文字列 DP (応用編)

文字列を華麗に扱おう

tsutaj (@_TTJR_)

Hokkaido University M1

July 27, 2018

1. LCS・編集距離の類題
   - ManageSubsequences
   - YahooYahooYahoo
2. 禁止文字列
   - 禁止文字列 (単一 ver.)
   - 禁止文字列 (複数 ver.)
   - Genetic engineering
3. 練習問題

ManageSubsequences

ManageSubsequences

文字列 S, A, B が与えられる。S に対して、(連続とは限らない) 部分列として A を含むが B を含まないように、いくつか文字を挿入したい。この条件を満たす文字挿入個数の最小値を求めよ。不可能な場合は -1 を出力せよ。

- $1 \leq |S|, |A|, |B| \leq 300$

出典: TopCoder SRM 727 Div2 Hard ▸ Link

サンプル

入力 1 :	`S = "ABXBCA", A = "ABCD", B = "XD"`	出力: 2
入力 2 :	`S = "BANANA", A = "APPLE", B = "ANNA"`	出力: -1

ManageSubsequences

- いったん B のことを無視する
  - S 中に部分列として A を含むための、最小の文字挿入個数は？

ManageSubsequences

- いったん B のことを無視する
  - S 中に部分列として A を含むための、最小の文字挿入個数は？
- これは LCS とほぼ同じ考えで解ける
  - $\mathrm{dp}[i][j] := S$ の i 文字目までを使って、A の j 文字目までを含むようにするために必要な文字挿入個数の最小
  - 遷移は次の 2 通り
    1. S の次の文字を普通に入れる
    2. A の j 文字目に相当する文字を入れる
  - 計算量 $O(|S||A|)$
- これに B が加わるとどうなるか？

ManageSubsequences

- いったん B のことを無視する
  - S 中に部分列として A を含むための、最小の文字挿入個数は？
- これは LCS とほぼ同じ考えで解ける
  - $\mathrm{dp}[i][j] := S$ の i 文字目までを使って、A の j 文字目までを含むようにするために必要な文字挿入個数の最小
  - 遷移は次の 2 通り
    1. S の次の文字を普通に入れる
    2. A の j 文字目に相当する文字を入れる
  - 計算量 $O(|S||A|)$
- これに B が加わるとどうなるか？
- A を含むが B は含まない → B と最後まで一致する前に、S と A について最後まで一致
- 上の DP の次元を増やし、以下のように解けばよい！
  - $\mathrm{dp}[i][j][k] := S$ の i 文字目までを使って、A の j 文字目まで、B の k 文字目までを含むようにするために必要な文字挿入個数の最小

ManageSubsequences

実装 (B を無視し、S に A を含ませる DP)

```
dp[0][0] = 0;
for(int i=0; i<=len_s; i++) {
    for(int j=0; j<=len_a; j++) {
        // insert S[i]
        if(i < len_s) {
            int nxt_s = i+1;
            int nxt_a = j + (j < len_a && S[i] == A[j]);
            dp[nxt_s][nxt_a] = min(dp[nxt_s][nxt_a], dp[i][j]);
        }

        // insert A[j]
        if(j < len_a) {
            int nxt_s = i + (i < len_s && A[j] == S[i]);
            int nxt_a = j+1;
            dp[nxt_s][nxt_a] = min(dp[nxt_s][nxt_a], dp[i][j] + 1);
        }
    }
}
```

ManageSubsequences

実装 (元々の問題の DP)

```
dp[0][0][0] = 0;

for(int i=0; i<=len_s; i++) {
    int cur = i % 2, nxt = cur ^ 1;
    for(int j=0; j<=len_a; j++) {
        for(int k=0; k<len_b; k++) {
            // insert S[i]
            if(i < len_s) {
                int nxt_a = j + (j < len_a && S[i] == A[j]);
                int nxt_b = k + (k < len_b && S[i] == B[k]);
                chmin(dp[nxt][nxt_a][nxt_b], dp[cur][j][k]);
            }

            // insert A[j]
            if(j < len_a) {
                int nxt_mo = (cur + (i < len_s && A[j] == S[i])) % 2;
                int nxt_b  = k + (k < len_b && A[j] == B[k]);
                chmin(dp[nxt_mo][j+1][nxt_b], dp[cur][j][k] + 1);
            }

            if(i != len_s) dp[cur][j][k] = INF;
        }
    }
}
```

YahooYahooYahoo

YahooYahooYahoo

文字列 S が与えられる。S と、「“yahoo” を 0 回以上繰りかえしてできた文字列」との編集距離の中で最小のものを求めよ。

- $1 \leq |S| \leq 10^5$

出典: 「みんなのプロコン」本戦 A ▸ Link

サンプル

入力 1 :	`"yfoo"`	出力:	2
入力 2 :	`"z"`	出力:	1
入力 3 :	`"yahooyahooyahooyahooyahoo"`	出力:	0

YahooYahooYahoo

- “yahoo” を繰り返してできた文字列を “yahoo 文字列” と称する
- S の先頭から、yahoo 文字列の $j \bmod 5$ 文字目まで一致させるためにコストがいくつ必要か考えよう
- $\mathrm{dp}[i][j] := S$ の i 文字目までを、yahoo 文字列の $j \bmod 5$ 文字目まで一致させるために必要なコストの最小値 として解けば良い
  - 基本的には普通の編集距離 DP
  - わからない人は前回の DP スライドを見よう！
- “yahoo” は 5 文字しかないので、5 を定数としてみると計算量 $O(|S|)$

