# Introduction to Algorithms

## Science Honors Program (SHP)

## Session 9

**Christian Lim**

Saturday, April 27, 2024

# Overview

- String Hashing
- Rabin-Karp
- Knuth-Morris-Pratt (KMP)
- Z-function
- Suffix Array

# String Hashing: Main Idea

$$\begin{aligned}\text{hash}(s) &= s[0] + s[1] \cdot p + s[2] \cdot p^2 + \ldots + s[n-1] \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,\end{aligned}$$

# String Hashing: Implementation

```
long long compute_hash(string const& s) {
    const int p = 31;
    const int m = 1e9 + 9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c-'a'+1)*p_pow)%m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
```

# Example Problem

- Given a list of $n$ strings $s_i$, each no longer than $m$ characters, find all the duplicate strings and divide them into groups.

# Solution

```
vector<vector<int>>
group_identical_strings(vector<string> const& s) {
    int n = s.size();
    vector<pair<long long, int>> hashes(n);
    for (int i = 0; i < n; i++)
        hashes[i] = {compute_hash(s[i]), i};
    sort(hashes.begin(), hashes.end());
```

# Solution

```
    vector<vector<int>> groups;
    for (int i = 0; i < n; i++) {
        if (i == 0 ||
            hashes[i].first != hashes[i-1].first)
            groups.emplace_back();
        groups.back().push_back(hashes[i].second);
    }
    return groups;
}
```

# Fast hash calculation of substrings of given string

- Given a string `s` and indices `i` and `j`, find the hash of the substring `s[i...j]`.

# Solution

$$\text{hash}(s[i \dots j]) = \sum_{k=i}^{j} s[k] \cdot p^{k-i} \mod m$$

$$\begin{aligned} \text{hash}(s[i \dots j]) \cdot p^i &= \sum_{k=i}^{j} s[k] \cdot p^k \mod m \\ &= \text{hash}(s[0 \dots j]) - \text{hash}(s[0 \dots i-1]) \mod m \end{aligned}$$

# Applications

- **Rabin-Karp** algorithm for pattern matching in a string in $O(n)$ time.
- Calculating the *number of different substrings* of a string in $O(n^2 \log n)$
- Calculating the *number of palindromic substrings* in a string.

# Determine the number of different substrings in a string

- Given a string s of length n, consisting only of lowercase English letters, find the *number of different substrings* in this string.

# Solution

```
int count_unique_substrings(string const& s) {
    int n = s.size();
    const int p = 31;
    const int m = 1e9 + 9;
    vector<long long> p_pow(n);
    p_pow[0] = 1;
    for (int i = 1; i < n; i++)
        p_pow[i] = (p_pow[i-1] * p) % m;
    vector<long long> h(n + 1, 0);
    for (int i = 0; i < n; i++)
        h[i+1] = (h[i] + (s[i]-'a'+1) * p_pow[i]) % m;
```

# Solution (con't)

```
    int cnt = 0;
    for (int l = 1; l <= n; l++) {
        set<long long> hs;
        for (int i = 0; i <= n - l; i++) {
            long long cur_h = (h[i + l]+m-h[i]) % m;
            cur_h = (cur_h * p_pow[n-i-1]) % m;
            hs.insert(cur_h);
        }
        cnt += hs.size();
    }
    return cnt;
}
```

# Rabin-Karp (1987): Problem

- Given two strings - a pattern `s` and a text `t`, determine if the pattern appears in the text and if it does, enumerate all its occurrences in $O(|s| + |t|)$ time.

# Rabin-Karp (1987): Main Idea

- Calculate the hash for the pattern `s`.
- Calculate hash values for all the prefixes of the text `t`.
- Now, we can compare a substring of length `|s|` with `s` in constant time using the calculated hashes.
- So, compare each substring of length `|s|` with the pattern.
- This will take a total of `O(|t|)` time.
- Hence the final complexity of the algorithm is `O(|t|+|s|)`
  - `O(|s|)` is required for calculating the hash of the pattern and;
  - `O(|t|)` for comparing each substring of length `|s|` with the pattern.

# Rabin-Karp: Implementation

```
vector<int> rabin_karp(string const& s,
                       string const& t) {
    const int p = 31;
    const int m = 1e9 + 9;
    int S = s.size(), T = t.size();
    vector<long long> p_pow(max(S, T));
    p_pow[0] = 1;
    for (int i = 1; i < (int)p_pow.size(); i++)
        p_pow[i] = (p_pow[i-1] * p) % m;
```

# Rabin-Karp: Implementation

```
vector<long long> h(T + 1, 0);
for (int i = 0; i < T; i++)
    h[i+1] = (h[i] + (t[i]-'a'+1) * p_pow[i]) % m;
long long h_s = 0;
for (int i = 0; i < S; i++)
    h_s = (h_s + (s[i]-'a'+1) * p_pow[i]) % m;
```

# Rabin-Karp: Implementation

```
    vector<int> occurences;
    for (int i = 0; i + S - 1 < T; i++) {
        long long cur_h = (h[i+S] + m - h[i]) % m;
        if (cur_h == h_s * p_pow[i] % m)
            occurences.push_back(i);
    }
    return occurences;
}
```

