String Algorithms

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

Table of Contents

- Given two strings, a text T and a pattern P, find the first position in T where the next $|P|$ letters form P.
- There is a clear, $O(|T|\,|P|)$ time algorithm for this, that iterates over all possible starting positions that even runs in $O(|T| + |P|)$ expected time on random inputs.
- There are several $O(|T| + |P|)$ deterministic algorithms, each with different characteristics.

Hashing

- A *hash function* is a function that takes something as input and returns an integer.
- In this case, our something is a string.
- Since we can compare a machine integer in constant time, if we could efficiently compute our hash function for each substring of size $|P|$, we can obtain a fast algorithm to solve the string matching problem.

Polynomial Hashing

- It turns out that with $O(|S|)$ preprocessing, we can compute a hash value for any substring of a string S in constant time, using polynomial hashing.
- We take the ASCII values of each character in the string as coefficients of some polynomial, and evaluate it at some base B to obtain our hash value.
- Then the hash of the first i characters will be

$$S_0 B^{|S|} + S_1 B^{|S|-1} + \cdots + S_{i-1} B^{|S|-i+1}.$$

- We also implicitly use `INT_MAX + 1` as a modulus, to keep the possible hash values to a reasonable size.
  - This is okay because a power of two is coprime to all primes other than two, so we just need to pick a prime B.

Polynomial Hashing

- To quickly compute the hash value for any substring, we first compute the hash value for each prefix of the string S and store it in an array H.
- Then, to obtain a hash value for any substring $s_i s_{i+1} \ldots s_{j-1}$, we can calculate $(H_j - H_i) * B^i$.
- Somewhat unintuitively, we are actually using polynomials of a fixed degree, $|S| - 1$, taking the values of our characters as the higher order coefficients, and filling the lower order coefficients with zeroes.
- Effectively, the multiplication by B^i serves to "shift up" the subset of coefficients we have selected to be the ones for the highest powers, regardless of where in the string we are. Why is this important?

Polynomial Hashing

- The values H_i and B^i mentioned previously can easily be computed in linear time, which allows us to easily compute the hash value of each substring with a simple expression
- Now, to solve the string matching problem, we just need to compute the hash value of our pattern P and compare it against the hash values of all $O(|T|)$ substrings of T which are of length $|P|$.

- **Implementation**

```
void init() {
  B[0] = 1;
  B[1] = 37; // pick a prime
  // compute all the powers
  for (int i = 2; i < n; i++) {
    B[i] = B[i-1] * B[1];
  }

  H[0] = 0;
  for (int i = 1; i <= n; i++) {
    // compute the hash of [0, i)
    // H[i] = s[0]*B[n] + s[1]*B[n-1] + ... + s[i-1]*B[n-i]
    H[i] = H[i-1] + s[i-1] * B[n-i];
  }
}

// the hash of [i, j) in the string
int hash (int i, int j) {
  return (H[j] - H[i]) * B[i];
}
```

Polynomial Hashing

- Obviously, the set of possible strings that we can hash is much larger than the total number of possible values of a machine word, so we will have collisions.
- However, if we pick a good base and a good modulus, the chance of a collision occurring is very low.
- Strictly speaking, we want to pick large random primes for our base and our modulus, but in practice it's easier to just use some nice constant value, like the ones in the code.

Robustness of Polynomial Hashing

- Although it's not always necessary, there are several techniques to improve robustness of hashing:
  - Use actual random probable primes for our base and modulus.
  - Hash multiple times with different bases and moduli, and compare every hash value to check for equality.
  - Use a larger data type (64-bit integers).
  - When finding a match, **check using the naïve algorithm if the match is a real match**. Since the probability of finding a collision is very low, the expected number of times this algorithm will need to be run is also very low.

Robustness of Polynomial Hashing

- It turns out that for arbitrary known bases and moduli, it's possible to quickly generate strings which will give a collision.
- But usually, such test cases won't appear.

Z Algorithm

- The Z algorithm computes the Z function for some string S, where Z_i is the length of the longest substring starting at s_i which is a prefix of S.
- If we had a method of computing the Z function quickly, then we can solve the string matching problem as well by computing the Z function for the string $P\$T$ and walking over the array, where '\$' is a special character that does not occur in P or T.

Z-boxes

- We can compute the Z function in linear time for some string S by using a concept called a Z-box.
- A Z-box for some position $i > 0$ of S is a pair $[l, r)$ where $1 \leq l \leq i < r \leq n$, r is maximal, and the substring $s_l, \ldots, s_{r-1}$ is equal to some prefix of S.
- Given the Z-box for every position i, we can easily compute the corresponding values Z_i.

