CSCI-SHU 220: Algorithms @ NYU Shanghai — Last Updated: September 28, 2020
Homework 2 — Due Date: October 16, 2020, 11:00pm

You are allowed to discuss with others but not allow to use references other than the course notes and reference books. Please list your collaborators for each questions. Write your own solutions and make sure you understand them.

There are 65 marks in total (including the bonus). The full mark of this homework is 50.

Enjoy :).

Please specify the following information before submission:

- Your Name: Zhenming Wang
- Your NetID: zw1806
- Collaborators:

Problem 1: Searching [15 marks]

(a) (5 marks) Consider the following `MysterySearch` algorithm.

```
// The classic binary search algorithm that we know.
int BinarySearch(int arr[], int low, int high, int x);

// The mystery search algorithm.
int MysterySearch(int arr[], int n, int x) {
  int low = 1; // arr[1] is the first element in arr[].
  while (low <= n && arr[low] < x) {
    low *= 2;
  }

  return BinarySearch(arr, low/2, min(n, low), x);
}
```

Assume that `x` can be found in the array `arr` at index r, that is, `arr`$[r]$ = `x`. What is the time complexity of `MysterySearch` in terms of r?

(b) (5 marks) Given two sorted arrays A and B, each of size n, devise an algorithm to find the k-th smallest element of among A and B. You will get full marks if the algorithm is of complexity $O(\log k)$.

(c) (5 marks) Let there be a $n \times n$ matrix $(a_{i,j})_{1 \le i \le n, 1 \le j \le n}$ such that $a_{i,j} \le a_{i+1,j}$ and $a_{i,j} \le a_{i,j+1}$, that is, the elements of each row or column of the matrix is in increasing order. Devise an algorithm that takes as inputs the matrix $(a_{i,j})_{1 \le i \le n, 1 \le j \le n}$ of size $n \times n$ and a number b, and that outputs whether there is an element $a_{i,j}$ such that $a_{i,j} = b$ for some number b. You will get full marks if the algorithm is correct of complexity $O(n^{1.59})$.

Solution. **Please write down your solution to Problem 1 here.**

(a) Problem1
Since we assume that x is to be found in the array at index r, the while loop would run in $O(\log r)$ time. Then, the Binary Search process which runs in the interval $low/2, min(n, low)$, would cost $O(\log r)$ time, since the interval can not be larger than r.
We combine the operations and get the time complexity of $O(\log r + \log r)$ which is $O(\log r)$.

(b) Problem2

```
int kthSmallest(int[] A,int[] B,k){
    if(k>A.length+B.length||k<1){
        return -1;
    }
    int high1 = A.length-1;
    int high2 = B.length-1;
    return helper(A,B,0,high1,0,high2,k);
}

int helper(int[]A,int[]B,int low1,int high1,int low2,int high2,k){
    if(low1 >= high1){
        return B[low1+k-1];//all of the elements in A have been excluded
    }
    if(low2>= high2){
        return A[low2+k-1];//elements in B have been excluded
    }

    i = min(high1,(low1+k)/2);
    j = min(high2,(low2+k)/2);

    if(A[i-1]<B[j-1]){
        return helper(A,B,i,high1,low2,high2,k-i-low1);
        //we excluded the i amount of element from A
    }

    if(A[i]>B[j-1]{
    return helper(A,B,low1,high1,j,high2,k-j-low2);
    //we excluded the j amount of element from B
    }

}
```

explanation
Since we have two sorted arrays and would like to find the *kth* smallest element among them, we could think of using binary search.
We start by checking the element at $min(k/2, size-1)$ position in each array. We denote the two indexes as i,j. If $A[i] < B[j]$, one thing we could be sure is that the element A[0] ... A[i] are all less than B[j], which is to say that we have found the i smallest elements. However, we are not eligible to judge the element before B[j] yet. So, we could now put the i elements in our pockets and carry on. Now that we have already found i elements, our goal now is to find the rest $k-j$ smallest elements, from the arrays A[i+1:] and B.
We could see that this process is essentially the same with binary search, that every time we work on a part less than before.

(c) We divide a big matrix into 4 parts

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

where A_{ij} is an $n/2 \cdot n/2$ matrix.
The small problem we identify for this problem is when a matrix contains a single element, so searching in such a matrix takes $O(1)$ time.

```
searchMatrix(A,n,value){
    //base case:matrix only have one element
    if(n==1){
        return A[1,1] == value;
    }
    else{
        mid = n/2;
        if(A[mid,mid] > value){
            return searchMatrix(A11,n/2,value)
        }
        if(A[mid,mid] < value{
            s1 = searchMatrix(A12,n/2,value);
            s2 = searchMatrix(A21,n/2,value);
            s3 = searchMatrix(A22,n/2,value);

            return s1||s2||s3;
        }
        if(A[mid,mid] == value{
            return True;
        }

    }
}
```

Since we already know that the matrix is sorted in each column and each row, we could pick the element that lies in the middle of an $n * n$ matrix A and compare A[n/2,n/2] with the value we want. If A[n/2,n/2]< *value*,then we know the value need to lie in the rest 3/4 half of the matrix, namely A_{12}, A_{21}, A_{22} so we recurse on the 3/4 portion. If A[n/2,n/2] > *value*, then we know that the value lies in the up-left 1/4 portion of the matrix, that is A_{11}
So in worst case, we could write the recursion relations into $T(n) = 3T(n/2)$, solving this recurrence, we get $T(n^2) = O(n^{\log_2 3})$, which is approximately $O(n^{1.59})$.

