Algorithm Homework 1

Jingwei Zhang 201528013229095

2015-10-8

1 Problem 1

1.1 Algorithm

The problem given can be converted to a equivalent problem: Given two sorted array A and B with length N, find the N-th element $C[N]$ in the merged sorted array C with length $2N$. Find a algorithm using $O(\log n)$ queries for $A[k]$ or $B[k']$.

The basic idea of my algorithm is dividing the array currently dealing with A' and B' with length l into two parts A'_l(left part, smaller than $A'[\frac{l}{2}]$), parts A'_r(right part, larger than $A'[\frac{l}{2}]$) and B'_l, B'_r. By comparing $A'[\frac{l}{2}]$ and $B'[\frac{l}{2}]$, determining the possible part containing medium of the two array.

For $A'[\frac{l}{2}] = B'[\frac{l}{2}]$, the medium is $A'[\frac{l}{2}]$ or $B'[\frac{l}{2}]$. If $A'[\frac{l}{2}] \leq B'[\frac{l}{2}]$, then $A'[i](i = 0, \ldots, \frac{l}{2} - 1)$ must be located at $[0, l)$ of the merged array and $B'[i](i = \frac{l}{2} - 1, \ldots, l - 1)$ must be located at $[l + 1, 2l)$ of the merged array, which do not contain the medium. So the Problem become find the medium in $A'[\frac{l}{2}, l)$ and $B'[0, \frac{l}{2}]$. For $A'[\frac{l}{2}] \geq B'[\frac{l}{2}]$, it is similar.

Pseudocode:

```
FIND-MEDIUM($A, B, l_A, r_A, l_B, r_B$)
     ▷ Find the medium number from $A[l_A, r_A)$ and $B[l_B, r_B)$
     ▷ Index starts from 0
     ▷ $A[k]$ means query the k-th smallest element in A
 1  if $r_A - l_A == 1$
 2     then return $min(A[l_a], B[l_a])$
 3  if $r_A - l_A == 1$
 4     then Query for all 4 numbers$A[l_A], A[l_A + 1], B[l_B], B[l_B + 1]$
 5           $ans$ =The second small number among these 4 numbers
 6           return ans
 7  $m_A = \lfloor \frac{l_A + r_A}{2} \rfloor$
 8  $m_B = \lceil \frac{l_B + r_B}{2} \rceil$
 9  if $A[m_A] == B[m_B]$
10     then return $A[m_A]$
11  elseif $A[m_A] \leq B[m_B]$
12     then return FIND-MEDIUM($A, B, m_A, r_A, l_B, m_B + 1$)
13  else return FIND-MEDIUM($A, B, l_A, m_A + 1, l_B, r_B$)
```

1.2 Sub problem Reduction Graph

See graph section.

1.3 Correctness

As mentioned in algorithm part, the cutting operation guarantees that the medium is not in the cut part and one cut part smaller than the medium and one part greater than the medium. And the two part have the exactly the same length. So by cutting the two part, the medium of original array is the medium of the array left and the two arrays left have exactly the same length, which are maintained during the recursion.

When the length of each array is reduced to one the medium of these two is the smaller one. When the length of each array is reduced to two, one possible condition is that we will not cut it since one part has length 0 and then recursion will not end. So we list all numbers then pick 2nd. This guarantees recursion will stop.

1.4 Complexity

We can make the length of array left denote the size of this problem. During each recursion, we query for $A[m_A]$and$B[m_B]$, that is $O(1)$ and the total length is half of origin. So we can get:

$$T(n) = T(n/2) + O(1)$$

Since:

$$\begin{aligned} T(n) &= T(n/2) + O(1) \\ &= \sum_{1}^{\log n} O(1) \\ &= O(\log n) \end{aligned}$$

So we will query for $O(\log n)$ totally.

2 Problem 3

2.1 Algorithm

The idea is almost the same as Merge and Count as long as we add one pointer pointing the possible inversions and count them. We count inversions of every number ing the right part, that is from the first number greater than 3 times of this in the left part to the last number of left part.