# Knuth-Morris-Pratt (KMP): Prefix function

- You are given a string s of length n.
- The **prefix function** for this string is defined as an array **π** of length n, where **π**[i] is the length of the longest proper prefix of the substring s[0...i] which is also a suffix of this substring.
- A proper prefix of a string is a prefix that is not equal to the string itself. By definition, **π**[0]=0.

$$\pi[i] = \max_{k=0\ldots i} \{k : s[0\ldots k-1] = s[i-(k-1)\ldots i]\}$$

# Knuth-Morris-Pratt (KMP): Prefix function Example

- prefix function of string `"abcabcd"` is `[0,0,0,1,2,3,0];`
- prefix function of string `"aabaaab"` is `[0,1,0,1,2,2,3];`

# Knuth-Morris-Pratt (KMP): Main Idea

- We compute the prefix values $\pi[i]$ in a loop by iterating from $i=1$ to $i=n-1$ ($\pi[0]$ just gets assigned with $0$).
- To calculate the current value $\pi[i]$ we set the variable $j$ denoting the length of the best suffix for $i-1$. Initially $j = \pi[i-1]$.
- Test if the suffix of length $j+1$ is also a prefix by comparing $s[j]$ and $s[i]$. If they are equal then we assign $\pi[i]= j+1$, otherwise we reduce $j$ to $\pi[j-1]$ and repeat this step.
- If we have reached the length $j = 0$ and still don't have a match, then we assign $\pi[i] = 0$ and go to the next index $i+1$.

# Knuth-Morris-Pratt (KMP): Implementation

```
vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j]) j++;
        pi[i] = j;
    }
    return pi;
}
```

# Example Problem

- Given a text `t` and a string `s`, we want to find and display the positions of all occurrences of the string s in the text `t`.

# Solution

- We generate the string `s+"#"+t`, where `"#"` is a separator that appears neither in `s` nor in `t`.
- If at some position `i` we have `π[i]=n`, then at the position `i-(n+1)-n+1=i-2n` in the string `t` the string `s` appears.

# Z-function: Definition

- Suppose we are given a string `s` of length `n`. The **Z-function** for this string is an array of length `n` where the `i`-th element is equal to the greatest number of characters starting from the position `i` that coincide with the first characters of `s`.

# Z-function: Example

- `"aaaaa" - [0,4,3,2,1]`
- `"aaabaab" - [0,2,1,0,2,1,0]`
- `"abacaba" - [0,0,1,0,3,0,1]`

# Z-function: Implementation

```
vector<int> z_function(string s) {
    int n = (int) s.length();
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r) z[i] = min (r - i + 1, z[i - l]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
    }
    return z;
}
```

# Example Problem

- Find all occurrences of the pattern `p` inside the text `t`.

# Solution

- To solve this problem, we create a new string `s=p+"$"+t`, that is, we apply string concatenation to `p` and `t` but we also put a separator character `"$"` in the middle.
- Compute the Z-function for `s`. Then, for any `i` in the interval `[0, len(t)-1]`, we will consider the corresponding value `k=z[i+len(p)+1]`.
- If `k` is equal to `len(p)` then we know there is one occurrence of `p` in the `i`-th position of `t`, otherwise there is no occurrence of `p` in the `i`-th position of `t`.

# Example Problem

- Find all occurrences of the pattern `p` inside the text `t`.

# Solution

- To solve this problem, we create a new string `s=p+"$"+t`, that is, we apply string concatenation to `p` and `t` but we also put a separator character `"$"` in the middle.
- Compute the Z-function for `s`. Then, for any `i` in the interval `[0, len(t)-1]`, we will consider the corresponding value `k=z[i+len(p)+1]`.
- If `k` is equal to `len(p)` then we know there is one occurrence of `p` in the `i`-th position of `t`, otherwise there is no occurrence of `p` in the `i`-th position of `t`.

# Suffix Array: Definition

- Let s be a string of length n.
- The $i^{th}$ suffix of s is the substring s[i...n-1].

- A suffix array will contain integers that represent the starting indexes of the all the suffixes of a given string, after the aforementioned suffixes are sorted.

- There are $O(n^2 \log n)$, $O(n \log n)$, and $O(n)$ solutions!

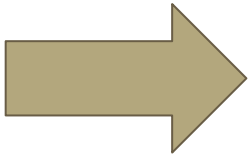
# Suffix Array: Example

- s = "abaab"

# Suffix Array: Example

- s = "abaab"

