Team notebook

Gabriel Gutierrez Tamayo

November 7, 2024

Contents

1 New implementations

1.1 edmonds karp

```
#include<bits/stdc++.h>
using namespace std;
#define inf 1e9

const int maxNodes = 500;
int residual[maxNodes][maxNodes];

int FindAugmentingPath(int n, int s, int t, bool unit){
	queue<int> Q;
	vector<int> S(n,0), P(n,-1), flow(n, 0);
	Q.push(s);
	S[s] = 1;
	flow[s] = inf;
	while(!Q.empty()){
		int x = Q.front(); Q.pop();
		for(int y=0; y<n; y++){
			if(residual[x][y] > 0 && S[y] == 0){
				Q.push(y);
				S[y] = 1;
				P[y] = x;
				flow[y] = min(flow[x], residual[x][y]);
			}
		}
	}
	if(flow[t] > 0){
		int f = unit ? 1 : flow[t];
		int x = t;
		while(x != s){
			int p = P[x];
			residual[p][x] -= f;
			residual[x][p] += f;
			x = p;
		}
		return f;
	}
	return 0;
}

int MaxFlow(int n, int s, int t){
	int flow = 0;
	for(int f; f = FindAugmentingPath(n,s,t,false); )
	flow += f;
	return flow;
}
```

1.2 fast matrix exponentiation

```
#include<bits/stdc++.h>
using namespace std;

// Allocates memory to create a square matrix nxn and returns it.
long long** MakeSquareMatrix(int n){
	long long** ans = (long long**)malloc(sizeof(long long*) * n);
	for(int i=0; i<n; i++)
	ans[i] = (long long*)malloc(sizeof(long long) * n);
	return ans;
}

// Frees the memory allocated to a square matrix.
void ClearSquareMatrix(long long** matrix, int n){
	for(int i=0; i<n; i++)
	free(matrix[i]);
	free(matrix);
}

// Multiply two square matrices nxn in O(n^3).
long long** MatrixMultiplication(long long** a, long long** b, int n,
    long long mod){
	long long** c = MakeSquareMatrix(n);
	for(int i=0; i<n; i++){
		for(int j=0; j<n; j++){
			long long sum = 0;
			for(int l=0; l<n; l++)
			sum = (sum + a[i][l] * b[l][j]) % mod;
			c[i][j] = sum;
		}
	}
	return c;
}

// Calculates f(k) in O((n^3) * log(k)) using fast matrix exponentiation.
//
// Where f(k) = c_k-1 * f_k-1 + c_k-2 * f_k-2 + c_k-3 * f_k-3 + ...
//
// Given:
//   c_0, c_1, c_2, ..., c_n-1
//   f_0, f_1, f_2, ..., f_n-1
//
// Initial matrix:
//
//        const  ...  f_k-4  f_k-3  f_k-2  f_k-1
// const    1    ...    0      0      0      0
// ...     ...   ...   ...    ...    ...    ...
// f_k-3    0    ...    0      1      0      0
// f_k-2    0    ...    0      0      1      0
// f_k-1    0    ...    0      0      0      1
// f_k      1    ...  c_k-4  c_k-3  c_k-2  c_k-1
long long SolveRecurrence(vector<long long>& c, vector<long long>& f,
    long long cst, long long k, long long mod){
	if(k < f.size())
	return f[k];
	k -= f.size() - 1;

	int n = f.size() + 1;

	// Build initial matrix.
	long long** mpw2 = MakeSquareMatrix(n);
	for(int i=0; i<n; i++){
		for(int j=0; j<n; j++)
		mpw2[i][j] = 0;
	}
	mpw2[0][0] = mpw2[n-1][0] = 1;
	for(int i=1; i<n-1; i++)
	mpw2[i][i+1] = 1;
	for(int j=1; j<n; j++)
	mpw2[n-1][j] = c[j-1];

	// Build answer matrix, initially equal to the identity matrix.
	long long** matrix = MakeSquareMatrix(n);
	for(int i=0; i<n; i++){
		for(int j=0; j<n; j++)
		matrix[i][j] = 0;
		matrix[i][i] = 1;
	}

	// Calculate f(k) using fast matrix exponentiation.
	for(int i=0; k > 0; i++){
		if((k & (1ll<<i)) > 0){
			long long** mult = MatrixMultiplication(matrix,
			    mpw2, n, mod);
			ClearSquareMatrix(matrix, n);
			matrix = mult;
			k ^= (1ll<<i);
		}
```

```
            long long** mult = MatrixMultiplication(mpw2, mpw2, n,
                mod);
            ClearSquareMatrix(mpw2, n);
            mpw2 = mult;
        }
        long long ans = (matrix[n-1][0] * cst) % mod;
        for(int j=1; j<n; j++)
        ans = (ans + matrix[n-1][j] * f[j-1]) % mod;

        ClearSquareMatrix(matrix, n);
        ClearSquareMatrix(mpw2, n);

        return ans;
}
```