Pseudocode:

```
SORT-COUNT($A$)
1  Divide A into two sub-sequence L and R
2  $(RC_L, L) = $ SORT-COUNT($L$)
3  $(RC_R, R) = $ SORT-COUNT($R$)
4  $(C, A) = $ MERGE-COUNT($L, R$)
5  return $(RC = RC_L + RC_R + C)$
```

```
MERGE-COUNT($L, R$)
 1  $RC = 0, i = 0, j = 0, p = 0$
 2  for $k = 0$ to $||L|| + ||R|| - 1$
 3        do
 4           while $L[p] < 3R[j]$
 5                do $p++$
 6           if $L[p] > R[j]$
 7              then $RC += (\frac{n}{2} - p)$
 8           if $L[i] > R[j]$
 9              then $A[k] = R[j++]$
10              else
11                    $A[k] = L[i++]$
```

2.2 Sub problem Reduction Graph

See graph section.

2.3 Correctness

During reduction, we guarantee that the array dealing with is sorted and we have counted all inversions within this problem. The correctness of sorting is the same as merge sort so we will not cover that.

Suppose we have two sub arrays(L and R) sorted and counted all inversions within each array. Now, for merged array(A), it only contains the "cross" inversions between the left sorted part and right sorted part. The total cross inversions can be the sum of inversions contains each element of R.

When scanning element in R from left to right(current index j). L[p] is always the smallest element guaranteeing $L[p] > 3R[j]$(note that $L[p] \leq 3R[j]$). For the first element it is obvious, the while loop begins on line 3 in MERGE-COUNT does that. So all elements right side $L[p]$(included) can from an inversion with $R[j]$. That is what line 7 does. When j goes to j+1, note that $R[j] \leq R[j+1]$, so $L[p-1] \leq 3R[j] \leq 3R[j+1]$, so the smallest element k guaranteeing $L[k] > 3R[j]$ is at least p. Then the While loop on line 3 performs a linear search which guarantees $L[p'] > 3R[j+1]$ after that.

From all above, we have proven the correctness of this algorithm.

2.4 Complexity

Dividing two sub problems costs only constant time. Problem is divided into two sub problems, each with half size.Since i, j, k only grow, it costs liner time to combine two sub problems. So we can get:

$$T(n) = 2T(n/2) + O(n)$$

According to master theory:

$$T(n) = O(nlogn)$$

3 Problem 6

3.1 Algorithm

For a given convex polygon,a specific edge may belong to different triangles. Such a triangle divide the polygon into two small polygons(it may degraded to one small polygon). Then when such a triangle exists, the number of total possible partitions will be the production of the numbers of possible partitions of these two small polygons. These small problems can be solved recursively since there is only one way for triangle. So we can enumerate all triangles containing this specific edge, compute all these number of partitions and then add them up.

To avoid computing the same problem repeatedly, we can store the result into an array A.

Pseudocode:

```
COUNT-PARTITION($n$)
    ▷ $A$ is the array storing results of smaller problems
    ▷ Index denotes the size of problem(vertices of the convex polygon)
1   for $i = 0$to 3
2       do $A[i] = 1$ ▷ Leting $A[0]$ to $A[2]$ equals 1 can make multiplication simpler.
3   for $i = 4$to $n$
4       do $A[i] = -1$ ▷ Marked not computed yet.
5   return COUNT-FORMULAR(n)
```

```
Count-Formular($n$)
1  if $A[n] > 0$
2     then return $A[n]$
3     else
4           $sum = 0$
5           for $i = 1$ to $n-2$
6                do $sum+ = A[i+1] * A[n-i]$
7  return $sum$
```

3.2 Sub problem Reduction Graph

See graph section.

3.3 Correctness

- Firstly, the correctness of division. From geometry, we know that linking every two non-adjacent vertices will form two small convex polygon. So such a triangle will divide the big convex polygon into two small ones.
- Then, the correctness of computing from small problems to big problem.
  - Since one edge can below to $n-2$ different triangles and in every partition this edge must from a triangle with another vertex, total partition can be divided into $n-2$ types,these include all possible partitions.
  - For partitions in different types, the dividing triangle is different, so they are different.
  - For one type, the two small problems are independent, so we can simply multiply the number of these two.