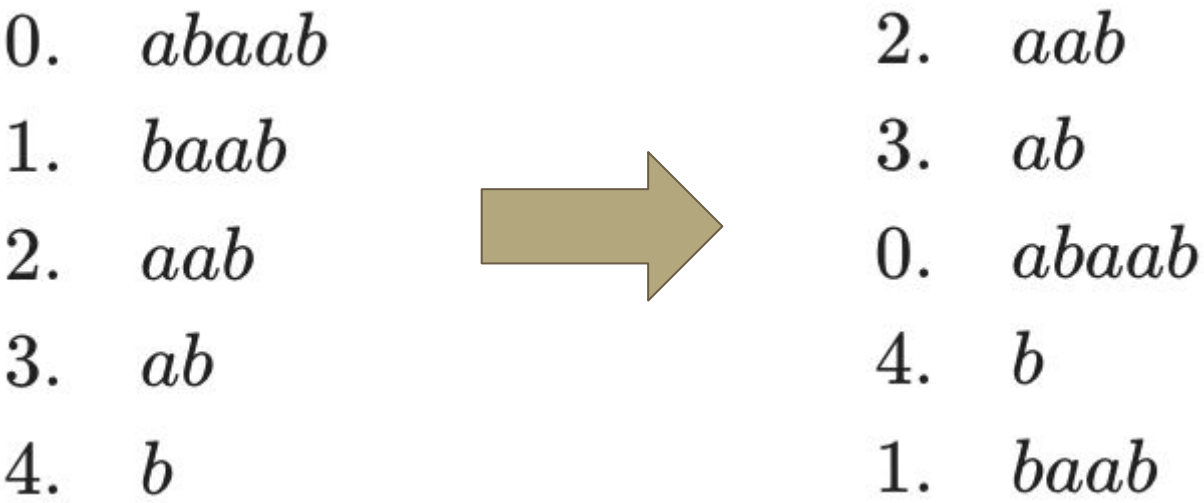
| | |
|---|---|
| 0. | $abaab$ |
| 1. | $baab$ |
| 2. | $aab$ |
| 3. | $ab$ |
| 4. | $b$ |



| | |
|---|---|
| 2. | $aab$ |
| 3. | $ab$ |
| 0. | $abaab$ |
| 4. | $b$ |
| 1. | $baab$ |

# Suffix Array: Example

- s = "abaab"




- the suffix array of s will be **[2, 3, 0, 4, 1]**.

# Naive Approach ($O(n^2 \log n)$)

- Get all the suffixes and sort them using quicksort or mergesort and simultaneously retain their original indices.
- Sorting uses O(n log n) comparisons
- Since comparing two strings will additionally take O(n) time, we get the final complexity of $O(n^2 \log n)$.

- **Too slow in programming contests!**

# "Good Enough" Approach (O(n log n))

- Sort the cyclic shifts of a string.
- We can very easily derive an algorithm for sorting suffixes from it: append '$'
- Example: 'dabbb'

| | | |
|---|---|---|
| 1. | $abbb\$d$ | $abbb$ |
| 4. | $b\$dabb$ | $b$ |
| 3. | $bb\$dab$ | $bb$ |
| 2. | $bbb\$da$ | $bbb$ |
| 0. | $dabbb\$$ | $dabbb$ |

# "Good Enough" Approach (O(n log n))

- The algorithm we discuss will perform log n + 1 iterations.
- In the $k^{th}$ iteration, we sort the n cyclic substrings of s of length $2^k$.
- In each iteration of the algorithm, in addition to the permutation p where p[i] is the index of the $i^{th}$ substring (starting at i and with length $2^k$) in the sorted order, we will also maintain an array c, where c[i] corresponds to the equivalence class to which the substring belongs.

# "Good Enough" Approach (O(n log n))

- Example: s = "aaba"
- In 0th iteration, array **p** can also be [3, 1, 0, 2] or [3, 0, 1, 2]
- But, array **c** is fixed!

$$
\begin{array}{lcll}
0: & (a,\ a,\ b,\ a) & p = (0,\ 1,\ 3,\ 2) & c = (0,\ 0,\ 1,\ 0) \\
1: & (aa,\ ab,\ ba,\ aa) & p = (0,\ 3,\ 1,\ 2) & c = (0,\ 1,\ 2,\ 0) \\
2: & (aaba,\ abaa,\ baaa,\ aaab) & p = (3,\ 0,\ 1,\ 2) & c = (1,\ 2,\ 3,\ 0)
\end{array}
$$

# Longest Common Prefix (LCP)

- For a given string s, we want to compute the longest common prefix (**LCP**) of two arbitrary suffixes with position i and j.

- We can use **suffix array** and **Kasai's algorithm** to compute this in **O(n)**.

# Example Problem

- Finding a substring in a string.

# Example Problem

- Finding a substring in a string.
- Comparing two substrings of a string.

# Example Problem

- Finding a substring in a string.
- Comparing two substrings of a string.
- Number of different substrings.

# Columbia University Local Contest (CULC)

- We will have 3rd **Columbia University Local Contest** (CULC)
  - Individual, not team, contest!
  - Date: **Saturday, April 27, 2024 (TODAY!)**
  - Time: **2pm ET ~ 7pm ET**
  - **Online**
- **https://bit.ly/spring2024-culc-flyer**
- Please sign up using: **https://bit.ly/icpc-culc-registration**
- Direct Invitation Link: **https://codeforces.com/contestInvitation/d1c8453f62d3fce50c0baa5eb223d991e80b1e34**

# Summer 2024 Coaching

- **Christian Lim** can help you getting better in programming contests throughout the summer and beyond via a small team-based coaching.