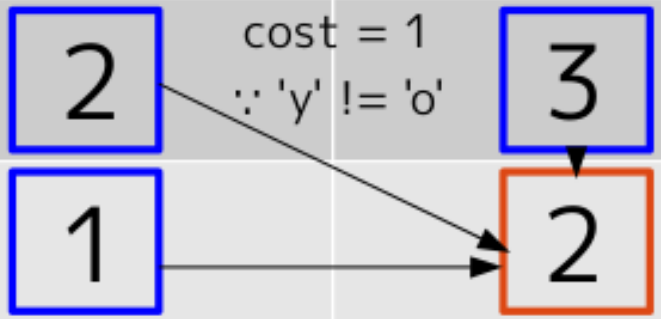
YahooYahooYahoo

基本的には、この表を更新していくことを考えれば良い

	yfoo	yfoo
...yahoo	2	3
...y	1	2
...ya	1	2
...yah	2	2
...yaho	3	2

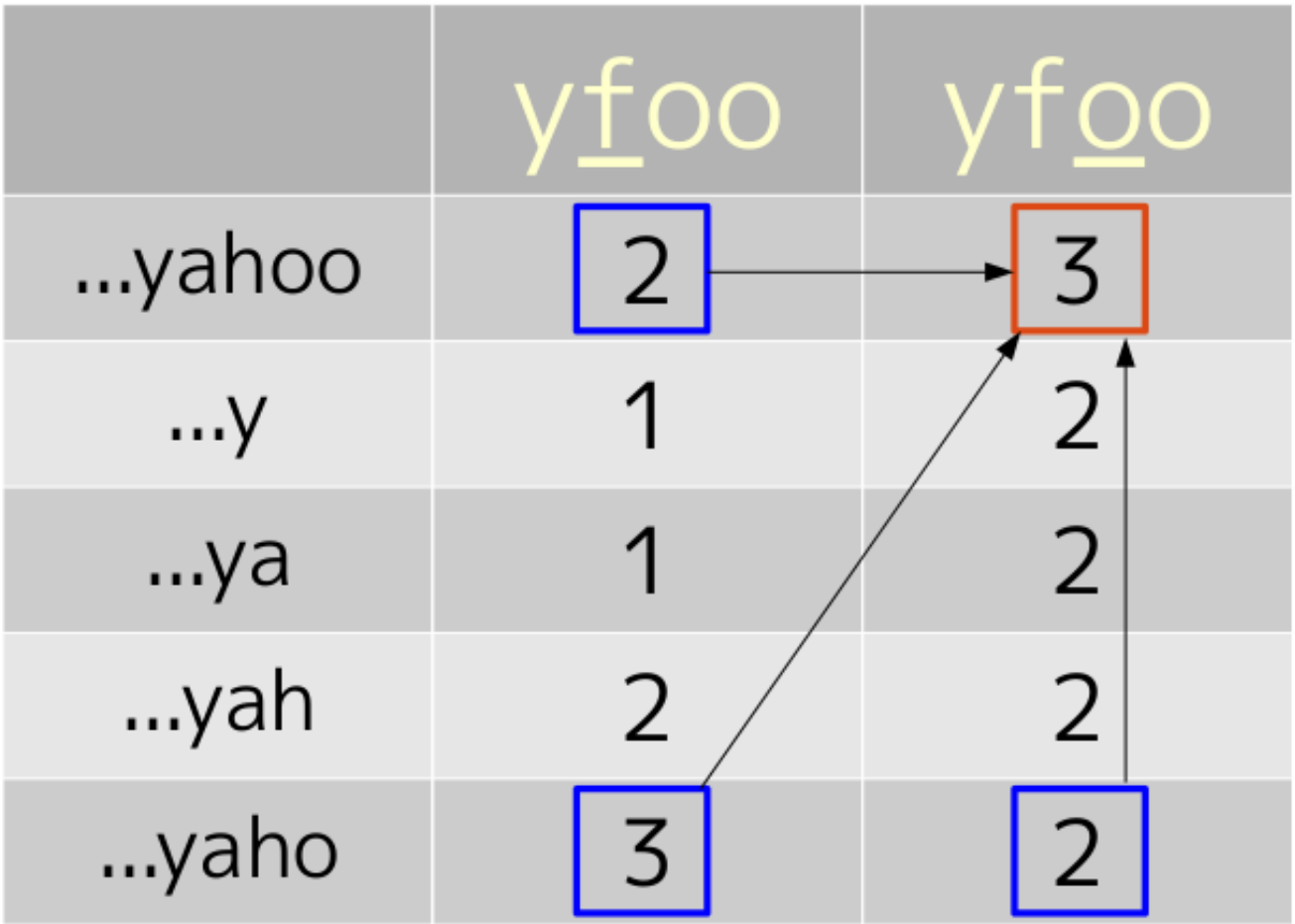
更新時に参照する場所

	yfoo	yfoo
...yahoo	2	3
...y	1	2
...ya	1	2
...yah	2	2
...yaho	3	2



YahooYahooYahoo

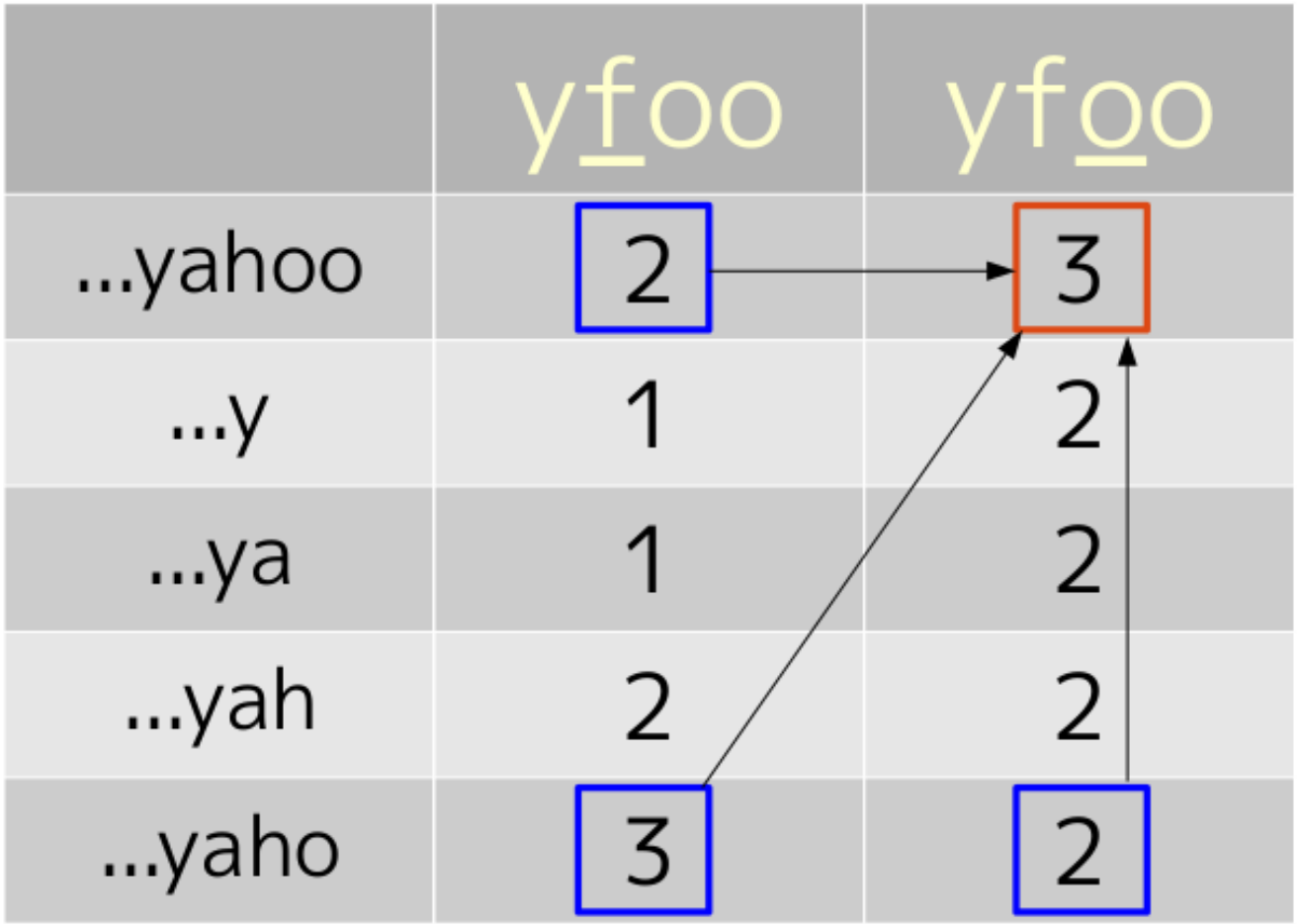
ところで、普通に更新すると以下のような問題点がある



上から順番に更新していくなら
ここの最適値は未知では？？

YahooYahooYahoo

DP を更新するときは、ループを 2 回まわそう！



上から順番に更新していくなら
最適値は未知では？？ → **DP のループ 2 周で解決！！**

備考 (2 周のループで最適性が保証されることの証明)

- 「文字置換」と「文字削除」の動作に関して、既に値が確定しているところから状態遷移が発生するので、これらの操作による最適性は保証されている
  - 以下、「文字追加」の最適性について議論しよう

YahooYahooYahoo

備考 (2 周のループで最適性が保証されることの証明)

- 「文字置換」と「文字削除」の動作に関して、既に値が確定しているところから状態遷移が発生するので、これらの操作による最適性は保証されている
  - 以下、「文字追加」の最適性について議論しよう
- “yahoo 文字列” は 5 文字 1 周期なので、5 文字以上の文字挿入が最適になるパターンは存在しない！
- よって、0 〜 4 文字の挿入が最適になる可能性がある

YahooYahooYahoo

備考 (2 周のループで最適性が保証されることの証明)

- 「文字置換」と「文字削除」の動作に関して、既に値が確定しているところから状態遷移が発生するので、これらの操作による最適性は保証されている
  - 以下、「文字追加」の最適性について議論しよう
- “yahoo 文字列” は 5 文字 1 周期なので、5 文字以上の文字挿入が最適になるパターンは存在しない！
- よって、0 〜 4 文字の挿入が最適になる可能性がある
- 状態は “...yahoo”, “...y”, “...ya”, “...yah”, “...yaho” の 5 通り存在するのだから、このうちのいずれかは 0 文字挿入が最適であるはず
- したがって、1 周目のループでいずれかの状態について最適性が保証されるので、2 周回すことで全ての状態について最適性が保証される

YahooYahooYahoo

実装は以下の通り

```
const int INF = 1 << 28;
int dp[100010][5];

int main() {
    string s, pat = "yahoo"; cin >> s;
    int N = s.length();

    fill(dp[0], dp[100010], INF);
    dp[0][0] = 0;
    for(int i=0; i<N; i++) {
        for(int j=0; j<11; j++) {
            int cur = j%5, nxt = (j+1)%5;
            int cost = !(s[i] == pat[cur]);

            dp[i+1][nxt] = min({dp[i  ][cur] + cost,   // replace
                                dp[i  ][nxt] + 1    ,   // delete
                                dp[i+1][cur] + 1    }); // insert
        }
    }
    cout << dp[N][0] << endl;
    return 0;
}
```

用語説明

禁止文字列の解説の前に、いくつか用語の説明

Prefix (接頭辞)

文字列の先頭から数文字とってきてできた文字列

- “akemihomura” の prefix の例
  - “”, “ak”, “akem”, “akemihom”, “akemihomura” など
  - このスライドでは、空でも良いし文字列全体でも良いとしています

Suffix (接尾辞)

文字列の後尾から数文字とってきてできた文字列

- “kanamemadoka” の suffix の例
  - “”, “oka”, “madoka”, “memadoka”, “kanamemadoka” など
  - このスライドでは、空でも良いし文字列全体でも良いとしています

禁止文字列 (単一 ver.)