  So, combining small problems into big one include all situations non-repeatedly.
- Finally, obviously, for $n = 3$ only one partition can be found.For $n = 2$ let it equal 2 to make it computable for multiplication.Thus the recursion will end.

3.4 Complexity

The size of this problem can be denoted by the number of vertices of convex polygon. Note for array A every item was computed at most once and if we separate the summing up part and recursion part on line 6, which have no influence on the running time, when computing $A[k]$(summing up part), $A[i], i = 1, \ldots, k-1$ has been computed. So if we analysis a "bottom-up" approach(actually it is the recall part of "top-down" approach), we can get:

$$\begin{aligned} T(n) &= T(n-1) + \sum_{1}^{n-2} O(1) \\ &= T(n-1) + O(n) \\ &= \sum_{i=3}^{n} O(i) \\ &= O(n^2) \end{aligned}$$

So the time complexity is $O(n^2)$ and space complexity is $O(n)$, since we use a extra array A with length n.

4 Sub problem Graph

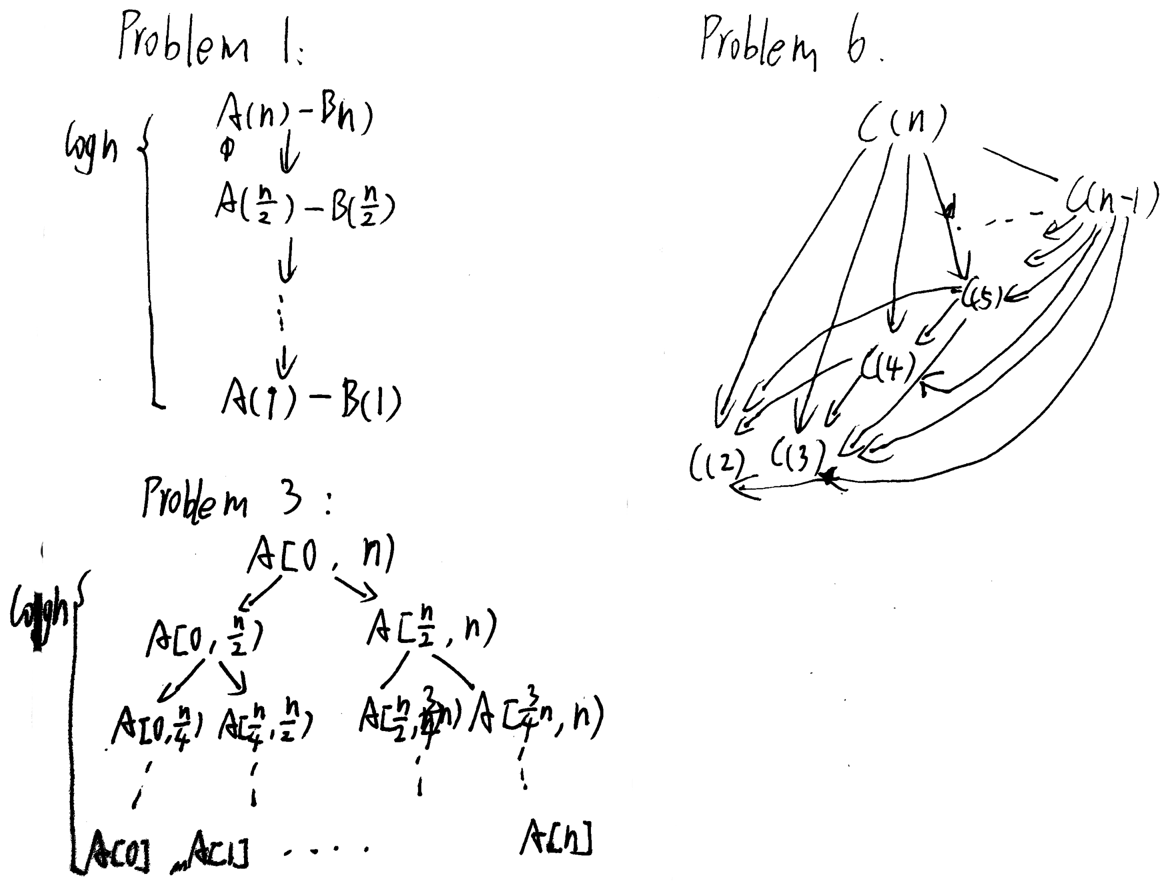

Figure 1: subproblem reduction graph

5 Problem 7

Method

It is Possible to count inversions using quick sort algorithm in $O(n \log n)$ time. We can count inversions during the partition. In every recursion:

- count the inversions between pivot and others,
- count "relative" inversions between left part(smaller than pivot) and right part(larger than pivot),
- guarantee the relative order of numbers in the left part is exact the same as the ralative order of these numbers in the original sequence, so does the right part.

Result

Both report that there are 2500572073 inversions in given sequence.

Running time varies when program runs repeatedly, generally they are almost the same, around 12ms. It is hard to say which runs faster under such circumstance.

Code

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cstdio>
#include <cstdlib>
#include <algorithm>
#include <chrono>
using namespace std;
typedef long long LL;
const string file_name("Q5.txt");
// Merge sort part
// Merge c[s,m)(a) and c[m,t)(b) to c[s,t)
vector<int> a;
vector<int> b;
LL merge_count(vector<int> &c,int s,int m,int t){
    LL cnt = 0;
    //vector<int> a(m - s);