Problem 2: Multiplication [10 marks]

(a) (Matrix-vector multiplication) Recursively define a sequence of matrices as follows. Let $H_0 = 1$ and for $k > 0$ let

$$H_k = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}. \tag{1}$$

As such, H_k is of size $n \times n$ where $n = 2^k$. Design an algorithm to compute the matrix-vector multiplication $H_k x$, where $x \in \{0,1\}^n$. You will get full marks if the algorithm is correct and takes $O(n \log n)$ integer multiplication and addition operations.

(b) (Polynomial multiplication) Given two polynomials $P(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$ and $Q(x) = b_n x^n + b_{n-1}x^{n-1} + ... + b_1 x + b_0$, provide an algorithm to multiply these two polynomials in $O(n^{\log_2 3}) = O(n^{1.59})$ word operations (i.e. each addition and multiplication in computing coefficients of $P \cdot Q(x)$ is considered one word operation). You can assume that $n + 1$ is a power of two. State the algorithm clearly and explain its time complexity.

Solution. **Please write down your solution to Problem 2 here.**

(a) We first propose a small problem for matrix-vector multiplication, that when $n \leq 2$, the problem is a small problem that could be solved in constant time. For example, a case like this:

$$H_1 \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

So, we divide the problem into small problems by dividing the dimension of the H_k and x into half. Since

$$H_k = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}, \qquad x \in \{0,1\}^n$$

we denote the product of $H_k x$ as A:

$$A_{[1,1]} = H_{k-1} \cdot x_{upperhalf} + H_{k-1} \cdot x_{lowerhalf}$$
$$A_{[1,2]} = H_{k-1} \cdot x_{upperhalf} + (-H_{k-1}) \cdot x_{lowerhalf}$$

Now we write some pseudo code for this.

```
mv(H,X,n){

    //constant time base case when n=2
    if n == 2{
        C[1,1] = 1*x_{upper half} + 1*x_{lower half};
        C[2,1] = 1*x_{upper half} + (-1)*x_{lower half};
        }
    else{
        mid = n/2;//divide the big problem into half each time
        C[1,1] = mv(H[1,1],X[1,1],mid) + mv(H[1,2],X[2,1],mid);
        C[1,2] = mv(H[2,1],X[1,1],mid) + mv(H[2,2],X[2,1],mid);

        return C;
    }
}
```

In the above algorithm, we divide the original problem into 4 sub problems, we could write the recurrence as

$$T(n) = 4T(n/2) + n$$

we have the operation of n steps at each recurrence call for matrix-vector addition, since vector x is of $n \times 1$. However, solving this recurrence we would get $O(n^2)$.
To improve this, recall that

$$H_k = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}$$

So H_{11},H_{12},H_{21} are essentially the same, H_{22} merely takes the negative value of the formers. So, the recursion call of

```
mv(H[1,1],X[1,1],mid);
mv(H[2,1],X[1,1],mid);
```

and

```
mv(H[1,2],X[2,1],mid);
mv(H[2,2],X[2,1],mid);
```

are redundant function calls.
To simplify, we rewrite our algorithm as

```
        mv(H,X,n){

        //constant time base case when n=2
        if n == 2{
            C[1,1] = 1*x_{upper half} + 1*x_{lower half};
            C[2,1] = 1*x_{upper half} + (-1)*x_{lower half};
            }
        else{
            mid = n/2;//divide the big problem into half each time
            temp1 = mv(H[1,1],X[1,1],mid);
            temp2 = mv(H[1,2],X[2,1],mid);
            C[1,1]=temp1 + temp2;
            C[1,2]=temp1 + (-temp2);
            return C;
        }
    }
```

Now, our recurrence could be written as

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

which is $O(n \log n)$.

