ĐỘ PHỨC TẠP MERGE SORT

+ Sắp xếp mảng a với n phần tử

$T(n) = t(k) + t(n - k) + cn$

+ t(k): thời gian sắp xếp mảng với k phần tử

+ t(n – k): thời gian sắp xếp mảng với n – k phần tử

+ cn: tổng thời gian sắp xếp lại mảng a

Ta sẽ sắp xếp 2 mảng:

+ Mảng 1: có n / 2 phần tử

+ Mảng 2: có n / 2 phần tử

⇨ $T(n) = t(n / 2) + t(n / 2) + cn$
$= 2t(n/2) + cn$
$= 2[2t(n / 4) + c*(n/2)] + cn$
$= 2^2t(n / 4) + 2cn$
$= 2^2[2t(n/8) + c*(n/4)] + 2cn$
$= 2^3t(n/8) + 3cn$
…
$= 2^kt(n/2^k) + kcn$ (1)

(1) Sẽ dừng khi đạt được t(1)

Ta sẽ đạt được t(1) ⇔ $n = 2^k$ <=> $\log_2 n = k$

⇨ $T(n) = 2^{\log_2 n}t(1) + cn * (\log_2 n)$

<=> $T(n) = nt(1) + cn*\log_2 n$

$= A + B$

+(A): độ phức tạp O(n)
+(B): độ phức tạp $O(n\log_2 n)$

⇨ Chi phí cho trường hợp tốt nhất là $O(n\log_2 n)$

Độ phức tạp QuickSort

+ Độ phức tạp của thuật toán Quick Sort phụ thuộc vào cách chọn Pivot.

1. Trường hợp tốt nhất của Quick Sort – Pivot nằm ở vị trí medium

+ Ở đây tôi lấy mốc là mid là phần tử nằm ở giữa => Lúc này bài toán chứng minh này trở thành chứng minh độ phức tạp của ***MergeSort***

2. Trường hợp xấu nhất của Quick Sort – Pivot nằm ở vị trí min, max

+ Sắp xếp mảng a với n phần tử

$T(n) = t(k) + t(n - k) + cn$

+ t(k): thời gian sắp xếp mảng với k phần tử

+ t(n - k): thời gian sắp xếp mảng với n - k phần tử

+ cn: tổng thời gian sắp xếp lại mảng a

Trường hợp xấu nhất: pivot nằm ở vị trí min, max

Ta sẽ sắp xếp 2 mảng:

+ Mảng 1: có 1 phần tử

+ Mảng 2: có n - 1 phần tử

⇨ $T(n) = t(n - 1) + t(1) + cn$

$= [t(n - 2) + t(1) + c(n - 1)] + t(1) + cn$

$= t(n - 2) + 2t(1) + c[(n - 1) + n]$

$= [t(n - 3) + t(1) + c(n - 2)] + 2t(1) + c[(n - 1) + n]$

$= t(n - 3) + 3t(1) + c[(n - 2) + (n - 1) + n]$

…

$= t(n - i) + it(1) + c[(n-i+1)+…+(n - 2) + (n - 1) + n]$

$= t(n - i) + it(1) + c(\sum_{j=0}^{i-1} n - j)$

$= t(n - n + 1) + (n - 1)t(1) + c(\sum_{j=0}^{n-2} n - j)$

$= t(1) + (n - 1)t(1) + c(\sum_{j=0}^{n-2} n - j)$

$= t(1) + nt(1) - t(1) + c(\sum_{j=0}^{n-2} n - j)$

$= nt(1) + c(\sum_{j=0}^{n-2} n - j)$

$= A + B$

Áp dụng cấp số cộng cho B

$B = c(\sum_{j=0}^{n-2} n - j)$

⇨ $B = c[(n - 2)(n + 1) / 2]$

Kết luân:

A: độ phức tạp là O(n)

B: độ phức tạp là O(n^2)

⇨ Chi phí thuật toán là O(n^2)