- https://bit.ly/christian-lim-coaching


THANK YOU

# Now, let's cover:

- Aho-Corasick
- Suffix Tree
- Suffix Automaton
- Lyndon factorization
- Manacher's

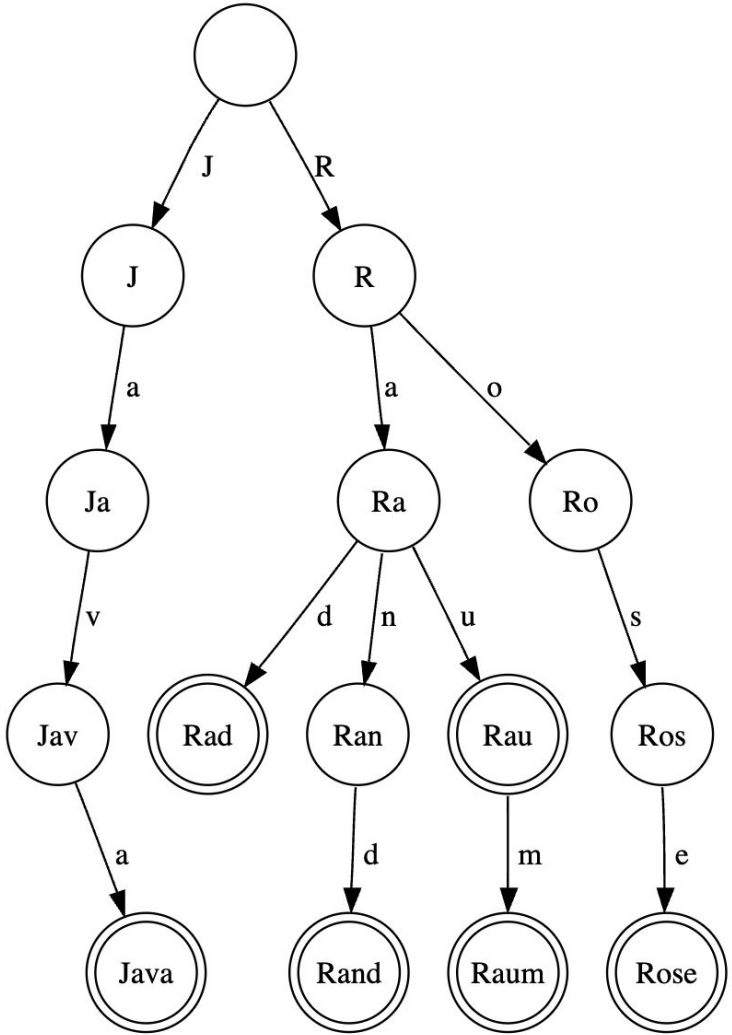
# Aho-Corasick (1975)

- quickly search for multiple patterns in a text.
- The set of pattern strings: **dictionary**.
- The algorithm constructs a **finite state automaton** based on a trie in linear time and then uses it to process the text.

# Trie

- A **trie** based on words "Java", "Rad", "Rand", "Rau", "Raum" and "Rose".

- A trie is a **rooted tree** where each edge of the tree is labeled with some letter and outgoing edges of a vertex have distinct labels.



# Trie: constructor

```
const int K = 26;
struct Vertex {
    int next[K];
    bool output = false;
    Vertex() {
        fill(begin(next), end(next), -1);
    }
};
vector<Vertex> trie(1);
```

# Trie: add_string()

```
void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (trie[v].next[c] == -1) {
            trie[v].next[c] = trie.size();
            trie.emplace_back();
        }
        v = trie[v].next[c];
    }
    trie[v].output = true;
}
```

# Automaton

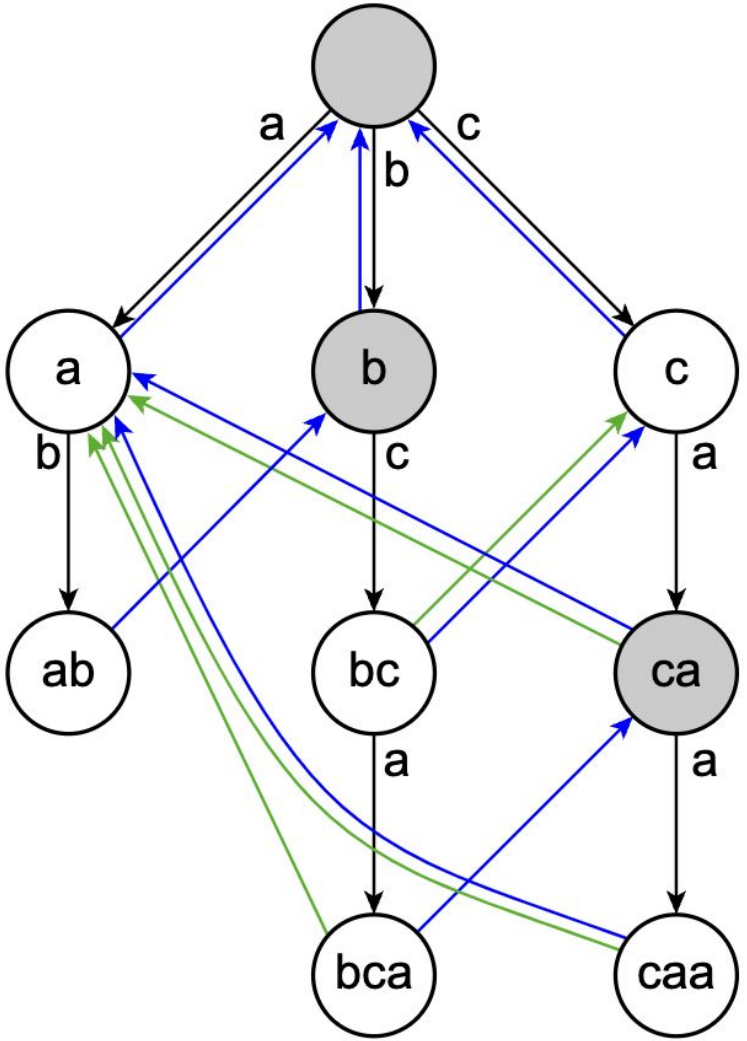
- The trie vertices can be interpreted as states in a **finite deterministic automaton**.
- From any state we can transition - using some input letter - to other states, i.e., to another position in the set of strings.