禁止文字列 (単一 ver.)

'A', 'G', 'C', 'T' の 4 種類の文字のみが含まれる文字列を DNA 文字列と称する。K 文字の文字列 S が与えられるので、N 文字からなる DNA 文字列であって S を含まない文字列の個数 $\mathrm{mod} 10,009$ を求めよ。

- $1 \leq K \leq 10^2$
- $1 \leq N \leq 10^4$

出典: 蟻本 P.327

サンプル

入力: `N = 3, K = 2, S = "AT"` 出力: `56`

- 条件を満たす文字列の例: `"ACG"`, `"AGT"`
- 条件を満たさない文字列の例: `"ATC"`, `"CAT"`, `"ACGG"`

何を状態として持つべき？

禁止文字列 (単一 ver.)

何を状態として持つべき？

- S を含まない → S と N 文字目まで一致してはいけない
- 現在の文字列の suffix が S とどれだけ一致しているかわかれば OK

禁止文字列 (単一 ver.)

何を状態として持つべき？

- S を含まない → S と N 文字目まで一致してはいけない
- 現在の文字列の suffix が S とどれだけ一致しているかわかれば OK
- $\mathrm{dp}[i][j] :=$ 作成している DNA 文字列が i 文字であって、その suffix が S の先頭 j 文字と一致している状態の総数 とする
- 適当にやるとハマるので注意！

禁止文字列 (単一 ver.)

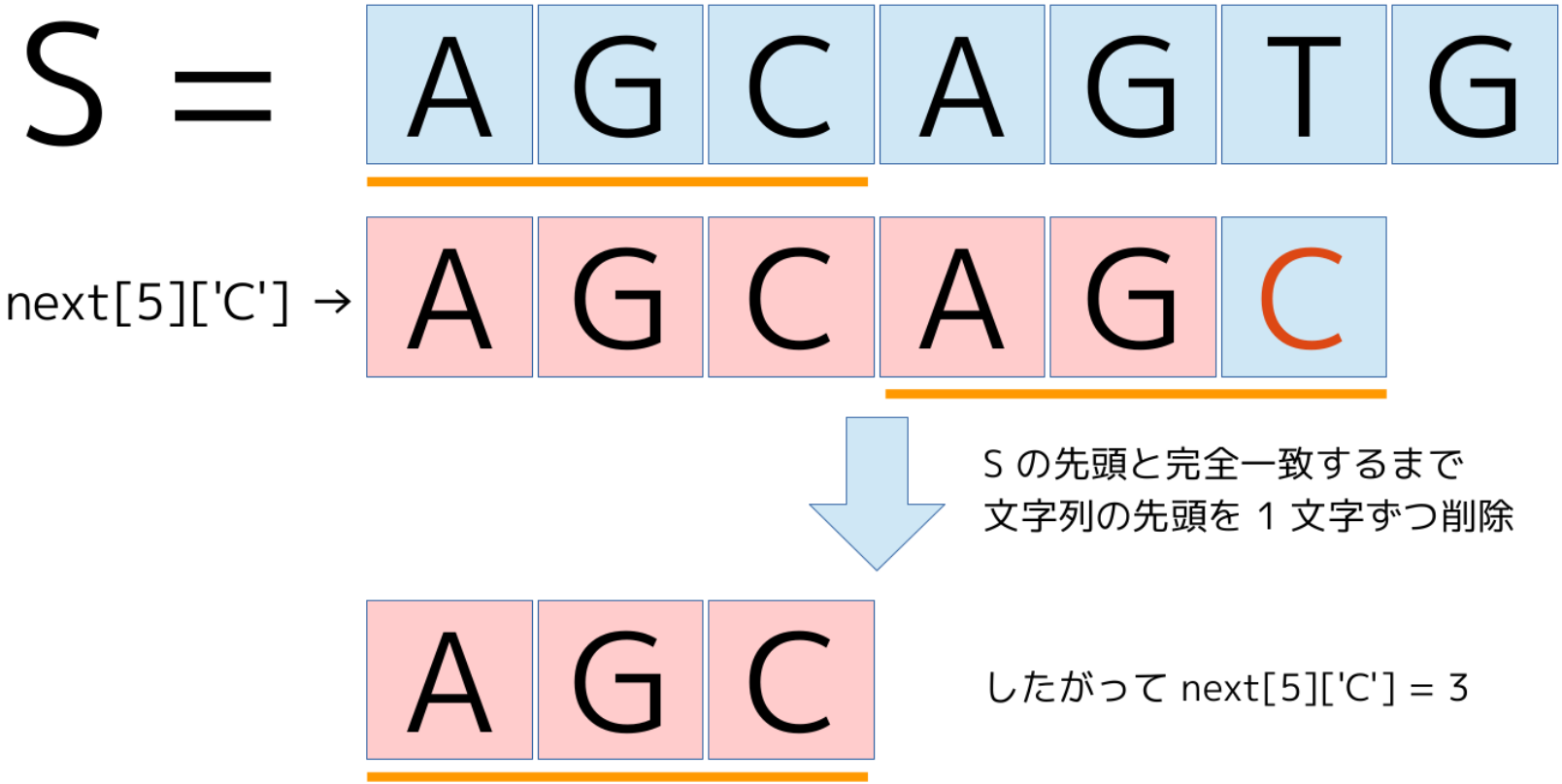
何を状態として持つべき？

- S を含まない → S と N 文字目まで一致してはいけない
- 現在の文字列の suffix が S とどれだけ一致しているかわかれば OK
- $\mathrm{dp}[i][j] :=$ 作成している DNA 文字列が i 文字であって、その suffix が S の先頭 j 文字と一致している状態の総数 とする
- 適当にやるとハマるので注意！
  - $S =$ "AGCAGTG" で、S の 5 文字目まで一致 ("AGCAG") する場合
  - 次の文字として 'C' を採用 → これは、S の先頭何文字目までと一致？
  - $S =$ "AGCAGTG", 作った文字列 = "...AGCAGC" → 先頭 3 文字
- 「S の先頭 x 文字目まで一致していて、それに文字 c を末尾に挿入した場合に、先頭何文字目まで一致するか？」の情報が必要

禁止文字列 (単一 ver.)

以下の前処理を考える

- $\mathrm{next}[x][c] :=$ S の先頭 x 文字目まで一致、それに文字 c を末尾に挿入した場合に、先頭何文字目まで一致するか？
- $O(K^3)$ で構築可能



禁止文字列 (単一 ver.)

この前処理が終われば、あとは素直な DP に

- $\mathrm{dp}[i][j] :=$ 作成している DNA 文字列が i 文字であって、その suffix が S の先頭 j 文字と一致している状態の総数
- $\mathrm{dp}[i][j]$ の状態に対し、末尾に文字 c を追加するとき
  - 次の状態において、S の先頭何文字と一致？ → $nj = \mathrm{next}[j][c]$ 文字
  - $nj = K$、つまり S 全体と一致してしまうならば無視
  - そうでないならば、$\mathrm{dp}[i+1][nj]$ に $\mathrm{dp}[i][j]$ を足そう
- 答え → $\sum_{i=0}^{K-1} \mathrm{dp}[N][i]$
- 全体で $O(K^3 + NK)$

禁止文字列 (単一 ver.)