RadixSort Using Binary

```
int Get_Binary_Length(int number){
	int count = 1;
	while (number/2 > 0){
		number /= 2;
		count++;
	}
	return count;
}

int Get_Binary_Digit(int number, int k){
	int mod;
	for (int i = 0; i < k; i++){
		mod = number % 2;
		number /= 2;
	}
	return mod;
}

void Distribute(int*m_arr, int left, int right, int k){
	if (left < right && k>0){
		int i, j; int mid;
		int n_c1 = 0, n_c0 = 0, *c1 = NULL, *c0 = NULL;

		for (i = left; i <= right; i++){
			int digit = Get_Binary_Digit(m_arr[i], k);
			if (digit == 1){
				n_c1++;
				c1 = (int*)realloc(c1, n_c1*sizeof(int));
				c1[n_c1 - 1] = m_arr[i];
			}
			Else {
				n_c0++;
				c0 = (int*)realloc(c0, n_c0*sizeof(int));
				c0[n_c0 - 1] = m_arr[i];
			}
		}
		i = left;
		for (j = 0; j < n_c0; j++){
			m_arr[i] = c0[j];
			i++;
```

```
            }

            for (j = 0; j < n_c1; j++){
                  m_arr[i] = c1[j];
                  i++;
            }

            delete[]c0;
            delete[]c1;

            /*Recurison*/
            if (n_c0 == n_c1 && n_c0 == 0) return;

            mid = left + n_c0;

            Distribute(m_arr, left, mid-1, k - 1);
            Distribute(m_arr, mid, right, k - 1);
      }
}

void RadixSortUsingBinary(int *m_arr, int m_size){
      int max = Max(m_arr, m_size);
      int k = Get_Binary_Length(max);
      Distribute(m_arr, 0, m_size - 1, k);
}
```

Bucket sort

```
void bucketSort(int *inArr, int arrSize, int variance)
{
      int *bucket = new int[variance];

      // Initialize all bucket nodes to have a count of 0
      for (int i = 0; i < variance; i++)
            bucket[i] = 0;

      // Throw the count of each node into the bucket
      for (int i = 0; i < arrSize; i++){
            bucket[inArr[i]]++;
      }

      // We'll fill the new array with whatever is in the bucket
      int newArrayPosition = 0;
      for (int x = 0; x < variance; x++){
            for (int j = 0; j < bucket[x]; j++){
                  inArr[j + newArrayPosition] = x;
            }
            newArrayPosition += bucket[x];
      }
      delete[]bucket;
}
```

ĐẢO NGƯỢC DSLK Đơn

```
Node* Reverse(Node *head){
    Node *dummy = new Node;
    dummy->next = head;
    Node *p1 = dummy->next, *p2 = p1;
    if(p1->next!=NULL){
        p2=p1->next;
        while(p2->next!=NULL){
            p1=p2;
            p2=p2->next;
            p1->next=head;
            head = p1;
        }
        p2->next=p1;
        head=p2;
        dummy->next->next=NULL;
        delete dummy;
    }
    return head;
}
```

Đảo nguoc DSLK Đôi

```
Node* Reverse(Node* head)
{
  // If empty list, return
  if (!head)
      return NULL;

  // Otherwise, swap the next and prev
  Node *temp = head->next;
  head->next = head->prev;
  head->prev = temp;

  // If the prev is now NULL, the list
  // has been fully reversed
  if (!head->prev)
      return head;

  // Otherwise, keep going
  return Reverse(head->prev);
}
```

Josephus (cycle linkedlist)

```
void main(){
  List l;  int k, n, x;
  l.Head = l.Tail = NULL;
  cout << "Input K:";        cin >> k;
  cout << "Input anmount:";  cin >> n;

  for (int i = 1; i <= n; i++)
     addTail(l, i);
  Node*p1,*p2;
  do{
     p1 = l.Head;
     p2 = p1->pNext;
     for (int i = 0; i < k - 1; i++){
        p1 = p2; p2 = p2->pNext;
     }

     cout << p2->key <<"  ";
     p2 = p2->pNext;

     removeAffter(l, p1);//remove p2

     l.Head = p2;
     l.Tail = p1;

     } while (l.Head !=l.Tail);
  cout << endl << "- Alive: " <<
                      l.Head->key;
}
```