- For example, if there is only one string abc in the dictionary, and we are standing at vertex ab, then using the letter c we can go to the vertex abc.

# Aho-Corasick Automaton

- An Aho-Corasick automaton based on words "a", "ab", "bc", "bca", "c" and "caa".

- **blue** arrows are suffix links, **green** arrows are terminal links.



# Aho-Corasick Automaton Implementation: constructor

```
const int K = 26;
struct Vertex {
  int next[K], go[K];
  bool output = false;
  int p = -1, link = -1
  char pch;
  Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
    fill(begin(next), end(next), -1);
    fill(begin(go), end(go), -1);
  }
};
```

# Aho-Corasick Automaton Implementation: add_string()

```
vector<Vertex> t(1);
void add_string(string const& s) {
  int v = 0;
  for (char ch : s) {
    int c = ch - 'a';
    if (t[v].next[c] == -1)
      t[v].next[c] = t.size(), t.emplace_back(v, ch);
    v = t[v].next[c];
  }
  t[v].output = true;
}
```

# Aho-Corasick Automaton Implementation: get_link()

```
int go(int v, char ch);

int get_link(int v) {
  if (t[v].link == -1) {
    if (v == 0 || t[v].p == 0)
      t[v].link = 0;
    else
      t[v].link = go(get_link(t[v].p), t[v].pch);
  }
  return t[v].link;
}
```

# Aho-Corasick Automaton Implementation: go()

```
int go(int v, char ch) {
  int c = ch - 'a';
  if (t[v].go[c] == -1) {
    if (t[v].next[c] != -1)
      t[v].go[c] = t[v].next[c];
    else
      t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
  }
  return t[v].go[c];
}
```

# Aho-Corasick Automaton: Implementation

- There is also BFS-based implementation too!

# Aho-Corasick: Applications

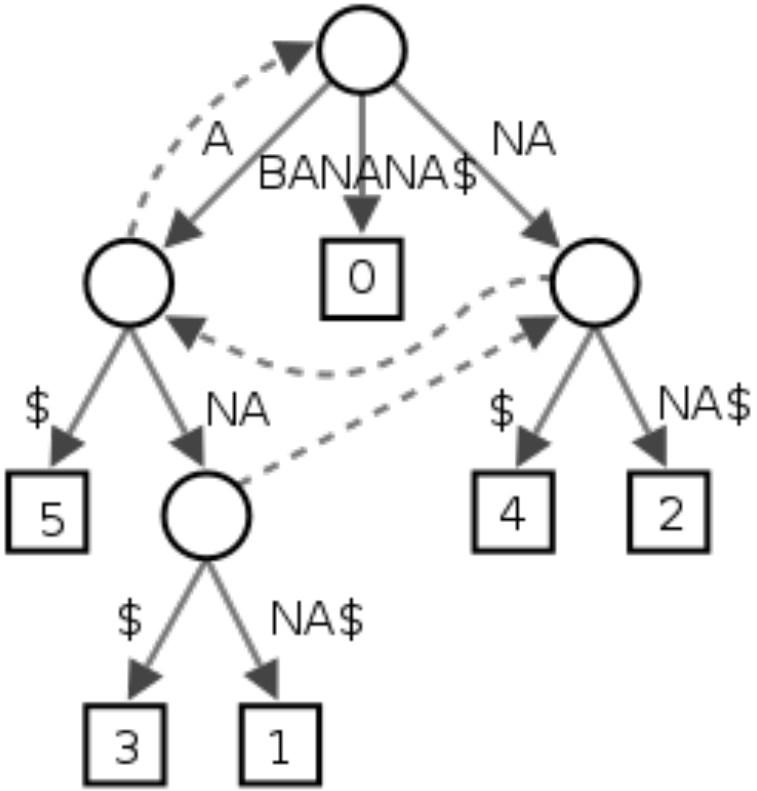
- Find all strings from a given set in a text;
- Finding the lexicographically smallest string of a given length that doesn't match any given strings;
- Finding the shortest string containing all given strings;
- Finding the lexicographically smallest string of length L containing k strings

# Suffix Tree: Definition

- A suffix tree is a compressed trie containing **all the suffixes** of the given text as their *keys* and **positions** in the text as their *values*.

- Suffix trees allow particularly fast implementations of many important string operations.

# Suffix Tree: Illustration




- Suffix tree for the text BANANA.
- Each substring is terminated with special character $.
- The six paths from the root to the leaves (shown as boxes) correspond to the six suffixes A$, NA$, ANA$, NANA$, ANANA$ and BANANA$.
- The numbers in the leaves give the start position of the corresponding suffix.
- Suffix links, drawn dashed, are used during construction.

# Suffix Tree: Ukkonen's Algorithm

- A suffix tree for a given string s of length n can be built in O(n log k) time where k is the size of the alphabet. So, if k is considered to be a constant, the asymptotic behavior is linear.
- The input to the algorithm are the string s and its length n.

