String Processing

Workshop in Competitive Programming – 234900

Data Structures

STL, Trie, Suffix tree and Suffix array

STL String

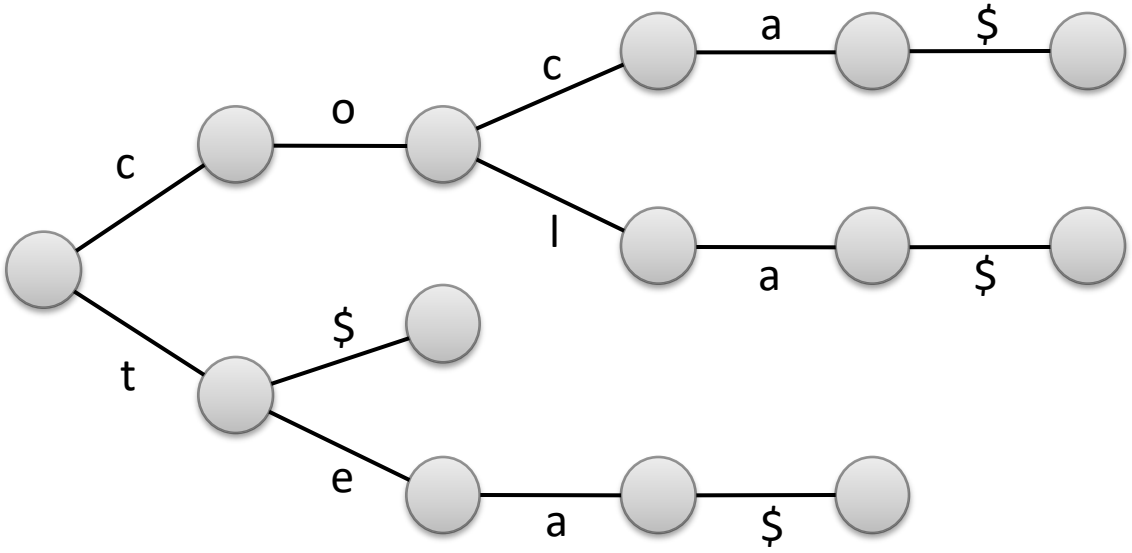
C++ String

- The standard library implements a *String* class.
- Mostly two options for input:
  1. cin >> s (reads until whitespace)
  2. getline(cin, input) (reads whole line)
- Supports all the common operations:
  - *begin(), end(), size(), push_back(), pop_back() etc…*
- Supports *find()* operation
  - Worst case complexity: $O(n \cdot m)$
- set<string> is a simple but powerful container.

Trie

Trie

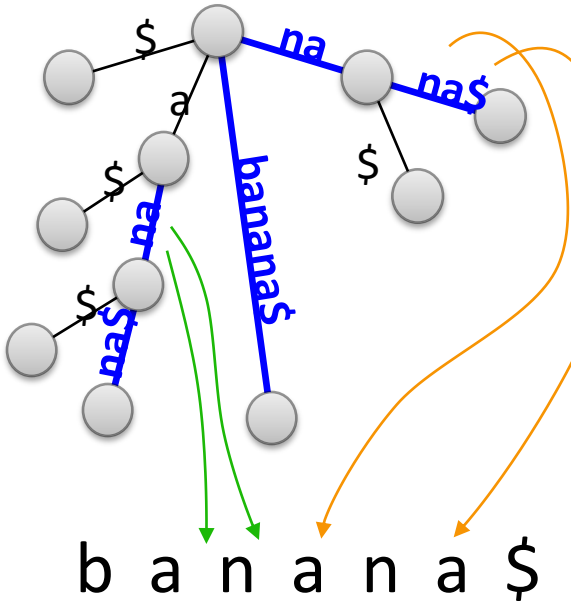
- String dictionary
  - $insert(s), find(s), delete(s)$ in $O(n)$
- Complicated implementation (but sometimes needed)



Suffix Tree and Array

Suffix Tree

- A (compresed) trie, containing all the suffixes of a string
  - Can be built in $O(n)$
- Complicated!
- Applications:
  - String matching ($O(m)$)
  - Longest repeated substring ($O(n)$)
  - Longest common substring ($O(n)$)



Suffix Array

- A sorted array containing all the suffixes
  - Actually, just the starting position
- Naïve implementation: $O(n^2 \log n)$
- Better implementation: $O(n \log n)$
  - Using smart radix sort
  - Code in webcourse
- Pros: (Relatively) simple to implement
- Cons: Sometimes a factor of $\log n$ in the complexity

i	A[i]	Suffix
0	12	$
1	11	i$
2	8	ippi$
3	5	issippi$
4	2	ississippi$
5	1	mississippi$
6	10	pi$
7	9	ppi$
8	7	sippi$
9	4	sissippi$
10	6	ssippi$
11	3	ssissippi$

Suffix Array

- Applications:
  - String Matching ($O(m \log n)$)
  - Longest Common Prefix
    - Naïve: $O(n^2)$
    - Better: $O(n)$
  - Longest Repeated Substring ($O(n)$)
    - Naïve: $O(n^2)$
    - Better: $O(n)$ (using LCP)
  - Longest Common Substring
    - $O(n)$ – similar idea to LCS in suffix tree
  - Further reading: Wikipedia
    GeeksForGeeks

i	A[i]	Suffix
0	12	$
1	11	i$
2	8	ippi$
3	5	issippi$
4	2	ississippi$
5	1	mississippi$
6	10	pi$
7	9	ppi$
8	7	sippi$
9	4	sissippi$
10	6	ssippi$
11	3	ssissippi$