Sprase Table

```
Node* getNode(int value, int x, int y)
{
	Node*p = new Node;
	p->val = value;
	p->x = x;
	p->y = y;
	p->right = p->down = NULL;
	return p;
}

Node* createSpraseTable()
{
	return getNode(-1, -1, -1);
}

Node* getPosition(Node*head, int x, int y)
{
	Node*pr, *pd;
	for (pr = head; pr != NULL && pr->x != x; pr = pr->down)
		;

	if (pr == NULL)
		return NULL;

	for (pd = pr; pd != NULL && pd->y != y; pd = pd->right)
		;
	if (pd == NULL)
		return NULL;

	return pd;
}

void printTable(Node*head)
{
	if (!head)
		cout << "-empty table";
	else
	{
		if (!head->down || !head->right)
			cout << "-empty table";
		else
		{
			Node*pd, *pr;
			for (pd = head->down; pd != NULL; pd = pd->down){
				for (pr = pd->right; pr != NULL; pr = pr->right)
					cout << pr->val << endl;
			}
		}
	}
}
```

```
//Thêm một giá trị vào bảng (không trùng)
void insertNode(Node*&head, int val, int x, int y){
	if (!head)
		head = createSpraseTable();
	Node*p = getNode(val, x, y);
	insert(head, p);
}

void insert(Node*&head, Node*p){

	Node*pr1 = head, *pr2 = pr1->right;
	//Tìm vị trí cột sẽ chưa p
	while (pr2 && pr2->y <= p->y){
		pr1 = pr2; pr2 = pr2->right;
	}

	//Chưa có con trỏ quản lí cột chứa p
	if (pr1->y != p->y){
		Node*newCol = getNode(-1, -1, p->y);
		pr1->right = newCol;
		newCol->right = pr2;
		pr1 = newCol;
	}

	//Tìm vị trí chính xác của p (theo cột)
	pr2 = pr1->down;
	while (pr2 && pr2->x <= p->x){
		pr1 = pr2; pr2 = pr2->down;
	}

	//làm tương tự cho dòng
	Node*pd1 = head, *pd2 = pd1->down;
	while (pd2 && pd2->x <= p->x){
		pd1 = pd2; pd2 = pd2->down;
	}
	if (pd1->x != p->x){
		Node*newRow = getNode(-1, p->x, -1);
		pd1->down = newRow;
		newRow->down = pd2;
		pd1 = newRow;
	}

	pd2 = pd1->right;

	while (pd2 && pd2->y <= p->y){
		pd1 = pd2; pd2 = pd2->right;
	}
	//lien ket
	pr1->down = p; p->down = pr2;
	pd1->right = p; p->right = pd2;
}
```

In ra dãy con tăng nghiêm ngặt dài nhất

```
struct NODE
{
	NODE* prev;
	NODE* next;
	int   x;
	NODE* myBestNextNode;
	int     longestTail;
}

//Build
for (NODE* p = tail; p != NULL; p = p->Prev)
{
	bestTail = 1;
	for (NODE* q = p->Next; q != NULL; q = q->Next)
	{
		if (q->x > p->x) // nối được
		{
			if (bestTail < q->longestTail + 1)
			{
				bestTail = q->longestTail + 1;
				p->bestNextNode = q;
			}
		}
		p->longestTail = bestTail;
	}
}

//ket qua khong chac la head->bestTail
int res = 0;
for (p = head; p <= tail; p = p->Next)
	if (res<p->bestTail)
		res = p->bestTail;
```

Insertion Sort using LinkedList

```
void InsertionSort(List&l){
	Node*dummy = new Node;
	dummy->pNext = l.pHead;
	Node*i, *iprev, *p1, *p2;
	for (i = l.pHead->pNext; i; i = i->pNext){
		p1 = dummy;
		p2 = p1->pNext;
		while (p2 != i->pNext && CompareData(p2->info, i->info)<0){
			p1 = p2; p2 = p2->pNext;
		}
		if (p2 != i){
			for (iprev = p1; iprev->pNext != i; iprev = iprev->pNext)
				;
			iprev->pNext = i->pNext;
			p1->pNext = i;
```

```
                i->pNext = p2;
                i = iprev;
            }
        }
        l.pHead = dummy->pNext;
        delete dummy;
}
```