1.3 fenwick tree

```
#include<bits/stdc++.h>
using namespace std;

struct FenwickTreeNode {
        long long value;
        FenwickTreeNode() : value(0) {}
        FenwickTreeNode(long long _value) : value(_value) {}
        void SetNeutral(){ value = 0; } // Modify according to the
            operation (e.g. 0 for sum, 1 for multiplication, ...)
};

struct FenwickTree{
        int size;
        vector<FenwickTreeNode> nodes, blocks;

        FenwickTreeNode combine(FenwickTreeNode a, FenwickTreeNode b);
        FenwickTreeNode decombine(FenwickTreeNode a, FenwickTreeNode b);
        FenwickTreeNode calculatePrefix(int pos);
        void Build(int size);
        void Update(int pos, FenwickTreeNode newValue);
        FenwickTreeNode Query(int l, int r);
};

FenwickTreeNode FenwickTree::combine(FenwickTreeNode a, FenwickTreeNode
    b){
        // Modify the combination logic here.
        return FenwickTreeNode(a.value + b.value);
}

FenwickTreeNode FenwickTree::decombine(FenwickTreeNode a, FenwickTreeNode
    b){
        // Modify the decombination logic here.
        return FenwickTreeNode(a.value - b.value);
}

FenwickTreeNode FenwickTree::calculatePrefix(int pos){
        FenwickTreeNode prefix = FenwickTreeNode();
        prefix.SetNeutral();
        while(pos > 0){
                prefix = combine(prefix, blocks[pos]);
                pos -= pos & -pos;
        }
        return prefix;
}

void FenwickTree::Build(int size) {
        this->size = size;
        nodes.clear(); blocks.clear();
        nodes.resize(size+1);
        blocks.resize(size+1);
        for(int i=0; i<=size; i++){
                nodes[i].SetNeutral();
                blocks[i].SetNeutral();
        }
}

void FenwickTree::Update(int pos, FenwickTreeNode newValue){
        FenwickTreeNode oldValue = nodes[pos];
        nodes[pos] = newValue;
        while(pos <= size){
                blocks[pos] = decombine(blocks[pos], oldValue);
                blocks[pos] = combine(blocks[pos], newValue);
                pos += pos & -pos;
        }
}

// 1 <= l <= r <= size
FenwickTreeNode FenwickTree::Query(int l, int r){
        return decombine(calculatePrefix(r), calculatePrefix(l-1));
}
```

1.4 lowest common ancestor

```
#include<bits/stdc++.h>
using namespace std;

const int maxN = 200000;
const int maxSTN = 2 * maxN - 1;
const int maxLogSTN = __lg(maxSTN);
vector<int> tree[maxN];
vector<int> P(maxN), depth(maxN), firstoccur(maxN);
vector<int> seq;
int stmindepth[maxSTN][maxLogSTN+1];

void DfsForLcaVisit(vector<int>& S, int p, int x){
	S[x] = 1;
	P[x] = p;
	depth[x] = (p == -1) ? 0 : depth[p] + 1;
	firstoccur[x] = seq.size();
	seq.push_back(x);
	for(int i=0; i<tree[x].size(); i++){
		int y = tree[x][i];
		if(S[y] == 0){
			DfsForLcaVisit(S, x, y);
			seq.push_back(x);
		}
	}
}

void DfsForLca(int n, int root){
	vector<int> S(n, 0);
	DfsForLcaVisit(S, -1, root);
}

void BuildLcaSolver(int n, int root){
	DfsForLca(n, root);
	int lg2 = __lg(seq.size());
	for(int i=0; i<seq.size(); i++)
	stmindepth[i][0] = seq[i];
	for(int j=1,pw2=1; j<=lg2; j++,pw2<<=1){
		for(int i=0; i<seq.size(); i++){
			stmindepth[i][j] = stmindepth[i][j-1];
			if(i+pw2 < seq.size()){
				if(depth[stmindepth[i+pw2][j-1]] <
				    depth[stmindepth[i][j-1]])
				stmindepth[i][j] = stmindepth[i+pw2][j-1];
			}
		}
	}
}

// Find LCA in O(1).
int LCA(int x, int y){
	int l, r, lg2, pw2, op1, op2;
	l = min(firstoccur[x], firstoccur[y]);
	r = max(firstoccur[x], firstoccur[y]);
	lg2 = __lg(r+1-l); pw2 = 1<<lg2;
	op1 = stmindepth[l][lg2];
	op2 = stmindepth[r+1-pw2][lg2];
	return (depth[op1] < depth[op2]) ? op1 : op2;
}
```