Z-boxes

- To compute a Z-box for the second character s_1 of S (the value for the first character s_0 is trivial), we can just naïvely compare substrings starting at the first character.
- Now, for positions $i > 1$, assuming that we've already computed the Z-box for position $i - 1$, we have a few cases to consider.

Z-boxes

- If $i \geq r$ for our previous Z-box then we can just start at i and naïvely compute a new Z-box starting at i and get Z_i from that.
- Otherwise, position i is contained in our previous Z-box.
- Let $k = i - l$. We know that we already have the value of Z_k computed, which we can use to see if we can extend our current Z-box.

Z-boxes

- We care about Z_k because we're trying to extend a k-character prefix we already know about *with a string that is also a prefix*, i.e. $Z_i \geq Z_k$.
- If $Z_k < r - i$, then the substring starting at index k which is a prefix of S is shorter than the distance from our current i to the end of the Z-box, which means Z_i can't be more than Z_k (or we could just extend Z_k more). So we set $Z_i = Z_k$ in this case.
- Otherwise, we might get a higher r by creating a new Z-box with $l = i$, so we again apply our naïve algorithm and obtain a new maximal r.
- Since our values l, r and i each increase monotonically from 0 to $|S|$, this algorithm runs in time $O(|S|)$.

Z Algorithm

Implementation

```
vector<int> z_algorithm(char* s) {
  int n = strlen(s);
  vector<int> z(n);
  int l = r = 0;
  for (int i = 1; i < n; i++) {
    // if we've gone outside the current z-box, start a new one
    if (i >= r) {
      l = r = i;
      while (s[r - l] == s[r]) r++;
      z[i] = r - l;
    } else {
      int k = i - l;
      // if we can use already computed information, do so
      if (z[k] < r - i) z[i] = z[k];
      // otherwise, expand the z-box as much as possible
      else {
        l = i;
        // we don't need to reset r because we already know the string
            matches earlier
        while (s[r - l] == s[r]) r++;
        z[i] = r - l;
      }
    }
  }
  return z;
}
```

Knuth-Morris-Pratt

- The Knuth-Morris-Pratt algorithm is the classical, deterministic linear time string matching algorithm.
- It is based on the idea that the naïve string matching algorithm is made slow by looking for matches for the same character from T multiple times, even after determining that there does not exist a match.

Knuth-Morris-Pratt

- Let's say we're looking for the string "ABACABAB" in the string "ABACABADABACABAB".
- In the naïve algorithm, we will start at position 0 and fail at position 7 of T, because B does not match D.
- Immediately, we can see that there's no point trying to start matching from positions 1, 2 and 3 will fail as well, so there's no point considering them at all.

Knuth-Morris-Pratt

- After some thought, we can see that if we've matched some prefix P' of our pattern P, then the number of characters we should skip can be computed from the longest prefix of P' that is a suffix of P'.
- If we could quickly compute for every prefix of our pattern this "longest prefix-suffix" value, called the *prefix function*, it turns out we can use these values to speed up the naïve matching algorithm in linear time.

Knuth-Morris-Pratt

- We can compute the prefix function by using a similar idea to string matching with this prefix function on our pattern.
- If we've already computed the prefix function pre_i for prefixes of length $0, 1, \ldots, i-1$ for some pattern P, then to compute the prefix function for position i, we can try to extend the prefix-suffix for the prefix with length $i-1$.
- If we can't extend the prefix-suffix for $i-1$, we have to go back and try to extend the prefix-suffix for the prefix with length pre_{i-1}.
- Failing that, we try $pre_{pre_{i-1}}$ until we reach the empty prefix.

- **Implementation**

```
vector<int> kmp(string s) {
  int n = s.size();
  vector<int> pre(n);
  for (int i = 1; i < n; i++) {
    int j = i;
    // keep making the pre/suffix we're extending shorter until it works
    while (j > 0 && s[i] != s[pre[j - 1]]) j = pre[j - 1];
    // did we end up finding anything?
    pre[i] = (j > 0) ? pre[j - 1] + 1 : 0;
  }
  return pre;
}
```

Knuth-Morris-Pratt

- If we match i characters of P starting at index m in a search string T, and then fail, we continue from index $m + i - pre_i$ in T, and pre_i in P, conveniently avoiding matching the same character in T more than once.
- KMP's ability to provide deeper insight into the string matching process makes it useful for more complicated problems.
- Given an alphabet A and a string S, how many strings of length N $(1 \leq N \leq 1,000,000,000)$ using only symbols from the alphabet do not contain S?