Selection Sort using LinkedList

```
void SelectionSort(List&l)
{
        Node *i = NULL, *j = NULL, *min = NULL;
        for (i = l.pHead; i->pNext; i = i->pNext){
            min = i;
            for (j = i->pNext; j; j = j->pNext){
                //if (CompareData(min->info, j->info) > 0)
                if (j->info.x <= min->info.x)
                    min = j;
            }
            Swap(l, min, i);
            //swap(min->info, i->info);
        }
}
//Swap 2 node without swap data
void Swap(List&l, Node*&a, Node *&b){
        Node*dummy = new Node;
        dummy->pNext = l.pHead;
        Node *a_prev = dummy, *b_prev = dummy, *tmp = NULL;
        for (a_prev; a_prev->pNext != a && a_prev; a_prev = a_prev->pNext) ;
        for (b_prev; b_prev->pNext != b && b_prev; b_prev = b_prev->pNext) ;
        if (!a_prev || !b_prev)
            return;//node *a and node *b is not exist in this list
        else if (a_prev == b){
            b->pNext = a->pNext;        a->pNext = b;          b_prev->pNext = a;
        }
        else if (b_prev == a){
            a->pNext = b->pNext;        b->pNext = a;          a_prev->pNext = b;
        }
        else {
            a_prev->pNext = b;
            b_prev->pNext = a;
            tmp= a->pNext;
            a->pNext = b->pNext;
            b->pNext = tmp;
        }
        //restore each pointer address (this func just swap data of node)
        tmp = a; //a_prev: temp
        a = b;
        b = tmp;
        l.pHead = dummy->pNext;
        delete dummy;
}
```

QuickSort using LinkedList

```
void QuickSort(List&l){
   if ((l.pHead != l.pTail))
   {
       Node*p = Partition(l);

       List l1, l2;
       InitList(l1); InitList(l2);
       Node*t = l.pHead;

       //split
       while(l.pHead && l.pHead != p)
       {
            t = l.pHead;
            l.pHead = t->pNext;
            t->pNext = NULL;
            AddTail(l1, t);
       }

       if (l.pHead)
       {
            l.pHead = p->pNext;

            while (l.pHead)
            {
                 t = l.pHead;
                 l.pHead = t->pNext;
                 t->pNext = NULL;
                 AddTail(l2, t);
            }
       }

       QuickSort(l1);
       QuickSort(l2);
```

```
Node *Partition(List&l){
   Node*dummy = new Node;
   dummy->pNext = l.pHead;
   Node* i = dummy;
   for (Node*j = l.pHead; j&&j !=
          l.pTail; j = j->pNext){
       if (j->info.x <= l.pTail-
                       >info.x){
          i = i->pNext;
          Swap(l, i, j);
       }
   }
   i = i->pNext;
   Swap(l, i, l.pTail);
   delete dummy; dummy = NULL;
   return i;
}
```

```
       //join
       if (l1.pHead){
          l.pHead = l1.pHead;
          l1.pTail->pNext = p;
       }
       else
          l.pHead = p;

        p->pNext = l2.pHead;

        if (l2.pHead)
          l.pTail = l2.pTail;
        else
          l.pTail = p;
  }//end if đầu
}//end hàm
```

MergeSort uisng LinkedList

```
void Merge(List&l, List&l1, List&l2){
     Node *p;
     while (l1.pHead && l2.pHead){
          if (l1.pHead->info.x <=
               l2.pHead->info.x){
               p = l1.pHead;
               l1.pHead = p->pNext;
          }
          else{
               p = l2.pHead;
               l2.pHead = p->pNext;
          }
          p->pNext = NULL;
          AddTail(l, p);
     }
```

```
void DistributeList(List&l, List&l1,
                                List&l2){
     Node *p;
     do{//spit l into l1 & l2
          p = l.pHead;
          l.pHead = p->pNext;
          p->pNext = NULL;
          AddTail(l1, p);
     } while ((l.pHead) &&(p->info.x
               <= l.pHead->info.x));
     if (l.pHead)
          DistributeList(l, l2, l1);
     else
          l.pTail = NULL;
}
```

```
      //noi phan con lai
      while (l1.pHead){
            p = l1.pHead;
            l1.pHead = l1.pHead->pNext;
            p->pNext = NULL;
            AddTail(l, p);
      }
      while (l2.pHead){
            p = l2.pHead;
            l2.pHead = l2.pHead->pNext;
            p->pNext = NULL;
            AddTail(l, p);
      }
}
```

```
//Merge Sort
void MergeSort(List&l)
{
      if (l.pHead == l.pTail) return;
      List l1, l2;
      InitList(l1); InitList(l2);
      DistributeList(l, l1, l2);
      if (l1.pHead && l2.pHead)
      {
            MergeSort(l1);
            MergeSort(l2);
      }
      Merge(l, l1, l2);
}
```