実装は以下の通り

```cpp
#include <string>
#include <iostream>
#include <algorithm>
using namespace std;

const int MAX_N = 10010;
const int MAX_K = 110;
const int MOD   = 10009;

string DNA = "AGCT";
int nxt[MAX_K][4], dp[MAX_N][MAX_K];

int main() {
    int N, K; cin >> N >> K;
    string S; cin >> S;

    // 前処理
    for(int x=0; x<K; x++) {
        string cur_str = S.substr(0, x);
        for(int c=0; c<4; c++) {
            string nxt_str = cur_str + DNA[c];
            while(S.substr(0, nxt_str.length()) != nxt_str) {
                nxt_str = nxt_str.substr(1);
            }
            nxt[x][c] = nxt_str.length();
        }
    }
```

禁止文字列 (単一 ver.)

実装は以下の通り

```
    // DP
    dp[0][0] = 1;
    for(int i=0; i<N; i++) {
        for(int j=0; j<K; j++) {
            for(int c=0; c<4; c++) {
                int nj = nxt[j][c];
                if(nj < K) (dp[i+1][nj] += dp[i][j]) %= MOD;
            }
        }
    }

    int ans = 0;
    for(int i=0; i<K; i++) (ans += dp[N][i]) %= MOD;
    cout << ans << endl;
    return 0;
}
```

禁止文字列 (複数 ver.)

禁止文字列 (複数 ver.)

'A', 'G', 'C', 'T' の 4 種類の文字のみが含まれる文字列を DNA 文字列と称する。元となる DNA 文字列 S と、禁止パターンとなる N 個の DNA 文字列 $P_1, \cdots, P_N$ が与えられるので、S 中の文字を少なくともいくつ変更すれば禁止パターンが全く含まれない文字列にできるか？ (不可能 → -1)

- $1 \leq |S| \leq 10^3$
- $1 \leq N \leq 50$
- $1 \leq |P_i| \leq 20$

出典: 蟻本 P.329

サンプル

入力 1:	`S = "AAAG", P = {"AAA", "AAG"}`	出力 1:	`1`
入力 2:	`S = "TGAATG", P = {"A", "TG"}`	出力 2:	`4`
入力 3:	`S = "AGT", P = {"A", "G", "C", "T"}`	出力 3:	`-1`

禁止文字列 (複数 ver.)

- 実は単一 ver. とほぼ同じ解法で解ける！
- S を前から見て、文字を書き換えていくことを考えよう
- 単一文字列の場合は、パターン文字列の prefix を状態としていたが・・・

禁止文字列 (複数 ver.)

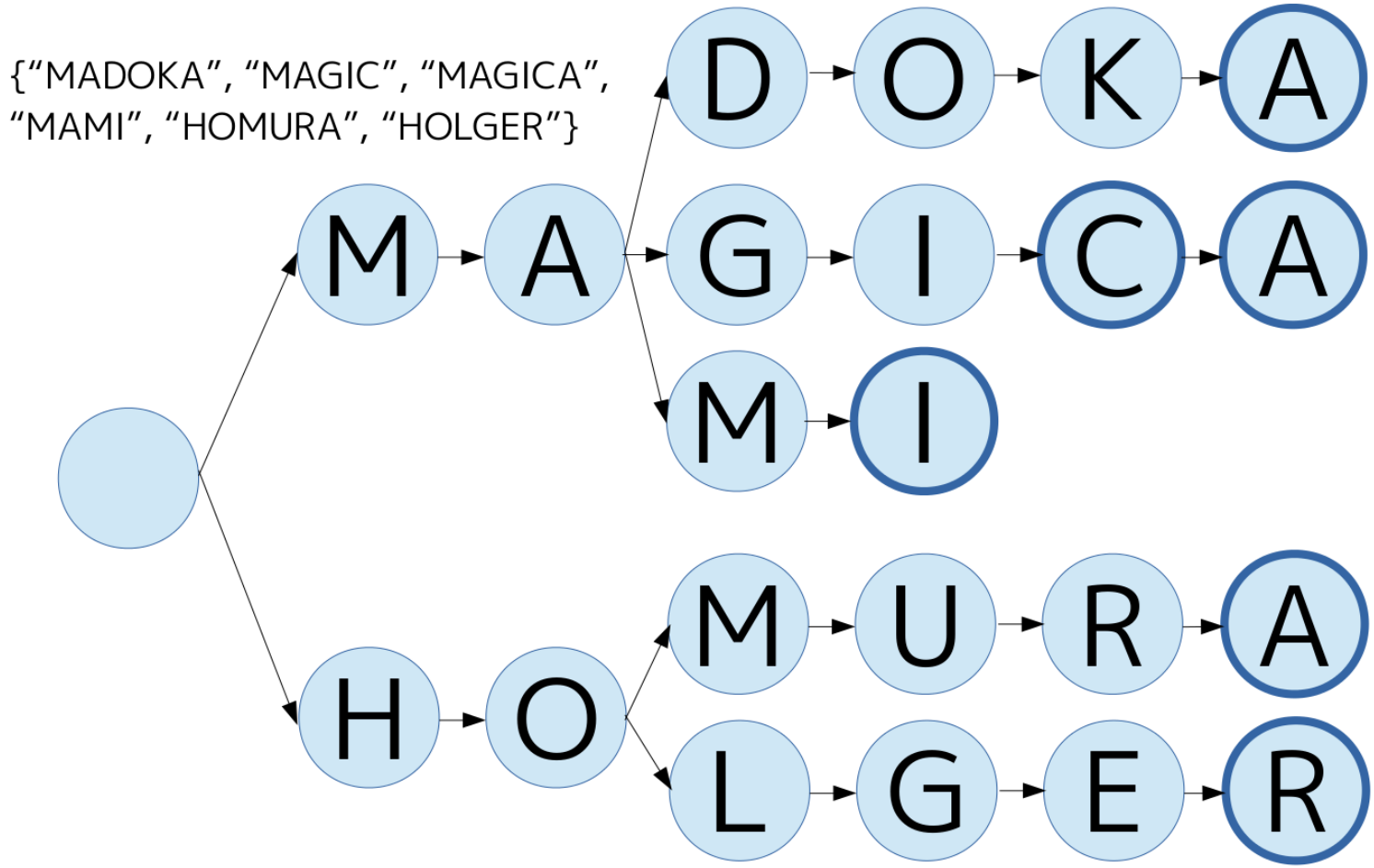
- 実は単一 ver. とほぼ同じ解法で解ける！
- S を前から見て、文字を書き換えていくことを考えよう
- 単一文字列の場合は、パターン文字列の prefix を状態としていたが・・・
  - 複数文字列であれば、各文字列の prefix を列挙して、それらをすべて状態としよう！
    - $P = \{$“AAA”, “AAG”, “T”, “TCA”, “TG”$\}$ の場合、あり得る状態は “A”, “AA”, “AAA”, “AAG”, “T”, “TC”, “TCA”, “TG”
  - この状態間の遷移が把握できれば、先ほどと同様に解ける！

禁止文字列 (複数 ver.)

- 実は単一 ver. とほぼ同じ解法で解ける！
- S を前から見て、文字を書き換えていくことを考えよう
- 単一文字列の場合は、パターン文字列の prefix を状態としていたが・・・
  - 複数文字列であれば、各文字列の prefix を列挙して、それらをすべて状態としよう！
    - $P = \{$“AAA”, “AAG”, “T”, “TCA”, “TG”$\}$ の場合、あり得る状態は “A”, “AA”, “AAA”, “AAG”, “T”, “TC”, “TCA”, “TG”
  - この状態間の遷移が把握できれば、先ほどと同様に解ける！
  - $\mathrm{dp}[i][j] := S$ を i 文字目まで見たとき、その suffix が j 番目の状態と一致しているようにするために必要な操作回数の最小値
- next 配列を得る前処理: $O(N^2L^2 + NL^3 \log(NL))$, DP $O(NL|S|)$
  - L は P_i の長さ上限

禁止文字列 (複数 ver.)

Trie 木の各ノードを状態と解釈できる (つまり Trie 木上で DP している)
各ノードについて、次に何の文字が来たらどのノードに移動するかを前処理



禁止文字列 (複数 ver.)

実装は以下の通り

```cpp
#include <string>
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

const int MAX_STATE = 50 * 20 + 10;
const int MAX_LEN   = 1010;

int nxt[MAX_STATE][4], dp[MAX_LEN][MAX_STATE];

int main() {
    // 入力
    int N, case_num = 0;

    while(cin >> N, N) {
        case_num++;
        vector<string> patterns(N);
        for(int i=0; i<N; i++) cin >> patterns[i];
        string S; cin >> S;

        // 必要な文字列について、prefix をすべて列挙
        vector<string> pfx;
        for(int i=0; i<N; i++) {
            int len = patterns[i].length();
            for(int k=0; k<=len; k++) {
                pfx.push_back(patterns[i].substr(0, k));
            }
        }

        // 重複要素を取り除く
        sort(pfx.begin(), pfx.end());
        pfx.erase(unique(pfx.begin(), pfx.end()), pfx.end());
```

禁止文字列 (複数 ver.)