1.5 max flow min cost

```
#include<bits/stdc++.h>
using namespace std;
#define inf 1e9
typedef pair<int,int> pii;

const int maxNodes = 500;
int residual[maxNodes][maxNodes];
int cost[maxNodes][maxNodes];

// Return the minimum cost between the all augmenting paths.
// use Bellman Ford algorithm.
pii FindAugmentingPath(int n, int s, int t, bool unit){
	queue<int> Q;
	vector<int> P(n,-1), dist(n,inf), flow(n,0);
	Q.push(s);
	dist[s] = 0;
	flow[s] = inf;
	while(!Q.empty()){
		int x = Q.front(); Q.pop();
		for(int y=0; y<n; y++){
			if(residual[x][y] > 0 && dist[x] + cost[x][y] <
			    dist[y]){
				Q.push(y);
				P[y] = x;
				dist[y] = dist[x] + cost[x][y];
```

```
                        flow[y] = min(flow[x], residual[x][y]);

                }
        }
    }

    if(dist[t] != inf){
        int f = unit ? 1 : flow[t];
        int x = t;
        while(x != s){
            int p = P[x];
            residual[p][x] -= f;
            residual[x][p] += f;
            x = p;
        }
        return pii(f, dist[t]);
    }
    return pii(0,0);
}

pii MaxFlowMinCost(int n, int s, int t){
    int flow = 0, cost = 0;
    while(true){
        pii p = FindAugmentingPath(n,s,t,false);
        flow += p.first;
        cost += p.first * p.second;
        if(p.first == 0)
        break;
    }
    return pii(flow, cost);
}
```

1.6 suffix array

```
#include<bits/stdc++.h>
using namespace std;

struct SuffixArray {
    int alpsize, size, levels;
    vector<int> text, sortedIds, lcp;

    vector<int> sortStringsFromLevel0();
    vector<int> sortStringsFromLevelK(int k, const vector<int>& rank);
    void Build(vector<int> text, int alpsize);
    void BuildLcpTable();
};

// Sort the suffixes for its first character.
// Sorting algorithm: Counting sort O(n).
vector<int> SuffixArray::sortStringsFromLevel0(){
    // Sort ids by first character.
    vector<int> v[alpsize];
    for(int i=0; i<size; i++)
    v[text[i]].push_back(i);
    for(int i=0,pos=0; i<alpsize; i++){
        for(int j=0; j<v[i].size(); j++)
        sortedIds[pos++] = v[i][j];
    }

    // Calculates the rank for each id.
    vector<int> rank2(size);
    rank2[sortedIds[0]] = 1;
    for(int i=1; i<size; i++){
        int id = sortedIds[i], previd = sortedIds[i-1];
        if(text[id] == text[previd]) rank2[id] = rank2[previd];
        else rank2[id] = rank2[previd] + 1;
    }
    return rank2;
}

// Sort the suffixes by its first 2^k characters.
// The sorting is on the pair<id_rank, next_id_rank> of previous level
// and run in O(n), taking advantage that suffixes are already sorted by
// id_rank (and therefore also by next_id_rank).
//
// Sorting algorithm:
//   Iterates through the suffixes in ascending order of next_id_rank and
    places
//   them in the next available position for the block with the same
    id_rank.
//   Complexity: O(n)
vector<int> SuffixArray::sortStringsFromLevelK(int k, const vector<int>&
    rank){
    // Sort ids by first 2^k characters.
    int pw2 = 1<<(k-1);
    vector<int> last(size+1, 0), auxSortedIds(sortedIds);
    for(int i=0; i<size; i++)
    last[rank[sortedIds[i]]]++;
```

```
	for(int i=1; i<last.size(); i++)
	last[i] += last[i-1];

	for(int i=-pw2; i<size; i++){
		int nextId = (i < 0) ? size + pw2 + i : auxSortedIds[i];
		int id = nextId - pw2;
		if(id >= 0)
		sortedIds[last[rank[id]-1]++] = id;
	}

	// Calculates the rank for each id.
	vector<int> rank2(size);
	rank2[sortedIds[0]] = 1;
	for(int i=1; i<size; i++){
		int id = sortedIds[i], previd = sortedIds[i-1];
		int c11, c12, c21, c22;
		c11 = rank[previd];
		c21 = rank[id];
		c12 = (previd+pw2 >= size) ? 0 : rank[previd+pw2];
		c22 = (id+pw2 >= size) ? 0 : rank[id+pw2];

		rank2[id] = rank2[previd];
		if(c21 > c11 || c22 > c12)
		rank2[id]++;
	}
	return rank2;
}

// Builds the suffix array using standard algorithm in O(n * log n).
void SuffixArray::Build(vector<int> text, int alpsize){
	this->text = text;
	this->alpsize = alpsize;
	size = text.size();
	levels = log2(size-1) + 2;

	sortedIds.resize(size);

	vector<int> rank = sortStringsFromLevel0();
	for(int k=1; k<levels; k++)
	rank = sortStringsFromLevelK(k, rank);
}

// Builds the Longest Common Prefix table in O(n).
void SuffixArray::BuildLcpTable(){
	vector<int> position(size);
	for(int i=0; i<size; i++)
	position[sortedIds[i]] = i;

	lcp.resize(size);
	for(int id=0; id<size; id++){
		lcp[position[id]] = 0;
		if(position[id] != 0){
			int previd = sortedIds[position[id]-1];
			int counter = (id == 0) ? 0 : max(0,
			    lcp[position[id-1]] - 1);
			while(max(id,previd)+counter < size &&
			    text[id+counter] == text[previd+counter])
			counter++;
			lcp[position[id]] = counter;
		}
	}
}
```