RadixSort using linkedlist

//Sách cấu trúc dữ liệu trang 132

KMP

```
int KMP_Search(char*P, char*T){
      int m = strlen(P);
      int *pi = NULL;
      ComputeArray(P, pi);
      int q = 0;
      for (int i = 0; i < strlen(T); i++){
            while (q>0 && P[q] != T[i])
                  q = pi[q];
            if (P[q] == T[i])
                  q++;
            if (q == m){
                  cout << i - m + 1 << endl;
                  q = pi[q];
            }
      }
      return 0;
}
void ComputeArray(char*P, int *&pi){
      int m = strlen(P);
      pi = new int[m + 1];
      pi[0] = -1; //bor khong dung`
      pi[1] = 0;
      int k = 0;
      for (int q = 2; q <= m; q++){
            while (k>0 && P[k]!=P[q-1]){
                  k = pi[k];// k giam
            }
            if (P[k] == P[q - 1])
                  k++;
            pi[q] = k;
      }
}
```

Rabin Karp

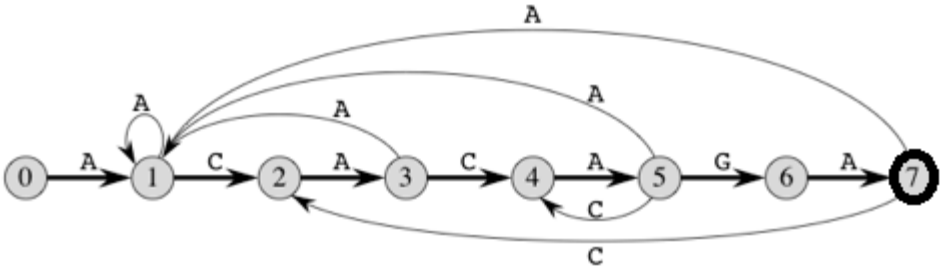
```
#define d 256 //xet bo ki ASCII 2566 ki tu
void search(char pat[], char txt[], int q)
{
	int M = strlen(pat);
	int N = strlen(txt);
	int i, j;
	int p = 0; // hash value for pattern
	int t = 0; // hash value for txt
	int h = 1;

	// The value of h would be "pow(d, M-1)%q"
	for (i = 0; i < M - 1; i++)
		h = (h*d) % q;

	// Calculate the hash value of pattern and first window of text
	for (i = 0; i < M; i++) {
		p = (d*p + pat[i]) % q;
		t = (d*t + txt[i]) % q;
	}

	// Slide the pattern over text one by one
	for (i = 0; i <= N - M; i++){
		// Check the hash values of current window of text
		// and pattern. If the hash values match then only
		// check for characters on by one
		if (p == t)
		{
			/* Check for characters one by one */
			for (j = 0; j < M; j++)
				if (txt[i + j] != pat[j])
					break;

			// if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
			if (j == M)
				printf("Pattern found at index %d \n", i);
		}

		// Calculate hash value for next window of text: Remove
		// leading digit, add trailing digit
		if (i < N - M)
		{
			t = (d*(t - txt[i] * h) + txt[i + M]) % q;

			// We might get negative value of t, converting it to positive
			if (t < 0)
				t = (t + q);
		}
	}
}
```