    a.resize(m-s);
    for(int i = s;i<m;i++){
        a[i-s] = c[i];
    }
    //vector<int> b(t - m);
    b.resize(t-m);
    for(int i = m;i<t;i++){
        b[i-m] = c[i];
    }

    int i = 0,j = 0,k = s;// a[i], b[j], c[k]
    while(i < a.size() && j< b.size()){
        if(a[i] < b[j]){
            c[k++] = a[i++];
        } else {
            c[k++] = b[j++];
            // inversion counting
            cnt += a.size() - i;
        }
    }
    while(i < a.size()){
        c[k++] = a[i++];
    }
    while(j < b.size()){
        c[k++] = b[j++];
    }
    return cnt;
}

LL count_inversions_merge_sort(vector<int> &v,int s,int t){
    LL cnt = 0;
    if(t-s>1){
        int m = (s + t) / 2;
        LL cnt_l = count_inversions_merge_sort(v,s,m);
        LL cnt_r = count_inversions_merge_sort(v,m,t);
        LL cnt_m = merge_count(v,s,m,t);
        cnt = cnt_l + cnt_r + cnt_m;
    } else {
        //contains only one or 0 elements
        cnt = 0;
    }
    return cnt;
}

// returns the index of pivot
vector<int> larger;
int partition_count(vector<int> &v,int s,int t,LL &cnt){
    int pi = rand() % (t - s) + s;
    int p = v[pi];
    //vector<int> larger;
    larger.clear();
    int j = s;// smaller part: v[s,j)
    for(int i = s;i<t;i++){
        if(i == pi) {
            continue;
        }
        if(v[i] < p){
            // pivot inversion
            if(i > pi){
                cnt++;
            }
            // v[i] inversion
            cnt += larger.size();
            v[j++] = v[i];
        } else if(v[i] > p){
            // pivot inversion
            if(i < pi){
                cnt++;
            }
            larger.push_back(v[i]);
        }
    }
    int m = j;
    v[m] = p;
    for(int i = 0;i<larger.size();i++){
        v[m+i+1] = larger[i];
    }
    return m;
}

LL count_inversions_quict_sort(vector<int> &v,int s,int t){
    LL cnt = 0;
    if(t-s>1){
        int m = partition_count(v,s,t,cnt);
        LL cnt_l = count_inversions_quict_sort(v,s,m);
        LL cnt_r = count_inversions_quict_sort(v,m+1,t);
        cnt += (cnt_l + cnt_r);
    }
    return cnt;
}


int main()
{
    vector<int> s;