- The main function build_tree builds a suffix tree.
- It is stored as an array of structures node, where node[0] is the root of the tree.

# Ukkonen's Algorithm: Implementation Details

- In order to simplify the code, the edges are stored in the same structures: for each vertex its structure node stores the information about the edge between it and its parent.
- Overall each node stores the following information:
  - `(l,r)`: left and right boundaries of the substring `s[l..r-1]` which correspond to the edge to this node;
  - `par`: the parent node;
  - `link`: the suffix link;
  - `next`: the list of edges going out from this node;

# Suffix Tree: Ukkonen's Algorithm

- A full implementation of Ukkonen's Algorithm can be found here.

# Suffix Automaton

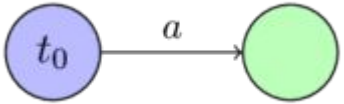
- A minimal **deterministic finite automaton** that accepts all the suffixes of the string.

# Suffix Automaton: Example


""

# Suffix Automaton: Example

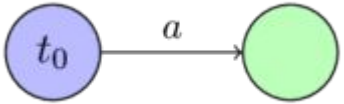

""



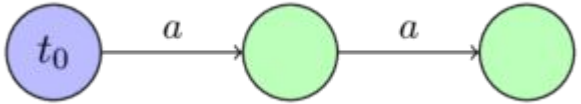

"a"

# Suffix Automaton: Example


""



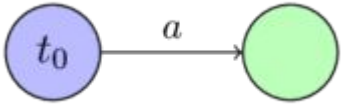

"a"



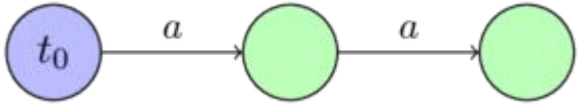

"aa"

# Suffix Automaton: Example

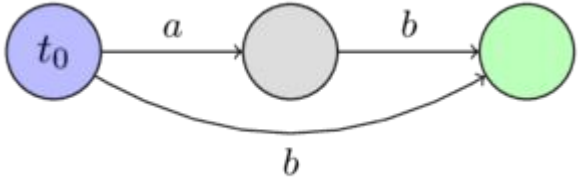

""




"a"

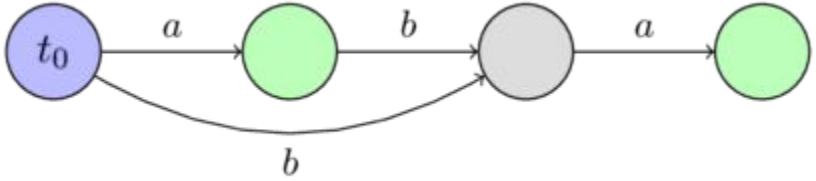



"aa"



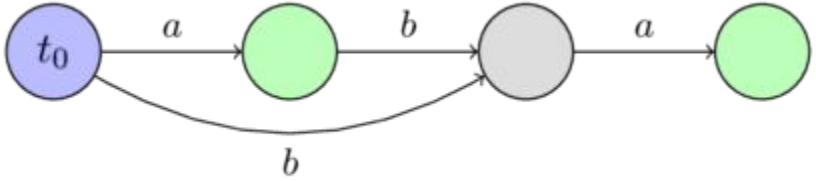

"ab"

# Suffix Automaton: Example

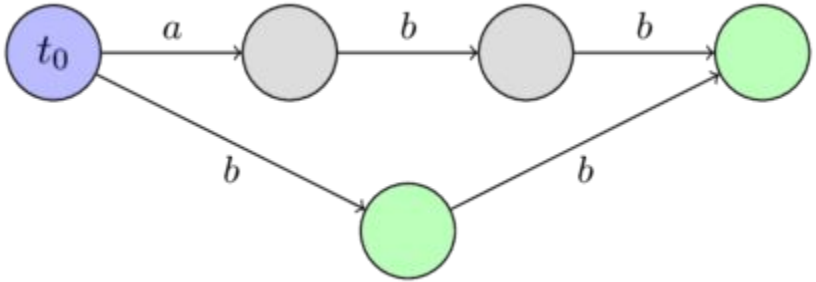



"aba"

# Suffix Automaton: Example

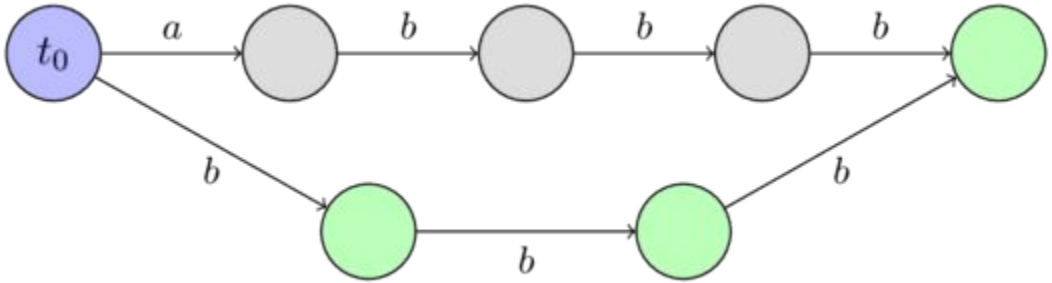



"aba"




"abb"