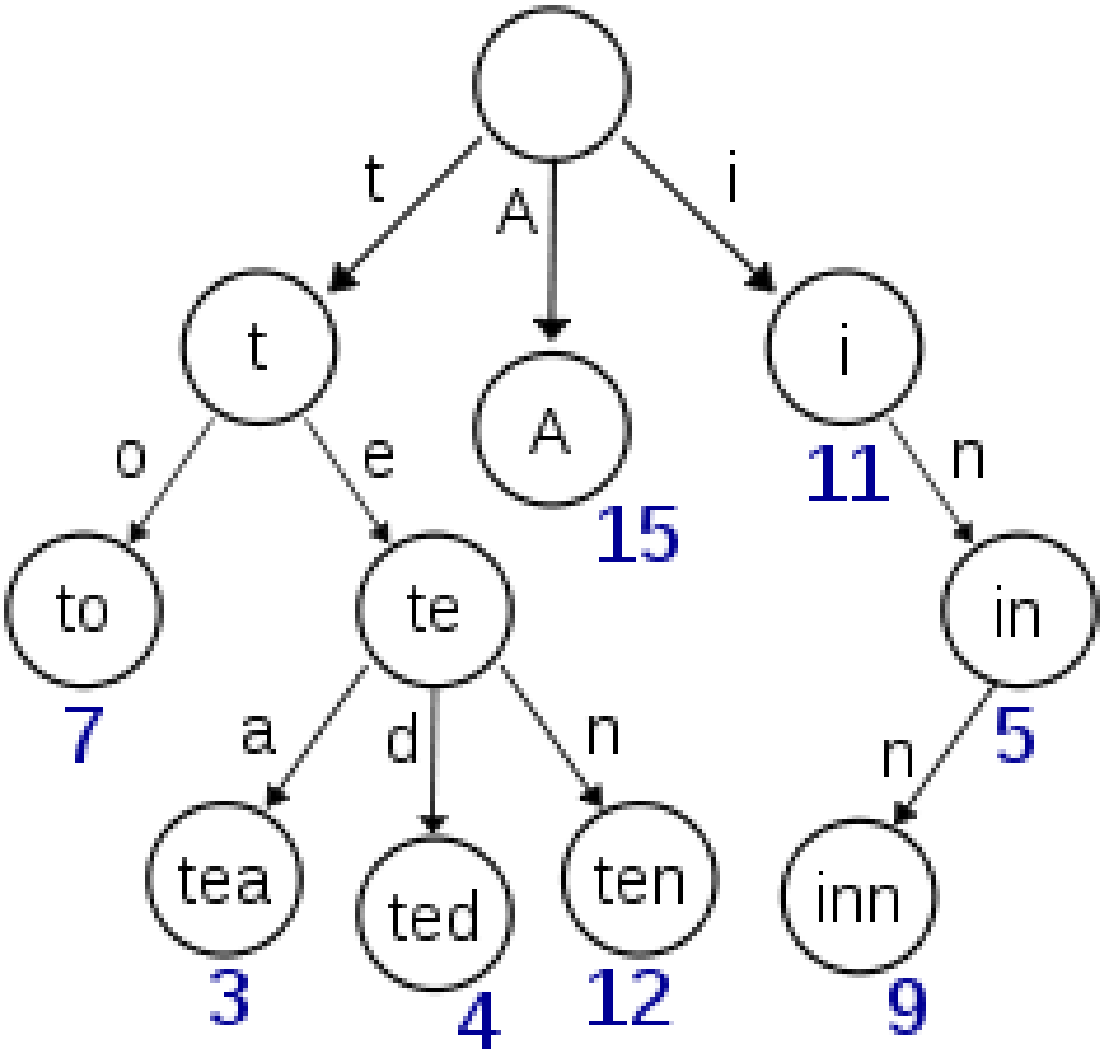
- All three presented algorithms each have their own strengths and weaknesses.
- For general string matching, hashing is the most useful, while the Z algorithm has no probability of failure and can solve many problems (but not all) which rely on the properties of the prefix function.
- Choose the one which affords the most natural reduction.
- In fact, the array *pre* for a string can be computed from the array Z and vice versa in linear time.

