반복 부분문자열
  - 문자열 S가 주어지면, 2번 이상 등장하는 S의 부분 문자열 중 가장 긴 것의 길이를 구하는 문제

  - lcp array의 정의에 의해 정답은 max(lcp) 이상인 것은 자명
  - Claim. 정답은 max(lcp) 이하다.
    - Pf) 만약 길이가 max(lcp) 초과인 두 번 이상 등장하는 부분 문자열 S[a..], S[b..]가 있다고 하자.
    - S[a..]와 S[b..]의 길이가 max(lcp)+1인 접두사가 동일하기 때문에
    - lcp array에는 max(lcp)+1 이상인 값이 있어야 한다.
    - lcp array의 최댓값이 max(lcp)라는 것에 모순이므로 정답은 max(lcp) 이하

  - max(lcp array)를 출력하면 된다.

질문?

Applications

- BOJ 11479 서로 다른 부분 문자열의 개수 2
  - 문자열 S가 주어지면, S의 모든 부분 문자열 중 서로 다른 부분 문자열의 개수를 구하는 문제

  - 각 문자열이 중복해서 등장하는 횟수(여사건)을 구해보자
    - 접미사 배열에서 인접한 두 접미사의 LCP가 X라는 것은
    - 두 접미사의 접두사 중 첫 X개가 중복된다는 것을 의미함

  - 여사건은 sum(lcp array)이므로 정답은 N(N+1)/2 – sum(lcp array)

Applications

- BOJ 9249 최장 공통 부분 문자열
  - 두 문자열 S, T가 주어졌을 때 S와 T의 최장 공통 부분 문자열의 길이를 구하는 문제

  - S#T의 접미사 배열을 구하자. (단, #은 S와 T의 문자보다 사전 순으로 앞선 문자)
  - 만약 sa[i]와 sa[i+1]이 서로 다른 문자열에서 시작하는 접미사라면
    - 즉, $(sa[i] < |S|) \neq (sa[i+1] < |S|)$
  - sa[i]와 sa[i+1]의 lcp는 정답의 후보

Applications

- BOJ 10413 반복되는 부분 문자열
  - 문자열 S가 주어지면, 두 번 이상 등장하는 서로 다른 부분 문자열의 개수를 구하는 문제

  - sum(lcp array)를 구하는 건 중복된 문자열을 여러 번 세기 때문에 안 됨
  - 어떻게 해야 할까?

Applications

- BOJ 10413 반복되는 부분 문자열
  - 문자열 S가 주어지면, 두 번 이상 등장하는 서로 다른 부분 문자열의 개수를 구하는 문제

  - sum(lcp array)를 구하는 건 중복된 문자열을 여러 번 세기 때문에 안 됨
  - 어떻게 해야 할까?

  - (sa[i-1], sa[i])의 lcp에 포함되는 문자열은 (sa[i], sa[i+1])에서 고려하면 안 됨
  - 따라서 정답은 max(0, lcp[i] – lcp[i-1])의 합

질문?

Applications

- BOJ 25546 가채점
  - 문자열 S가 주어지면, S에서 P_i가 K_i번째로 등장하는 위치를 구하는 쿼리를 처리하는 문제

  - 오프라인으로 처리하자.
  - S#P_1#P_2#…#P_Q의 접미사 배열과 LCP 배열을 구하면
  - P_i가 등장하는 접미사의 구간을 각 쿼리마다 O(log N)에 찾을 수 있음
  - 구간에서 K_i번째로 작은 수는 Persistent Segment Tree를 이용해 O(log N)에 구할 수 있음

Applications

- BOJ 9483 Tandem Repeats
  - 동일한 문자열이 2번 연속해서 등장하는 문자열을 tandem repeats라고 정의하자.
  - 문자열 S가 주어지면, S의 부분 문자열인 tandem repeats의 개수를 구하는 문제

  - 주기가 p인 tandem의 개수를 구하는 프로시저가 있으면
  - p = 1, 2, … , n/2에 대해 호출해서 해결할 수 있음

Applications

- BOJ 9483 Tandem Repeats
  - 문자열을 p글자씩 분할해서 B[i] = i번째 블록이라고 하자.
  - 주기가 p인 tandem repeats는 두 가지 형태 중 하나
    - B[i] = B[i+1]
      - LCP 배열에서 RMQ하면 블록마다 O(1)에 확인 가능
    - (B[i], B[i+1])의 최장 공통 접두사 + (B[i], B[i+1])의 최장 공통 접미사 ≥ p
      - S의 LCP 배열과 S를 뒤집은 문자열의 LCP 배열에서 각각 RMQ하면 블록마다 O(1)에 ghkre인 가능

  - 각 블록마다 O(1)이므로 주기가 p인 tandem repeats의 개수 O(n/p)에 구할 수 있음
  - sum n/p = O(n log n)

  - off by one error 주의

Applications

- BOJ 25505 공통 부분 문자열 쿼리
  - 문자열 $S_1, S_2, \dots, S_N$이 주어지면
  - 두 문자열 S_i, S_j의 서로 다른 공통 부분 문자열의 개수를 구하는 쿼리를 처리하는 문제

  - 오늘 배운 모든 테크닉을 사용하는 문제
  - https://github.com/justiceHui/Sunrin-Contest/blob/main/Sunrin-OI-2022/editorial.pdf
  - 17페이지 이후 참고

질문?