# Suffix Automaton: Example




"abbb"

# Suffix Automaton: Implementation

```
struct state {
  int len, link;
  map<char, int> next;
};

const int MAXLEN = 100000;
state st[MAXLEN * 2];
int sz, last;
```

# Suffix Automaton: Implementation

```
void sa_init() {
  st[0].len = 0;
  st[0].link = -1;
  sz++;
  last = 0;
}
```

# Suffix Automaton: Implementation

```
void sa_extend(char c) {
  int cur = sz++;
  st[cur].len = st[last].len + 1;
  int p = last;
  while (p != -1 && !st[p].next.count(c)) {
    st[p].next[c] = cur;
    p = st[p].link;
  }
```

# Suffix Automaton: Implementation

```
if (p == -1) {
  st[cur].link = 0;
} else {
  int q = st[p].next[c];
  if (st[p].len + 1 == st[q].len) {
    st[cur].link = q;
  }
```

# Suffix Automaton: Implementation

```
    else {
      int clone = sz++;
      st[clone].len = st[p].len + 1;
      st[clone].next = st[q].next;
      st[clone].link = st[q].link;
      while (p != -1 && st[p].next[c] == q)
        st[p].next[c] = clone, p = st[p].link;
      st[q].link = st[cur].link = clone;
    }
  }
  last = cur;
}
```

# Suffix Automaton: Applications

- Check for occurrence
- Number of different substrings
- Total length of all different substrings
- Lexicographically $k^{th}$ substring
- Smallest cyclic shift
- Number of occurrences
- First occurrence position
- All occurrence positions
- Shortest non-appearing string
- Longest common substring of two or more strings

# Lyndon factorization

- A string is called **simple** (or a **Lyndon word**) if it is strictly smaller than any of its own nontrivial suffixes.
- Examples of simple strings are: a, b, ab, aab, abb, ababb, abcd.
- It can be shown that a string is simple if and only if it is strictly smaller than all its nontrivial cyclic shifts.
- The **Lyndon factorization** of the string s is a factorization $s = w_1 w_2 ... w_k$ where all strings $w_i$ are simple and they are in non-increasing order $w_1 \geq w_2 \geq ... \geq w_k$.
- It can be shown such a factorization **exists** and it is **unique**.

# Lyndon factorization: Duval

- The **Duval** algorithm constructs the Lyndon factorization in O(n) time and O(1) additional memory. Similar to **Booth** algorithm (Michael's presentation last Wednesday).

- A string t is called **pre-simple** if it has the form t = ww...wp, where w is a simple string and p is a prefix of w (possibly empty). A simple string is also pre-simple.

# Lyndon factorization: Duval

- **Greedy**
- At any point during its execution, the string s will actually be divided into three strings $s = s_1 s_2 s_3$ where:
  - the Lyndon factorization for $s_1$ is already found and finalized;
  - the string $s_2$ is pre-simple (and we know the length of the simple string in it);
  - $s_3$ is completely untouched;
- In each iteration, look at the first character of the string $s_3$ and tries to append it to the string $s_2$. If $s_2$ is no longer pre-simple then the Lyndon factorization for some part of $s_2$ becomes known. This part goes to $s_1$.

# Lyndon factorization: Duval: Implementation Details

- The pointer i will always point to the beginning of the string $s_2$.
- The outer loop will be executed as long as i<n.
- Inside the loop we use two additional pointers:
  - j which points to the beginning of $s_3$;
  - k which points to the current character that we are currently comparing to;
- We want to add the character s[j] to the string $s_2$, which requires a comparison with the character s[k].

# Lyndon factorization: Duval: Implementation Details

- There can be three different cases:
  - **s[j]=s[k]**: if this is the case then adding the symbol s[j] to $s_2$ does not violate its pre-simplicity. So we simply increment the pointers j and k.
  - **s[j]>s[k]**: here, the string $s_2$+s[j] becomes simple. We can increment j and reset k back to the beginning of $s_2$, so that the next character can be compared with the beginning of the simple word.
  - **s[j]<s[k]**: the string $s_2$+s[j] is no longer pre-simple. We will split the pre-simple string $s_2$ into its simple strings and the remainder. The simple string will have the length j-k. In the next iteration we start again with the remaining $s_2$.

# Lyndon factorization: Duval: Implementation

```
vector<string> duval(string const& s) {
  int n=s.size(), i=0;
  vector<string> ret;
  while (i<n) {
    int j=i+1, k=i;
    while (j<n && s[k]<=s[j]) {
      if (s[k]<s[j]) k=i; else k++;
      j++;
    }
    while (i <= k) ret.push_back(s.substr(i,j-k)), i+=j-k;
  }
  return ret;
}
```

# Application: finding the smallest cyclic shift

- Given string s, construct the Lyndon factorization for the string s+s in O(n) time.
- Look for a simple string in the factorization which starts at a position less than n (i.e. it starts in the first instance of s) and ends in a position greater than or equal to n (i.e. in the second instance of s).
- The position of the start of this simple string will be the beginning of the desired **smallest cyclic shift**.

# Manacher's Algorithm

- Given: string s with length n.
- Find all the pairs (i, j) such that substring s[i...j] is a palindrome.

- We describe the algorithm to find all the sub-palindromes with **odd** length.