2 Old implementations

2.1 extended euclidean

```
#include<bits/stdc++.h>
using namespace std;

// ax + by = mcd
// datos de entrada: a, b
// datos de salida: x, y, mcd
void extendedEuclid(long long a, long long b, long long &x, long long &y,
    long long &mcd){
	if(b == 0){
		x = 1;
		y = 0;
		mcd = a;
	}
	else{
		extendedEuclid(b, a % b, x, y, mcd);
		x = y;
		y = (mcd - a*x) / b;
	}
}
```

```
// retorna (a/b) mod m
// a*(b^(-1)) mod m
long long moduloDivision(long long a, long long b, long long modulo){
	long long x, y, mcd;
	extendedEuclid(b, modulo, x, y, mcd);
	if(x < 0)
	x += modulo;
	return (x * (a / mcd)) % modulo;
}
```

2.2 fft

```
#include<bits/stdc++.h>
using namespace std;
#define endl '\n'
typedef complex<double> complejo;
const double pi = 3.14159265358979;

vector<complejo> fft(vector<complejo> &a, bool inverse=false){
	int n = a.size();
	if(n == 1)
	return a;

	complejo wn, w;
	if(inverse)
	wn = complejo(cos(2*pi/n), -sin(2*pi/n));
	else
	wn = complejo(cos(2*pi/n), sin(2*pi/n));
	w = 1.0;

	vector<complejo> aEven(n/2), aOdd(n/2);
	vector<complejo> yEven, yOdd, y(n);

	for(int i=0,j1=0,j2=1; i<n/2; i++,j1+=2,j2+=2){
		aEven[i] = a[j1];
		aOdd[i] = a[j2];
	}

	yEven = fft(aEven, inverse);
	yOdd = fft(aOdd, inverse);

	for(int i=0,j=n/2; i<n/2; i++,j++){
		y[i] = yEven[i] + w*yOdd[i];
		y[j] = yEven[i] - w*yOdd[i];
		w *= wn;
	}
	if(inverse){
		for(int i=0; i<n; i++){ y[i] /= 2.0; }
	}

	return y;
}

void convolution(vector<long long> V1, vector<long long> V2, vector<long
    long> &V3){
	vector<complejo> A, B, C, C1, C2, Z;

	int n,pot;
	n = 2 * max(V1.size(), V2.size());
	pot = pow(2, int(log2(n)));
	if(n > pot)
	n = 2*pot;

	A.resize(n,0);  B.resize(n,0);
	for(int i=0; i<V1.size(); i++){ A[i] = V1[i]; }
	for(int i=0,j=V2.size()-1; i<V2.size(); i++,j--){ B[i] = V2[j]; }
	C1 = fft(A);
	C2 = fft(B);
	C.resize(n);
	for(int i=0; i<n; i++){ C[i] = C1[i] * C2[i]; }
	Z = fft(C, true);

	V3.clear();
	for(int i=0; i<n; i++){
		double x = round(real(Z[i]));
		V3.push_back(x);
	}
}

int main(){
	ios_base::sync_with_stdio(0);cin.tie(NULL);
	int n1,n2;
	vector<int> alp = {'A', 'C', 'T', 'G'};
	string cad1, cad2;
	cin>>cad1>>cad2;
	n1 = cad1.size();
	n2 = cad2.size();
```

```
	vector<int> acu(n1,0);
	vector<long long> V1, V2, V3;
	V1.resize(n1); V2.resize(n2);
	for(int l=0; l<alp.size(); l++){
		for(int i=0; i<n1; i++){
			if(cad1[i] == alp[l]) V1[i] = 1;
			else V1[i] = 0;
		}
		for(int i=0; i<n2; i++){
			if(cad2[i] == alp[l]) V2[i] = 1;
			else V2[i] = 0;
		}
		convolution(V1, V2, V3);

		for(int i=n2-1; i<=n1-1; i++)
		acu[i] += V3[i];
	}

	int mayor = 0;
	for(int i=n2-1; i<n1; i++)
	mayor = max(mayor, acu[i]);

	int ans = n2 - mayor;
	cout<<ans<<endl;
}
```