実装は以下の通り

```
const string DNA = "AGCT";
int M = pfx.size();
vector<int> ng_state(M, false);
for(int i=0; i<M; i++) {
    for(int k=0; k<N; k++) {
        // 元のパターンと一致するような prefix であれば、遷移不可
        int lenA = patterns[k].length(), lenB = pfx[i].length();
        if(lenA <= lenB) {
            if(pfx[i].substr(lenB - lenA, lenA) == patterns[k]) {
                ng_state[i] = true;
            }
        }
    }

    // 1 文字足して、どの状態に行くか判定
    for(int c=0; c<4; c++) {
        string str = pfx[i] + DNA[c]; int k;
        while(1) {
            k = lower_bound(pfx.begin(), pfx.end(), str) - pfx.begin();
            if(k < M && pfx[k] == str) break;
            str = str.substr(1);
        }
        nxt[i][c] = k;
    }
}
```

禁止文字列 (複数 ver.)

実装は以下の通り

```
        // DP
        const int INF = 1 << 28, L = (int)S.length();
        fill(dp[0], dp[1010], INF);
        dp[0][0] = 0;
        for(int i=0; i<L; i++) {
            for(int j=0; j<M; j++) {
                for(int c=0; c<4; c++) {
                    int nj = nxt[j][c];
                    if(ng_state[j] || ng_state[nj]) continue;
                    int cost = (S[i] != DNA[c]);
                    dp[i+1][nj] = min(dp[i+1][nj], dp[i][j] + cost);
                }
            }
        }

        // 答えを出力
        int ans = INF;
        for(int i=0; i<M; i++) if(ng_state[i] == false) ans = min(ans, dp[L][i]);
        cout << "Case " << case_num << ": " << (ans == INF ? -1 : ans) << endl;
    }
    return 0;
}
```

Genetic engineering

'A', 'G', 'C', 'T' の 4 種類の文字のみが含まれる文字列を DNA 文字列と称する。M 個の DNA 文字列 $P = \{p_1, \cdots, p_M\}$ が与えられるので、このパターンを (区間被りありで) 連結させて N 文字の文字列を作りたい。生成可能な N 文字の文字列は何通りあるか？

- $1 \leq N \leq 10^3$
- $1 \leq M \leq 10$
- $1 \leq |p_i| \leq 10$, パターン文字列は相異なる

出典: Yandex. Algorithm 2011 Round 2: C ▸ Link

サンプル

入力 1: `N = 2, M = 1, P = {"A"}` 出力 1: `1`

入力 2: `N = 6, M = 2, P = {"CAT", "TACT"}` 出力 2: `2`

(入力 2 は、"CATCAT", "CATACT" の 2 通り)

Genetic engineering

- パターン文字列の長さが高々 10 なので、後ろ 10 文字だけ覚えて DP できるか？
  - これは無理 (DP の状態数だけで 10^9 個くらいある)
- Trie 木上の DP に落とし込めないか？

Genetic engineering

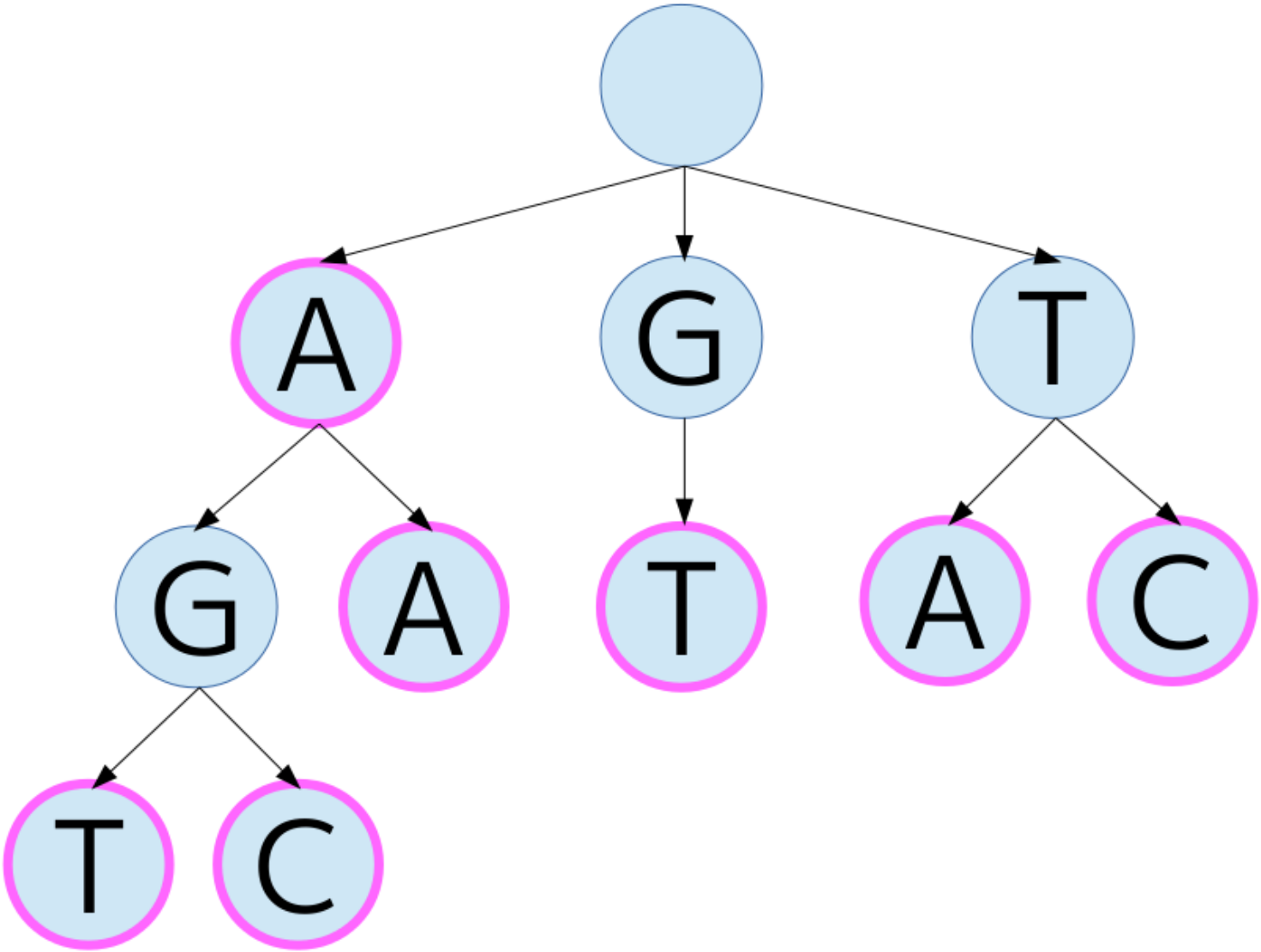
- パターン文字列の長さが高々 10 なので、後ろ 10 文字だけ覚えて DP できるか？
  - これは無理 (DP の状態数だけで 10^9 個くらいある)
- Trie 木上の DP に落とし込めないか？
  - 結論から言うと、できる
  - 状態はどのように持つか？

Genetic engineering

- パターン文字列の長さが高々 10 なので、後ろ 10 文字だけ覚えて DP できるか？
  - これは無理 (DP の状態数だけで 10^9 個くらいある)
- Trie 木上の DP に落とし込めないか？
  - 結論から言うと、できる
  - 状態はどのように持つか？
  - $\mathrm{dp}[i][j][k] := i$ 文字目まで文字列を作成・Trie 木上の k 番目の状態・パターン文字列のいずれかと最後に一致したのが j 文字前 であるような状態の総数 とする
  - 次のページで詳しく見ていこう