Table of Contents

Prefix Trees

- A prefix tree (also known as a *trie*) is a data structure which represents a dictionary of words.
- It is a rooted tree where every path from the root to a leaf of the tree is a word in the dictionary.
- Each vertex in the tree represents a symbol in the alphabet, except for the leaves, which represent string terminators.
- If any two words in the dictionary share a prefix, then they also share a path from the root of the tree which represents that prefix.

Prefix Trees



Prefix Trees

- It's easy to imagine a method to insert words into this tree in $O(|L|)$ time, where L is the length of the word.
- Furthermore, this can be used for set and map lookups in $O(L)$ time as well.
- There is an analogous relationship between prefix trees and radix-based sorting methods as well as binary search trees and partition-based sorting methods such as quick sort.

- **Implementation**

```
struct trie {
  int size;
  int next[MAXN][ALPHA];

  trie() : size(1) {}

  int makeNode() {
    return size++;
  }

  void reset() {
    fill(&next[0][0], &next[0][0] + size*ALPHA, 0);
    size = 1;
  }

  // doesn't store any info to indicate the end of a word; you should add
       that if you need it
  void insert(const char* s, int i) {
    int p = 0;
    while (*s) {
      if (!next[p][(int)*s]) next[p][(int)*s] = makeNode();
      p = next[p][(int)*s];
      s++;
    }
  }
};
```

Aho-Corasick

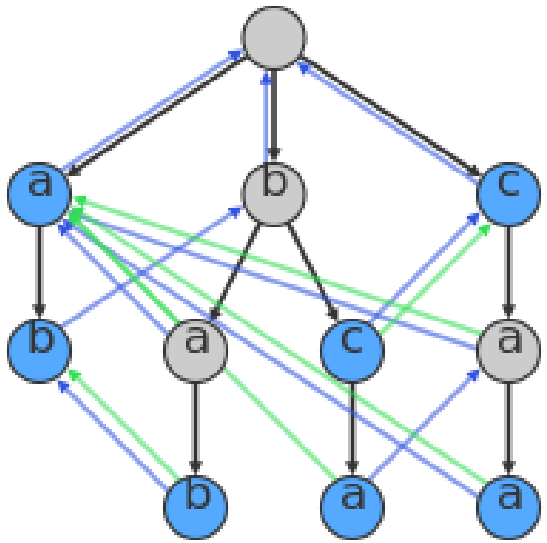
- The Aho-Corasick algorithm is a generalisation of KMP that searches for the matches of a dictionary of words in some string T, instead of a single string.
- It works by first computing the prefix tree of the dictionary to be matched, and then computing the same failure function for every vertex in this prefix tree.
- Because we are no longer working with a single string, but a tree, the failure function must be computed with a BFS instead of iteration over an array, but otherwise the algorithm is exactly the same.

Aho-Corasick

- The edges that the failure function represents on the prefix tree are called suffix links.
- An important thing to note is that a suffix link from some vertex does not necessarily point to an ancestor of that vertex.
- To find the first position that every word in some dictionary D occurs in some text T, using Aho-Corasick, takes time on the order of the sum of the lengths of all the strings in D, plus the length of the string T.

Aho-Corasick

- A visualisation for the dictionary {a, ab, bab, bc, bca, c, caa}:



- Blue: in dictionary, grey: not
- Blue arrow: shortcut to the longest suffix of this string
- Green arrow: shortcut to the next thing that's in the dictionary if you follow blue arrows

- **Implementation (trie)**

```
struct trie {
  int size;
  int f[MAXN];
  int next[MAXN][ALPHA];
  vector<int> matches[MAXN];
  trie() : size(1) {}
  int makeNode() {
    return size++;
  }

  void reset() {
    fill(matches, matches + size, vector<int>());
    fill(f, f + size, 0);
    fill(&next[0][0], &next[0][0] + size*ALPHA, 0);
    size = 1;
  }

  void insert(const char* s, int i) {
    int p = 0;
    while (*s) {
      if (!next[p][(int)*s])
        next[p][(int)*s] = makeNode();
      p = next[p][(int)*s];
      s++;
    }
    matches[p].pb(i);
  }
  // to be continued
```

- **Implementation (suffix links/failure function)**

```
int fail(int x, int ch) {
  while (x > 0 && !next[x][ch])
    x = f[x];
  return x;
}

void compute() {
  queue<int> q;
  for (q.push(0); !q.empty(); q.pop()) {
    int v = q.front();
    for (int i = 0; i < ALPHA; i++) {
      int u = next[v][i];
      if (u) {
        q.push(u);
        int x = fail(f[v], i);
        f[u] = v > 0 ? next[x][i] : 0;
        matches[u].insert(matches[u].end(), matches[f[u]].begin(),
              matches[f[u]].end());
      }
    }
  }
};
```