DFA

```
#include "func.h"

int DFA_Search(char*P, char*T)
{
	char*Sigma = getSigma(T);
	int m = strlen(P);
	int n = strlen(Sigma);
	int **dfa = new int*[m + 1];

	for (int i = 0; i < m + 1; i++)
	{
		dfa[i] = new int[n];
		for (int j = 0; j < n; j++)
			dfa[i][j] = 0;
	}

	ComputeTable(dfa, P, Sigma);

	int q = 0;
	//Tìm trong van bản T
	for (int i = 0; i < strlen(T); i++){
		int j = getIndexSigma(Sigma, T[i]);
		q = dfa[q][j];
		if (q == m)
			cout << (i-m+1) << endl;
	}
	return 0;
}

void ComputeTable(int **&dfa, char*P, char*Sigma)
{
	int m = strlen(P);
	for (int q = 0; q <= m;q++)
	for (int i = 0; i < strlen(Sigma); i++)
	{
		int k = (m + 1 < q + 2) ? m + 1 : q + 2;
		do
		{
			k--;
		} while (!isPosfix(P,k,q,Sigma[i]));//Pk la hậu tố của Pq+Sigma[i]
		dfa[q][i] = k;
	}
}

char* getSigma(char*T)
{
	char*Sigma = NULL;
	int k = 0;
	bool flag = true;
```

```
	for (int i = 0; i < strlen(T); i++)
	{
		flag = true;
		for (int j = 0; j < k; j++)
		{
			if (Sigma[j] == T[i])
			{
				flag = false; break;
			}
		}
		if (flag)
		{
			Sigma = (char*)realloc(Sigma, (k + 1)*sizeof(char));
			Sigma[k++] = T[i];
		}
	}
	Sigma[k] = 0;
	return Sigma;
}

int getIndexSigma(char* Sigma, char a)
{
	for (int i = 0; i < strlen(Sigma);i++)
	if (Sigma[i] == a)
		return i;
	return -1;
}

bool isPosfix(char*P, int k, int q, char a)
{//kiem tra Pk hau to cua Pq+a
	if (k == 0)
		return true;
	int i, j;

	bool kq = P[k-1] == a; //kí tự cuối trùng

	if (kq)
	{
		for (i = k - 2, j = q-1; i >= 0 && j >= 0; i--, j--)
		{
			if (P[i] != P[j])//P[i] thuộc Pk, P[j] thuộc Pq+a
			{
				kq = false; break;
			}
		}
	}
	return kq;
}
```

EDFA – Efficient Construction of Finite Automata



state	character A	C	G	T
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

P= "ACACAGA"

1) Fill the first row.All entries in first row are always 0 except the entry for pat[0] character.For pat[0] character, we always need to go to state 1.

2) Initialize lps as 0. lps for the first index is always 0.

3) Do following for rows at index i = 1 to M. (M is the length of the pattern)
   a) Copy the entries from the row at index equal to lps.

   b) Update the entry for pat[i] character to i + 1.
   c) Update lps "lps = TF[lps][pat[i]]" where TF is the 2D array which is being constructed.

1) Điền hàng đầu tiên. Tất cả ô hàng đầu luôn là 0 trừ ô thứ **'pat[0]=1'**

2) Khởi tạo **lps** = 0. lps cho chỉ mục đầu tiên luôn luôn là 0.

3) Làm theo các hàng tại chỉ số i = 1 đến M. (M là chiều dài của mẫu)
   a) Sao chép các mục từ hàng tại chỉ mục bằng lps.
   b) Cập nhật: 'TF[i][**pat[i]**] = **i+1**'.
   c) Cập nhật lps "lps = TF [lps] [pat [i]]" trong đó TF là mảng 2D đang được xây dựng.

```
#define NO_OF_CHARS 256
/* Builds the TF table which represents Finite Automata for a given pattern */
void computeTransFun(char *pat, int M, int**TF)
{
	int i, lps = 0, x;

	// Fill entries in first row
	for (x = 0; x < NO_OF_CHARS; x++)
		TF[0][x] = 0;
	TF[0][pat[0]] = 1;

	// Fill entries in other rows
	for (i = 1; i <= M; i++)
	{
		// Copy values from row at index lps
		for (x = 0; x < NO_OF_CHARS; x++)
			TF[i][x] = TF[lps][x];

		// Update the entry corresponding to this character
		TF[i][pat[i]] = i + 1; //Từ trạng thái i gặp P[i]=>trạng thái i+1

		// Update lps for next row to be filled
		if (i < M)
			lps = TF[lps][pat[i]];//cập nhật lps = tr/thái lps khi gặp P[i]
	}
}

/* Prints all occurrences of pat in txt */
void search(char *pat, char *txt)
{
	int M = strlen(pat);
	int N = strlen(txt);

	int**TF = new int*[M + 1];
	for (int i = 0; i <= M; i++)
		TF[i] = new int[NO_OF_CHARS];

	computeTransFun(pat, M, TF); //Precompute table

	// process text over FA.
	int i, j = 0;
	for (i = 0; i < N; i++)
	{
		j = TF[j][txt[i]];
		if (j == M)
			printf("\n pattern found at index %d", i - M + 1);
	}
	for (int i = 0; i <= M; i++)
		delete TF[i];
}
```