KMP Algorithm

String Matching

- Problem:
- Given String s and Pattern p find p in s.
- Example: $s = aaaaaaaaab\$, p = aaab\$$
- Solutions:
  - Naïve approach: complete search, $O(nm)$.
  - Suffix array – $O(n \log n)$
  - Suffix tree – $O(n)$ efficient but complicated
- Goal: Simple algorithm with $O(n)$ complexity

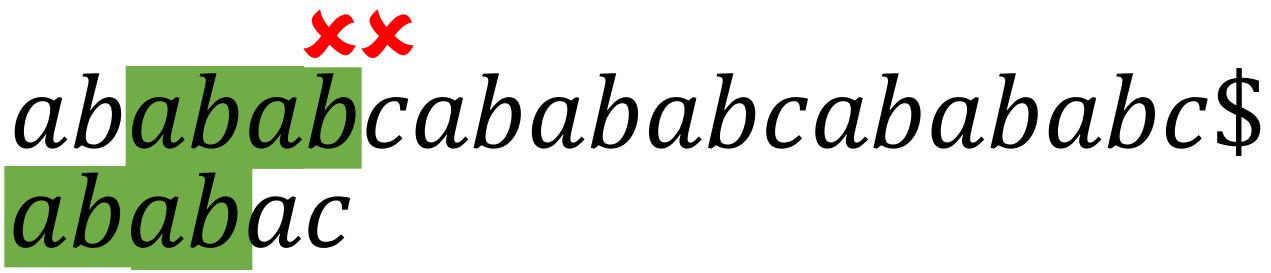
Knuth-Morris-Pratt's (KMP) Alg.

- Consider the naïve approach:

- What can be improved?
- Comparing the "aaa" is redundant!

Knuth-Morris-Pratt's (KMP) Alg.

- Slightly more complicated



- What can be improved here?
- And here?
- We would like to know where to restart in case of failure

Knuth-Morris-Pratt's (KMP) Alg.

- Lets generalize

$$\ldots \mathbf{a}\mathbf{b}\mathbf{a}\mathbf{b}\mathbf{a}\mathbf{x} \ldots$$
$$a\,b\,a\,b\,a\,c$$

Restart points: -1,0,0,1,2,3

- Define: $lps[i]$ = the longest **proper** prefix of $p[0..i]$ which is also a suffix of $p[0..i]$
  - $lps[0] = -1$
- Can be computed in $O(n)$

Knuth-Morris-Pratt's (KMP) Alg.

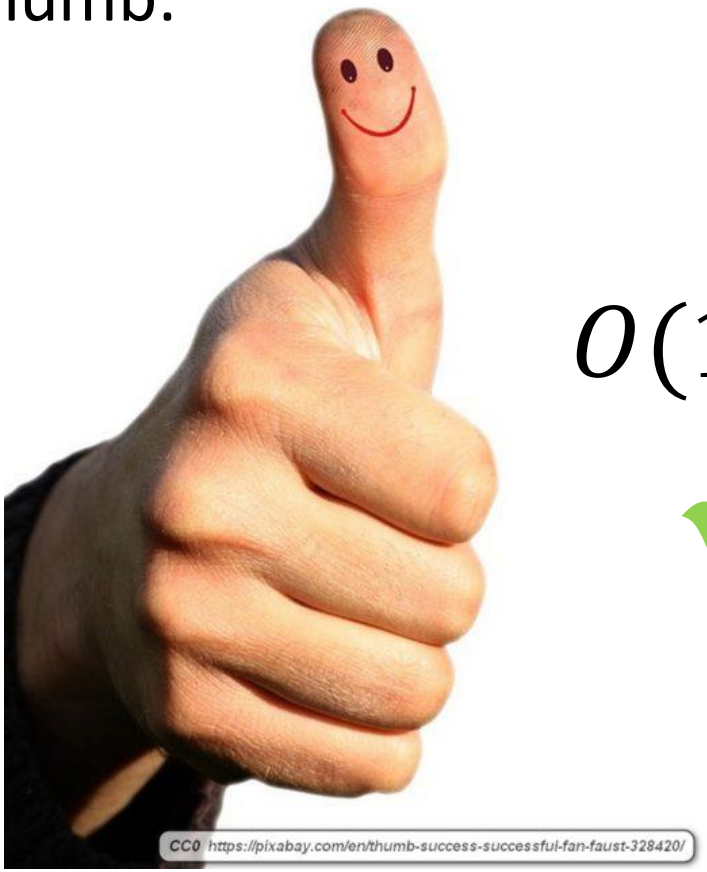
```
string s; // The string to search in
string pat; // The pattern to search
vi lps;

// KMP Init
void KMP_init(){
    int m = pat.length();
    lps.resize(m,0);
    lps[0]=-1;
    int i = 0, j = -1;
    while (i < m) {
        while (j >= 0 && pat[i] != pat[j])
                j = lps[j];
        i++; j++;
        lps[i] = j;
    }
}
```

```
void KMP_search() {
    int n = s.length();
    int m = pat.length();
    int i = 0, j = 0;
    while (i < n) {
        while (j >= 0 && s[i] != pat[j])
                j = lps[j];
        i++; j++;
        if (j == m) { // Pattern found
            cout << "The pattern is
                    found at index " <<
                    i-j << endl;
            j = lps[j];
        }
    }
}
```

Pay attention to input size

- Sometimes the naïve solution is good enough
- Rule of thumb:



$O(10^7)$