- **Implementation (usage)**

```
ac.compute();
int p = 0;
for (int i = 0; i < n; i++) {
  char c = s[i];
  p = ac.fail(p, c);
  p = ac.next[p][c];
  // add ac.matches[p] to output set
}
```

Table of Contents

Suffix Arrays

- A suffix array is an array which represents all of the suffixes of a string in lexicographical order.
- It is usually paired together with a longest common prefix array, which stores the length of the longest common prefix of the adjacent suffixes in the array.
- There is a trivial algorithm which computes a suffix array of a string S in $O(|S|^2 \log |S|)$ time, as well as a couple of relatively easy to understand algorithms which run in $O(|S| \log^2 |S|)$ time. The best known algorithm runs in $O(|S|)$.

Suffix Arrays

- The bottleneck in the obvious $O(|S|^2 \log |S|)$ time algorithm is in the comparison of different substrings, which can take up to $O(|S|)$ time.
- However, notice that if we use hashing, we can binary search for the location of the longest common prefix of two substrings in $O(\log |S|)$ time, and then just check the first character that differs for which comes lexicographically earlier, in constant time.
- With an $O(\log |S|)$ comparison function, we can then plug in any fast sorting algorithm to obtain a suffix sort that takes $O(|S| \log^2 |S|)$ time total.
- To compute the LCP array, we can again use our binary search and hash based approach.

Suffix Arrays

- There exists another $O(|S| \log^2 |S|)$ time suffix sort, which does not use hashing, but is a little more complicated, as well as a way to compute the LCP array in $O(|S|)$ time after suffix sorting.
- Both of these follow, without proof. The basic idea is to sort suffixes by their first character, then by their first two characters, then by their first four characters, and so on.

- **Implementation**

```
int pos[MAXN], rank[MAXN], tmp[MAXN], lcp[MAXN];
int cmpsz;
int length;

bool sufcmp(int i, int j) {
  return rank[i] == rank[j] ? (i + cmpsz < length && j + cmpsz < length ?
        rank[i + cmpsz] < rank[j + cmpsz] : i > j) : rank[i] < rank[j];
}

void construct_sa() {
  length = strlen(s);
  for (int i = 0; i < length; i++) {
    pos[i] = i;
    rank[i] = s[i];
  }

  // to be continued
```

- **Implementation (continued)**

```
  for (cmpsz = 1; cmpsz >> 1 < length; cmpsz += cmpsz) {
    sort(pos, pos + length, sufcmp);
    for (int i = 1; i < length; i++)
      tmp[i] = tmp[i - 1] + sufcmp(pos[i - 1], pos[i]);
    for (int i = 0; i < length; i++)
      rank[pos[i]] = tmp[i];
  }

  lcp[0] = 0;
  for (int i = 0, h = 0; i < length; i++)
    if (rank[i] > 0) {
      for (int j = pos[rank[i] - 1]; s[i + h] == s[j + h]; h++);
      lcp[rank[i]] = h;
      if (h > 0)
        h--;
    }
}
```

Table of Contents

Making Palindromes

- **Problem statement** Given some string S $(1 \leq |S| \leq 1,000,000)$, what is the least number of symbols that need to be added to the end of S to make a palindrome? You are only allowed to add letters to the end of S. Output the length of the final palindrome.
- **Example** The string "pqrq" can be extended to "pqrqp", and so has answer 5. The string "a" is already a palindrome, so has answer 1.

Making Palindromes

- Let's begin by examining the definition of a palindrome: a palindrome reads the same from front to back as it is read from back to front.
- From this, it seems sensible to consider the reverse of our input string.

Making Palindromes

- Further, let's look at what a potential solution looks like.
- A trivial possible solution is to just extend the string by appending its reverse, but this is clearly not optimal.
- How can we improve this solution?

Making Palindromes

- From this, it can be noticed that all valid solutions are formed by taking the input string and overlaying its reverse over this string, so that all overlapping characters match and the length of the total string is minimal.
- We can find such a minimal overlap by trying to match prefixes of the string's reverse with suffixes of the input.
- KMP, Z algorithm, hashing and suffix arrays can all achieve this quickly.