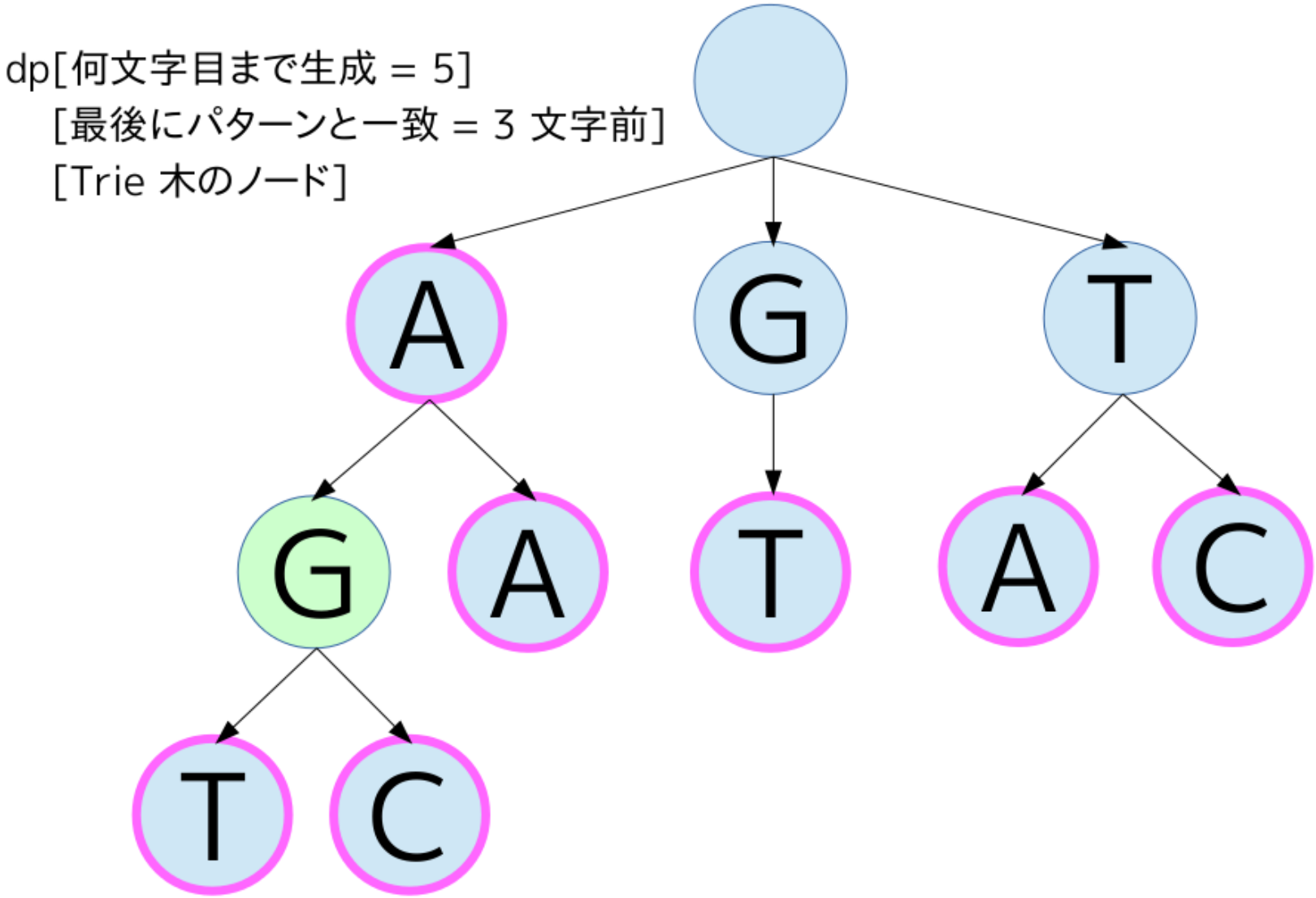
Genetic engineering

禁止文字列のときと同様に、パターン文字列から Trie 木を作る



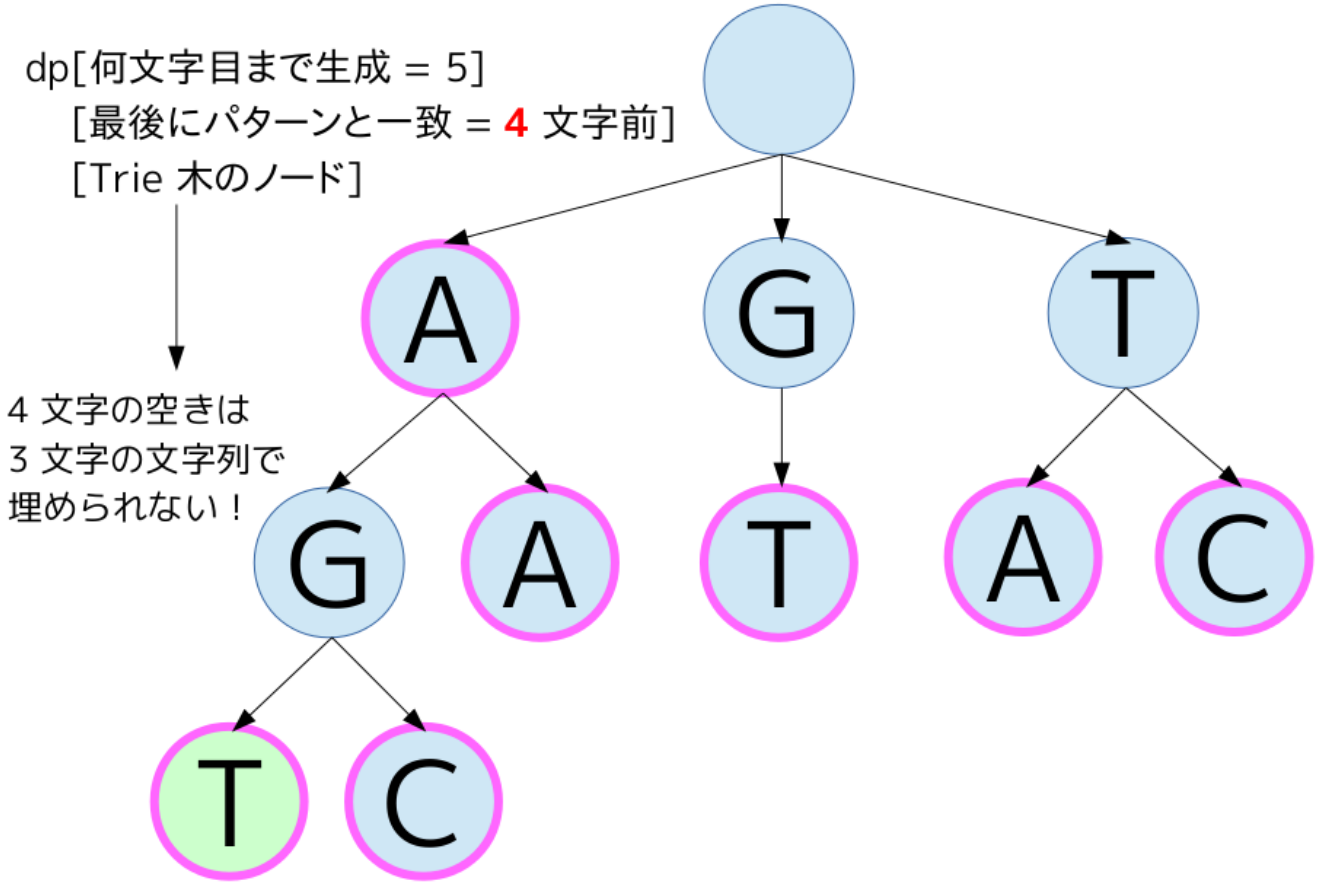
Genetic engineering

DP が以下の状態であったとする



Genetic engineering

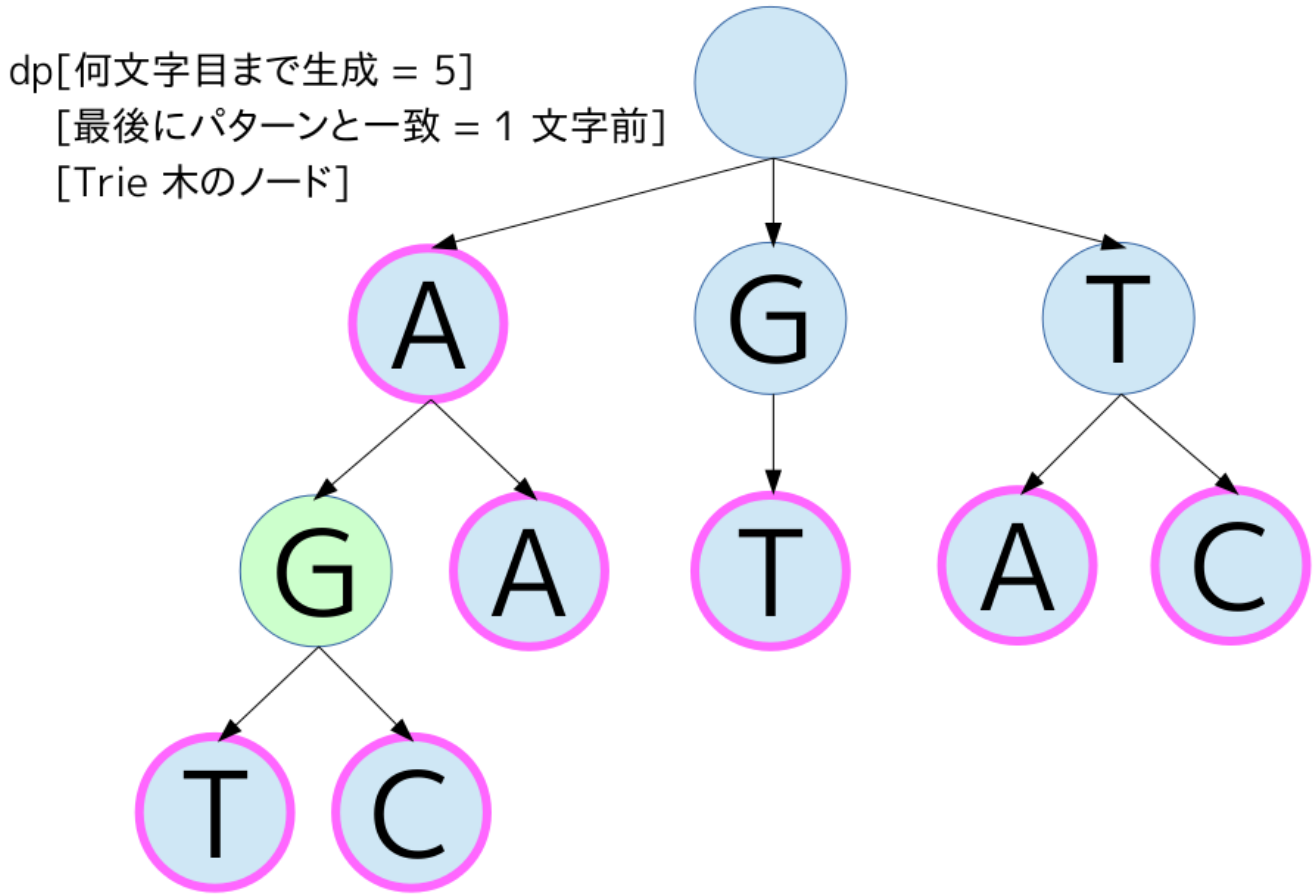
次の文字として ‘T’ を選択すると、以下の状態になる