    // Input part
    ifstream s_file;
    s_file.open(file_name);
    int a;
    while(s_file>>a){
        s.push_back(a);
    }
    s_file.close();

    // Merge Sort part;
    vector<int> m(s);
    auto start = chrono::high_resolution_clock::now();
    cout<<count_inversions_merge_sort(m,0,m.size())<<"\t";
    auto diff = chrono::duration_cast<chrono::milliseconds>
        (chrono::high_resolution_clock::now()-start);
    cout<<diff.count()<<"ms"<<endl;
    // Quick Sort part
    vector<int> q(s);
    start = chrono::high_resolution_clock::now();
    cout<<count_inversions_quict_sort(q,0,q.size())<<"\t";
    diff = chrono::duration_cast<chrono::milliseconds>
        (chrono::high_resolution_clock::now()-start);
    cout<<diff.count()<<"ms"<<endl;
    return 0;
}
```

6 Problem 10

Result

Below is a table showing the Runing time of Karatsuba Algorithm and grade school method.

Table 1: Camparation: Running Time of Two algorimthss

Length N (Decimal)	1280	2560	5120	10240	20480	40960
Grade School Method(ms)	21	86	346	1389	5526	22047
Karatsuba Algorithm(ms)	86	241	738	2192	7010	19952

From table above we can see that grade school method runs faster when N is small, but when N grows, their difference gets smaller and finally when N = 40960 Karatsuba Algorithm wins.

Code

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <vector>
#include <algorithm>
#include <ctime>
#include <chrono>
using namespace std;
class big_int{
public:
```

```
vector<char> n;
// Constructor
big_int(){}
// digits range from n[left,right)
big_int(big_int &a,int left, int right){
    for(int i = left;i < right;i++){
        n.push_back(a[i]);
    }
}

// Common operations
const int size() const{
    return n.size();
}
void clear(){
    n.clear();
}
void init0(){
    n.clear();
    n.push_back(0);
}
char &operator[](const int i){
    return n[i];
}
// Plus equal
big_int &operator+=(const big_int &b){
    int l = min(this->size(),b.size());
    int carry = 0;
    vector<char> &ans = this->n;
    for(int i = 0;i < l;i++){
        ans[i] = ans[i] + b.n[i] + carry;
        if(ans[i] >= 10){
            ans[i] -= 10;
            carry = 1;
        } else {
            carry = 0;
        }

    }
    // If a is longer than b
    for(;l<ans.size();l++){
        ans[l] = ans[l] + carry;
        if(ans[l] >= 10){
            ans[l] -= 10;
            carry = 1;
        } else {
            carry = 0;
        }
    }
    // If b is longer than a
    for(;l<b.size();l++){
        ans.push_back(b.n[l] + carry);
        if(ans[l] >= 10){
            ans[l] -= 10;
            carry = 1;
        } else {
                carry = 0;
            }
        }
        if(carry > 0){
            ans.push_back(carry);
        }
        return *this;
    }

    // Plus operation
    big_int &operator+(const big_int &b){
        big_int &ret = *new big_int(*this);
        ret += b;
        return ret;
    }

    // this must be larger than b
    big_int &operator-=(const big_int &b){
        int carry = 0;
        big_int &a = *this;
        int l = min(a.size(),b.size());
        for(int i = 0;i<l;i++){
            a[i] = a[i] - b.n[i] + carry;
            if(a[i] < 0){
                a[i] += 10;
                carry = -1;
            } else {
                carry = 0;
            }
        }
        for(int i = l;carry<0 && i<a.size();i++){
            a[i] += carry;
            if(a[i] < 0){
                a[i] += 10;
                carry = -1;
            } else {
                carry = 0;
            }
        }
        return a;
    }

    big_int &operator<<(int b){
        big_int &a = *this;
        for(int i = 0;i < b;i++){
            a.n.push_back(0);
        }
        for(int i = a.n.size()-1;i >= b;i--){