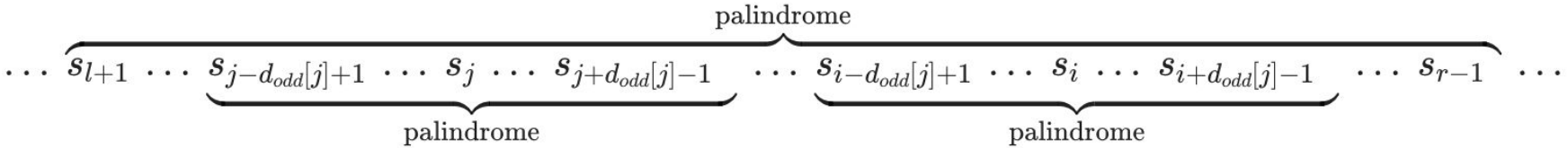
# Manacher's Algorithm: Implementation Details

- Maintain the borders (l, r) of the rightmost found (sub-)palindrome (i.e., the current rightmost (sub-)palindrome is s[l+1..r-1]).

- Initially we set l=0, r=1, which corresponds to the empty string.

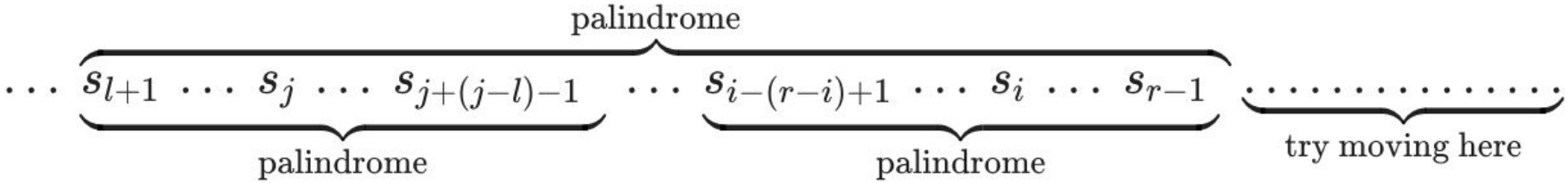
# Manacher's Algorithm: Implementation Details (con't)

- **Case (i≥r)**: check trivially.
- **Case (i<r)**: try to extract some information from the already calculated values. Let's find the "mirror" position of i in the sub-palindrome (l, r). Because j is the position symmetrical to i with respect to (l+r)/2, we can almost always assign d[i]=d[j].



# Manacher's Algorithm: Implementation Details (con't)

- **Tricky Case (i<r)**: "inner" palindrome reached the borders of the "outer" one (i.e., j-d[j]≤l). Because the symmetry outside the "outer" palindrome is not guaranteed, just assigning d[i]=d[j] is incorrect here. After assigning d[i]=r-i, check trivially by increasing d[i] while it is possible.



# Manacher's Algorithm: Complexity

- At the first glance it is **not obvious** that this algorithm has linear time complexity because we often run the naive algorithm while searching the answer for a particular position.
- However, a more careful analysis shows that the algorithm is linear.
- In fact, Manacher looks similar to Z-function.
- Key Observation: **Every iteration of trivial algorithm increases r by one**. Also, r cannot be decreased during the algorithm. So, trivial algorithm will make O(n) iterations in total. Other parts of Manacher's algorithm work obviously in linear time. Thus, we get O(n) time complexity.

# Manacher's Algorithm: Implementation (odd)

```
vector<int> manacher_odd(string s) {
  int n = s.size(), l = 1, r = 1;
  s = "$" + s + "^";
  vector<int> p(n + 2);
  for(int i = 1; i <= n; i++) {
    p[i] = max(0, min(r - i, p[l + (r - i)]));
    while(s[i - p[i]] == s[i + p[i]]) p[i]++;
    if(i + p[i] > r) l = i - p[i], r = i + p[i];
  }
  return vector<int>(begin(p) + 1, end(p) - 1);
}
```

# Manacher's Algorithm: working with parities

- Although it is possible to implement Manacher's algorithm for odd and even lengths separately, the implementation of the version for even lengths is often deemed more difficult, as it is less natural and easily leads to off-by-one errors.
- To mitigate this, it is possible to reduce the whole problem to the case when we only deal with the palindromes of **odd length**. To do this, we can put an additional # character between each letter in the string and also in the beginning and the end of the string:

$$abcbcba \to \#a\#b\#c\#b\#c\#b\#a\# \qquad d = [1, 2, 1, 2, 1, 4, 1, 8, 1, 4, 1, 2, 1, 2, 1]$$

# Manacher's Algorithm: working with parities (con't)

- As you can see, $d[2i]=2d_{even}[i]+1$ and $d[2i+1]=2d_{odd}[i]$ where d denotes the Manacher array for odd-length palindromes in #-joined string while $d_{odd}$ and $d_{even}$ correspond to the arrays defined above in the initial string.

- Indeed, # characters do not affect the odd-length palindromes, which are still centered in the initial string's characters, but now even-length palindromes of the initial string are odd-length palindromes of the new string centered in # characters.

# Manacher's Algorithm: Implementation (full)

```
vector<int> manacher(string s) {
  string t;
  for(auto c: s)
    t += string("#") + c;
  auto res = manacher_odd(t + "#");
  return vector<int>(begin(res) + 1, end(res) - 1);
}
```