最後にパターンと一致したのは 4 文字前だが、3 文字のパターン文字列ではこの溝を埋められないので、この状態は捨てられる



“A”, “AGT”, “AGC”, “AA”, “GT”, “TA”, “TC”

Genetic engineering

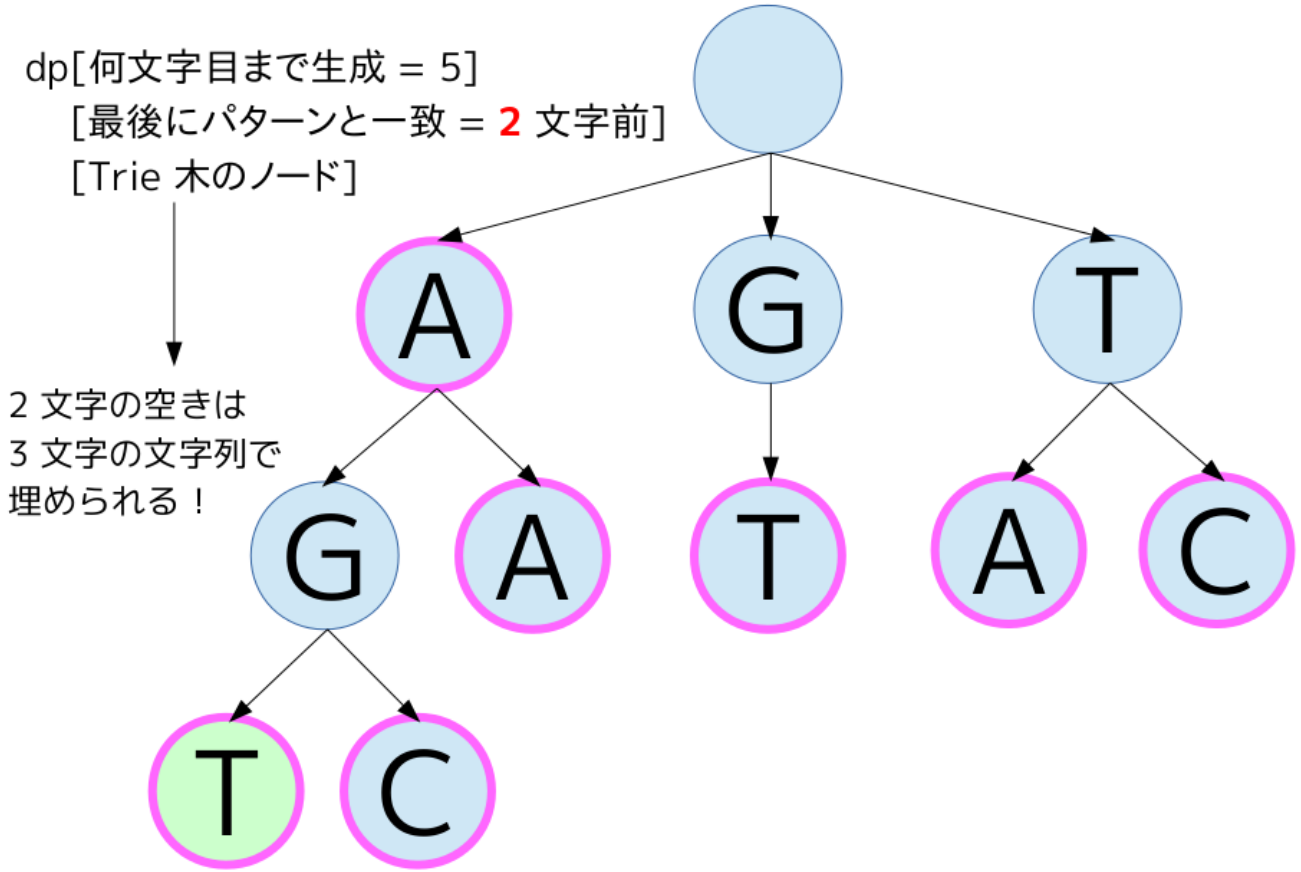
最後にパターンと一致したのが、もう少し近かったらどうなるか？



“A”, “AGT”, “AGC”, “AA”, “GT”, “TA”, “TC”