            a[i] = a[i-b];
        }
        for(int i = 0;i < b;i++){
            a[i] = 0;
        }
        return a;
    }
    big_int &naive_mul(big_int &b);
    big_int &operator*(big_int &b);

    big_int &operator*(int b){
        big_int &a = *new big_int(*this);
        int carry = 0;
        for(int i = 0;i< a.size();i++){
            a[i] = a[i] * b + carry;
            carry = a[i] / 10;
            a[i] %= 10;
        }
        if(carry > 0){
            a.n.push_back(carry);
        }
        return a;
    }
};
// Stream out operation
ostream &operator<<(ostream &os, big_int &a){
    bool leading_zero = true;
    for(auto rit = a.n.crbegin();rit != a.n.crend(); rit++){
        if(leading_zero && *rit != 0){
            leading_zero = false;
        }
        if(!leading_zero || (rit+1) == a.n.crend() ){
            os<< (char)(*rit + '0');
        }
    }
    return os;
}
// Stream in operation
istream &operator>>(istream &in, big_int &a){
    //first clear n
    a.n.clear();
    string buf;
    in>>buf;
    if(in){
        for(auto rit = buf.crbegin(); rit != buf.crend(); rit++){
            a.n.push_back(*rit - '0');
        }
    } else {
        // Do nothing
    }
    return in;
}
big_int& big_int::naive_mul(big_int &b){
    big_int &res = *new big_int();
    big_int &a = *this;
    res.init0();
    for(int i = 0;i < b.size();i++){
        big_int &t = a * (int)b[i];
        res += (t<<i);
        delete &t;
    }
    return res;
}
big_int& big_int::operator*(big_int &b){
    big_int &a = *this;
    if(a.size() <= 1){
        // when a contains only one digit
        return (b * (int)a[0]);
    } else if(b.size() <= 1){
        // when b contains only one digit
        return (a * (int)b[0]);
    } else {
        int n = min(a.size(),b.size());
        n /= 2;
        big_int &x_l = *new big_int(a,0,n);
        big_int &x_h = *new big_int(a,n,a.size());
        big_int &y_l = *new big_int(b,0,n);
        big_int &y_h = *new big_int(b,n,b.size());

        big_int &x_h_y_h = x_h * y_h;
        big_int &x_l_y_l = x_l * y_l;

        big_int &sum = (x_h + x_l) * (y_h + y_l);
        sum -= (x_h_y_h + x_l_y_l);
        big_int &ans = (x_h_y_h<<(n*2)) + (sum<<n) + x_l_y_l;
        big_int &ret = *new big_int(ans);
        delete &x_l;delete &x_h;delete &y_l;delete &y_h;
        return ret;
    }
}

// below for testing
inline int gen_digit(){
    return rand()%10;
}
inline int gen_not0_digit(){
    return rand()%9+1;
}
big_int& gen_big_int(int len){
    big_int &ret = *new big_int();
    if(len>1){
        ret.n.push_back(gen_not0_digit());
        for(int i = 1;i<len;i++){
            ret.n.push_back(gen_digit());
        }
    } else if(len == 1){
        ret.n.push_back(gen_digit());
    }
    return ret;
}


int main()
{
    freopen("result.out","w",stdout);
    srand(time(NULL));
    int i;
    for(i = 10;i<=40960;i=i<<1){
        big_int a = gen_big_int(i);
        big_int b = gen_big_int(i);
        cout<<i<<endl;
        // Compare multiplication execution time
        auto start = chrono::high_resolution_clock::now();
        big_int &ans1 = a.naive_mul(b);
        //cout<<ans1<<endl;
        auto diff = chrono::duration_cast<chrono::milliseconds>
            (chrono::high_resolution_clock::now()-start);
        cout<<"grade_method:"<<diff.count()<<"ms"<<endl;
        delete &ans1;

        start = chrono::high_resolution_clock::now();
        big_int &ans2 = a*b;
        //cout<<ans2<<endl;
        diff = chrono::duration_cast<chrono::milliseconds>
            (chrono::high_resolution_clock::now()-start);
        cout<<"Karatsuba's_method:"<<diff.count()<<"ms"<<endl;
        delete &ans2;
        cout<<endl;
    }
    return 0;
}
```