2.3 knuth morris pratt

```
#include <bits/stdc++.h>
using namespace std;

vector<int> prefixFunctionKMP(string cadena){
	vector<int> prefixFunction(cadena.size());
	prefixFunction[0] = 0;
	for(int i=1,j=0; i<cadena.size(); i++){
		while(j > 0 && cadena[i] != cadena[j]){
			j = prefixFunction[j-1];
		}
		if(cadena[i] == cadena[j])
		j++;
		prefixFunction[i] = j;
	}
	return prefixFunction;
}
```

2.4 segment tree

```
#include<bits/stdc++.h>
using namespace std;

struct SegmentTreeNode{

	int null, valor;

	SegmentTreeNode(){ null = 1; }

	SegmentTreeNode(const SegmentTreeNode& s){
		null = s.null;
		valor = s.valor;
	}

	SegmentTreeNode(int valor){
		null = 0;
		this->valor = valor;
	}

};

struct SegmentTree{
	int n;
	vector<SegmentTreeNode> vect;
	vector<SegmentTreeNode> T;

	SegmentTreeNode funcion(SegmentTreeNode a, SegmentTreeNode b){
		if(a.null == 1 && b.null == 1)
		return SegmentTreeNode();
		if(a.null == 1)
		return b;
		if(b.null == 1)
		return a;
		return SegmentTreeNode(min(a.valor, b.valor));
	}

	void build(vector<SegmentTreeNode> V){
		vect.clear();
		vect.resize(V.size());
```

```
        for(int i=0; i<vect.size(); i++){
                vect[i] = V[i];
        }
        n = 4 * V.size();
        T.clear();
        T.resize(n, SegmentTreeNode());
        build2(1,0,vect.size()-1);
}

void build2(int pos, int i, int j){
        if(i==j)
        T[pos] = SegmentTreeNode(vect[i]);
        else{
                build2(2*pos, i, (i+j)/2);
                build2(2*pos+1, (i+j+2)/2, j);
                T[pos] = funcion(T[2*pos], T[2*pos+1]);
        }
}

void update(int id, SegmentTreeNode valor){
        vect[id] = valor;
        update2(1,0,vect.size()-1,id,valor);
}

void update2(int pos, int i, int j, int id, SegmentTreeNode valor){
        if(i==j){
                if(id==i)
                        T[pos] = valor;
                }
                else{
                        if(i<=id && id<=j){
                                update2(2*pos, i,(i+j)/2, id, valor);
                                update2(2*pos+1, (i+j+2)/2,j, id, valor);
                                T[pos] = funcion(T[2*pos],T[2*pos+1]);
                        }
                }
        }

        SegmentTreeNode query(int i, int j){
                return query2(1, 0,vect.size()-1, i,j);
        }

        SegmentTreeNode query2(int pos, int x, int y, int i, int j){
                if(j<x || i>y)
                return SegmentTreeNode();
                else if(i<=x && y<=j)
                return T[pos];
                else
                return funcion(query2(2*pos, x,(x+y)/2, i,j),
                    query2(2*pos+1, (x+y+2)/2,y, i,j));
        }
};
```