Genetic engineering

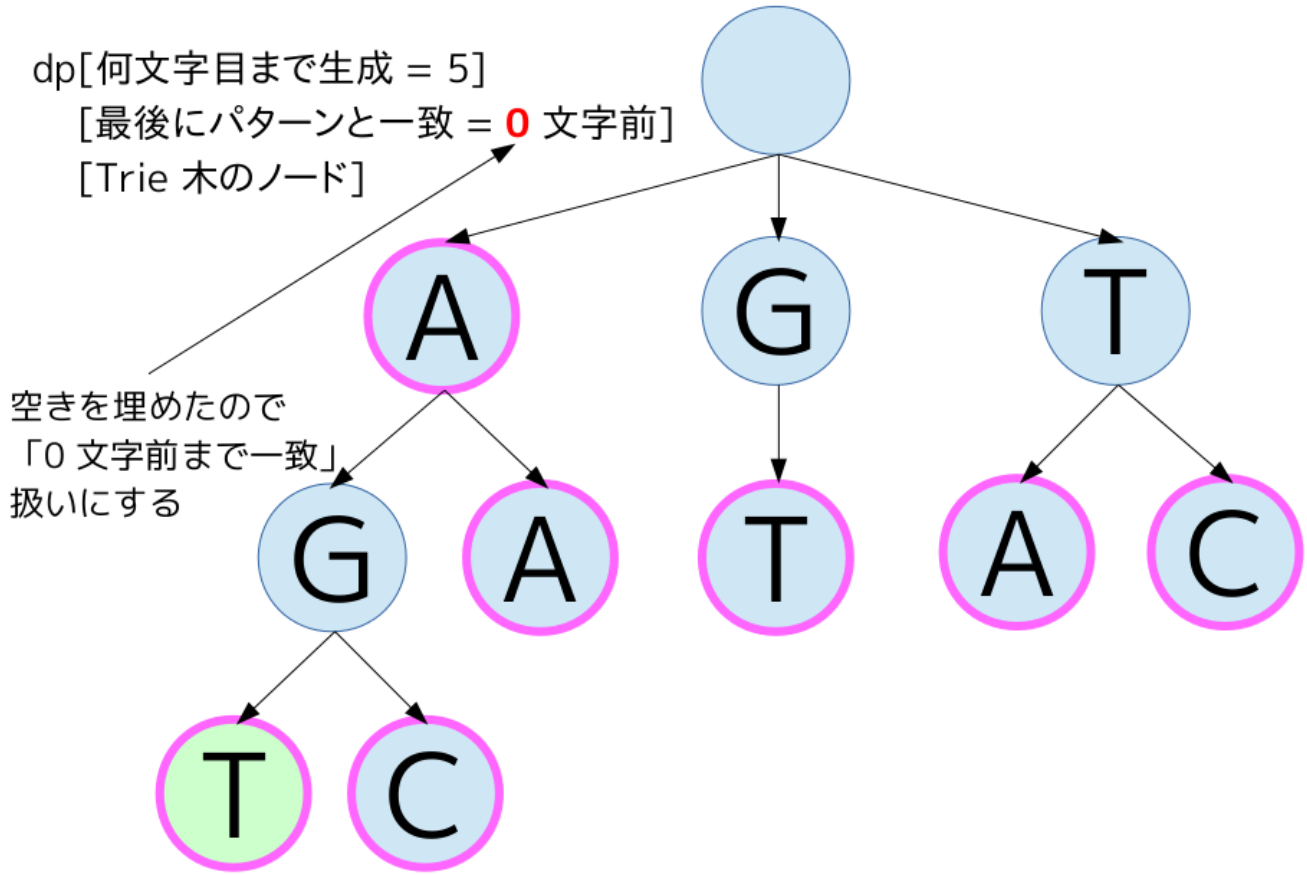
この場合は、パターン文字列 “AGT” によって溝を埋められる！



“A”, “AGT”, “AGC”, “AA”, “GT”, “TA”, “TC”

Genetic engineering

今の遷移によってパターンと一致したため、最後にパターンと一致したのは 0 文字前に更新される



“A”, “AGT”, “AGC”, “AA”, “GT”, “TA”, “TC”

Genetic engineering

- Trie 木内の状態数最大は、だいたいパターン文字列長の総和 $L = \sum_i |p_i|$ (高々 100)
- 最後にパターンと一致するインデックスは、高々「最も長いパターン文字列長」前まで考慮すれば良い
  - それ以上遠くまで遡らなければいけない場合、溝を埋めるのは不可能
  - $X = \max_i |p_i|$ (高々 10)
- DP の状態数は $O(NLX)$ で、遷移 $O(1)$ なので、DP の計算量は $O(NLX)$ → 間に合う！

Genetic engineering

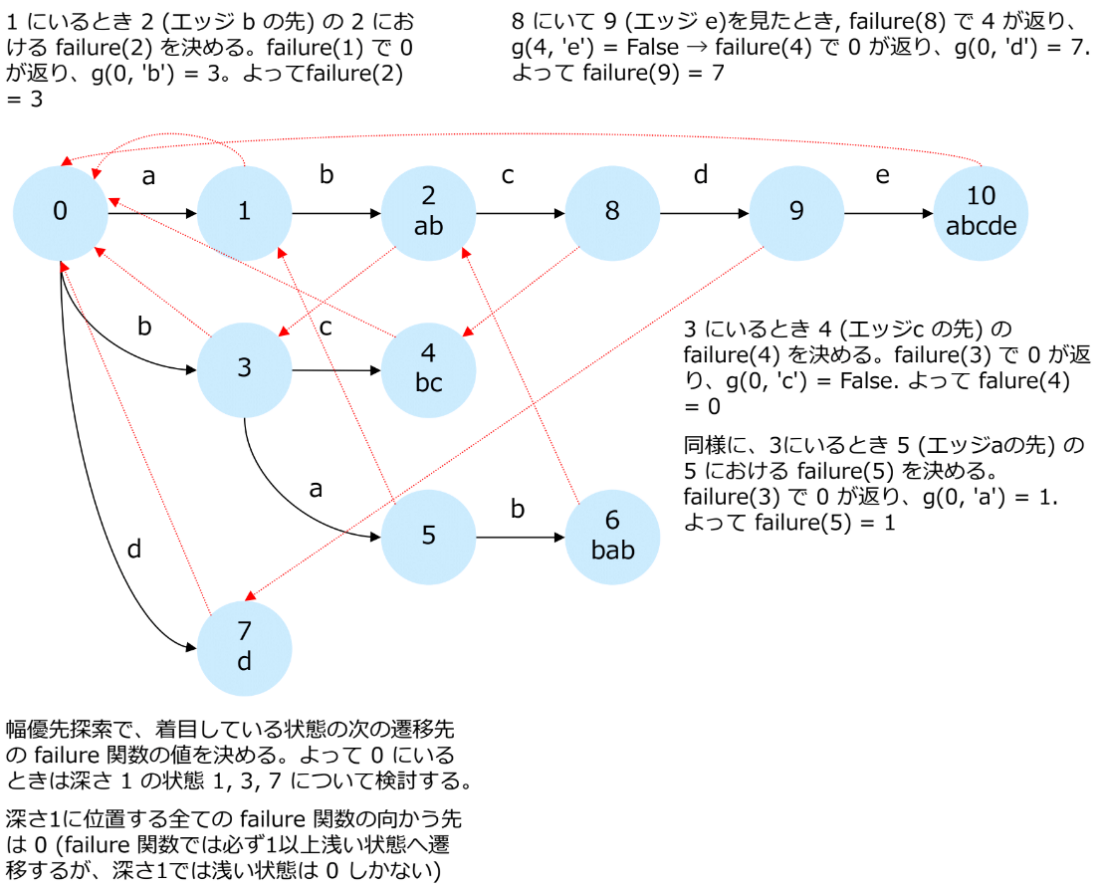
実装 (長いので DP 部分のみ紹介)

```
dp[0][0][0] = 1;
for(int i=0; i<N; i++) {
    for(int j=0; j<10; j++) {
        for(int k=0; k<P; k++) {
            for(int c=0; c<4; c++) {
                // next_state := 状態 k の文字列の末尾に文字 c を追加すると
                //               次の状態はどれになるか？

                // max_len    := ある状態において、そこを終端とするような
                //               パターン文字列の中で長さが最大のものは何か？
                //               (-1 で初期化、終端存在するなら更新)
                int nk = next_state[k][c], ma = max_len[nk];
                if(j < ma) {
                    (dp[i+1][0  ][nk] += dp[i][j][k]) %= MOD; // 一致
                }
                else if(j + 1 < 10) {
                    (dp[i+1][j+1][nk] += dp[i][j][k]) %= MOD; // 一致しない
                }
            }
        }
    }
}
```

備考

- Trie 木において「次にどのノードに移動するか」の前処理は、実は線形時間で構築可能 (Aho-Corasick 法)



画像出典: Aho Corasick 法 - naoya のはてなダイアリー ▸ Link

練習問題

- OnlySanta (TopCoder SRM 727 Div1 Easy) Link
  - ManageSubsequences と同じ回の Div1 Easy
  - 解法も非常に似ています
- 有向グラフ (AtCoder Regular Contest 030 C) Link
  - 典型よりの文字列 DP
  - 前回 monkukui 君が紹介してくれたグラフ分野の復習にもなります
- たんごたくさん (天下一プログラマーコンテスト 2016 本戦 C) Link
  - Trie 木に慣れるのに最適な問題だと思います
- 宝くじ (東京大学プログラミングコンテスト 2014 E) Link
  - これも Trie 木
- 文字列 (Typical DP Contest O) Link
  - まだ解いてない・・・