Making Palindromes

- **Implementation**

```
int n = strlen(s);
strncpy(s + n + 1, s, n);
reverse(s, s + n);
s[n] = '$';
vector<int> z = z_algorithm(s);
for (int i = n + 1; i < n + n + 1; i++) {
  if (z[i] == n + n + 1 - i) {
    printf("%d\n", i - (n + 1) + n);
    break;
  }
}
```

Least Lexicographical Rotation

- **Problem statement** A rotation of some string S $(1 \leq |S| \leq 1,000,000)$ is a string that is obtained by removing some suffix of S and attaching to the front. Find the rotation of S that comes first when considered in lexicographic order.
- **Example** The lexicographically least rotation of "bbaaccaadd" is "aaccaaddbb".

Least Lexicographical Rotation

- When considering rotations of a string, it's very useful to consider the string concatenated with itself, because every rotation is a length $|S|$ substring of this doubled string.
- If we now suffix sort this double string, we can read the answer directly off the suffix array.
- However, there is a simpler, more fundamental method to solve this problem.

Least Lexicographical Rotation

- Recall that one of the methods we used to construct a suffix array was to use hashing and binary search to lexicographically compare two strings in $O(\log |S|)$ time.
- We can use this same comparison algorithm to find the lexicographically least substring of size $|S|$ in our doubled string, in a similar way to finding the minimum value of an array to solve this problem in $O(|S| \log |S|)$ time total.

- **Implementation (setup)**

```
int k;
ll hash(int i, int size) {
  return H[i] - H[i + size] * P[size];
}

s[n] = 0;
strncpy(s + n, s, n);
H[n+n] = 0;
for (int i = n+n - 1; i >= 0; i--) {
  H[i] = H[i + 1] * p + s[i];
}
```

- **Implementation (comparison)**

```
bool cmp(int i, int j) {
  if (hash(i, k) == hash(j, k))
    return false;
  int lo = 0, hi = k;
  while (hi - lo > 1) {
    int mid = (lo + hi) >> 1;
    if (hash(i, mid) == hash(j, mid))
      lo = mid;
    else
      hi = mid;
  }
  return s[i + lo] < s[j + lo];
}

int best = 0;
for (int i = 0; i < n; i++)
  if (cmp(i, best))
    best = i;
```

Growing Strings

- **Problem statement** We say that some string S can be grown from some other string T if T is a substring of S. Given a set of strings D ($1 \leq |D| \leq 10,000$), what is the size of the longest sequence of strings from D that can be created such that each string in the sequence can be grown from the previous string? Each string is up to 1,000 characters long and there are no more than 1,000,000 characters in the entire input set.
- **Example** The string "ant" can grow into the strings "cant" and "plant". If these strings formed D, then the answer would be 2, because "ant" can grow into both of the other words, but neither of the other two can grow into the remaining one.

Growing Strings

- If we had a fast way of computing the "can be grown from" relation and then consider these relations as edges, then the problem reduces to finding the longest path in a DAG, which can be solved in $O(|D|^2)$ using dynamic programming.
- Computing this graph is tricky though, because a naïve algorithm requires a linear time comparison to check for the existence of every edge in the DAG, which will time out.

Growing Strings

- Notice however, that computing the edges of the DAG is equivalent to string matching for every word in a dictionary in every word in a dictionary, which we can solve with Aho-Corasick in time linear to the total number of characters in the input set.
- So we can directly run the Aho-Corasick algorithm on every string in our input, to construct a DAG, from which we compute the longest path as before.

- **Implementation (graph construction)**

```
trie ac;
ac.compute();
vector< vector<int> > adj(n);
for (int i = 0; i < n; i++) {
  int p = 0;
  for (int j = 0; j < sz(dict[i]); j++) {
    p = ac.fail(p, dict[i][j] - 'a');
    p = ac.next[p][dict[i][j] - 'a'];
    adj[i].insert(adj[i].end(), ac.matches[p].begin(), ac.matches[p].end()
        );
  }
}

for (int i = 0; i < n; i++) {
  set<int> t(adj[i].begin(), adj[i].end());
  adj[i] = vector<int>(t.begin(), t.end());
}
```

- **Implementation (longest path)**

```
int max_depth(int u, vector<vector<int> > & adj, vector<int>& d) {
  if (d[u] == -1) {
    int res = 0;
    for (int i = 0; i < sz(adj[u]); i++) {
      if (adj[u][i] == u)
        continue;
      res = max(res, max_depth(adj[u][i], adj, d));
    }
    d[u] = res + 1;
  }
  return d[u];
}

vector<int> d(n, -1);
int res = 0;
for (int i = 0; i < n; i++) {
  res = max(res, max_depth(i, adj, d));
}
```