(b) We could rewrite $P(x)$ and $Q(x)$ into the form of two terms $P_1 + P_2$ and $Q_1 + Q_2$, where

$$P_1 = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$P_2 = a_{\frac{n}{2}} x^{\frac{n}{2}} + \cdots + a_n x^n$$

$$Q_1 = b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$Q_2 = b_{\frac{n}{2}} x^{\frac{n}{2}} + \cdots + b_n x^n$$

Then,

$$\begin{aligned} P(x) \cdot Q(x) &= (P_1 + P_2) \cdot (Q_1 + Q_2) \\ &= P_1 Q_1 + P_2 Q_1 + P_1 Q_2 + P_2 Q_2 \end{aligned}$$

By the definition of Divide and Conquer, we have now divide the big problem into 4 small problems, with each of the problem size $n/2$, and $O(n)$ time to combine each sub problem, since the combine operation is addition. So

$$T(n) = 4T(n/2) + n$$

This is $O(n^2)$ time complexity.
We observe that the term $P_2Q_1 + P_1Q_2$ could be obtained through

$$(P_1 + P_2) \cdot (Q_1 + Q_2) - P_1Q_1 - P_2Q_2$$

So instead of using 4 sub problems, we could solve it in 3 sub problems.

```
polyMutiply(P(x),Q(x),n){
    P_1 = a_0 + a_1 x + ··· + a_{n/2-1} x^{n/2-1};
    P_2 = a_{n/2} x^{n/2} + ··· + a_n x^n;
    Q_1 = b_0 + b_1 x + ··· + b_{n/2-1} x^{n/2-1};
    Q_2 = b_{n/2} x^{n/2} + ··· + b_n x^n;
    Y(x)=polyMultiply(P_1,Q_1,n/2);
    Z(x)=polyMultiply(P_2,Q_2,n/2);
    U(x)=polyMultiply(P_1+P_2,Q_1+Q_2,n/2);
    return ((U(x)−Y(x)−Z(x))+Y(x)+Z(x));
}
```

In the above algorithm, we divide the problem into 3 sub-problems, each of size $n/2$, now we could write the recurrence as

$$T(n) = 3T(n/2) + n$$

Solving this using recurrence tree method, we would get $T(n) = O(n^{\log_2 3})$

Problem 3: Mergesort [10 marks]

(a) (5 marks) Given four lists of n numbers each, devise an $O(n \log n)$ algorithm to determine if there is any number common to all four lists.

(b) (5 marks) Given an array $A[1, \ldots, n]$, its subarray $A[i, \ldots, j]$ is called a maximal increasing sequence (MIS) if and only if it is sorted and either of the following holds:

- $i = 1$ and $j = n$,
- $i > 1, j = n$, and $A[i-1] > A[i]$,
- $i = 1, j < n$, and $A[j] > A[j+1]$,
- $i > 1, j < n$, and $A[i-1] > A[i], A[j] > A[j+1]$.

Example: in the array $(3, 1, 2, 4, -1, 9, 8)$, there are 4 maximal increasing sequences say $(3), (1, 2, 4), (-1, 9), (8)$.

Consider the following version of mergesort.

```
// merge the sorted arrays B and C into the output array.
int* merge(int B[], int C[]);

int* mergesort2(int A[1,...,n]) {
    int MIS[];
    int i = 1;

    Find j such that A[i,...,j] is a maximal increasing sequence.
    MIS = A[i,...,j];

    while(j+1 <= n) {
        Find k such that A[j+1,...,k] is a maximal increasing sequence.
        MIS = merge(MIS, A[j+1,...,k];
        j = k;
    }

    return MIS;
}
```

What is the running time of `mergesort2` in terms of the array size and the number of maximal increasing sequences in the array?

Solution. **Please write down your solution to Problem 3 here.**

(a) The thought is that we first sort the four lists , and we apply binary search.

```
commonNumber(A,B,C,D){

    //applying merge sort on the four arrays
    sort(A);
    sort(B);
    sort(C);
    sort(D);

    //run a for loop in one of the four array
    for(i=0;i<n;i++){

        //use binary search to find the common element
        bs1=binarySearch(B,0,n-1,A[i]);
        bs2=binarySearch(C,0,n-1,A[i]);
        bs3=binarySearch(D,0,n-1,A[i]);

        //the return value is -1 if the element is not founded
        if(bs1 !=-1 && bs2 !=-1 && bs3 !=-1){
            return True;
        }
    }
}
```

We would apply merge sort for the sorting part, which would be $O(n \log n)$, and the for loop would be $O(n \log n)$, since applying binary search on each of the three arrays is $O(\log n)$, and we combine these operations, we would get

$$O(n \log n)) + O(n \log n) = O(n \log n)$$

(b) we rewrite the algorithm in a recursive pattern

```
merge(int B[],int C[]);

mergesort2(int A[]){
    if(len(A)==1){
        return A;
    }
    else{
        int ptr=0;

        //find MIS
        for(i=0;i<len(A);i++){
            if(A[ptr+1]>=A[ptr]){
                ptr+=1;
            }
        }

        MIS = A[:ptr];
        //recursive call with MIS part excluded
        return merge(MIS,mergesort2(A[ptr+1:]));
    }
}
```

We denote the array size as m and the number of *MIS* as n, the level of the recurrence tree is determined by the number of *MIS* we found. I attached the recurrence tree I draw below, **figure 1**. We denote the *MIS* we found each time with size k_i, and we could write the recurrence into the form

$$T(m) = T(m - k_1) + n$$

as the graph shows, we solve the recurrence and get

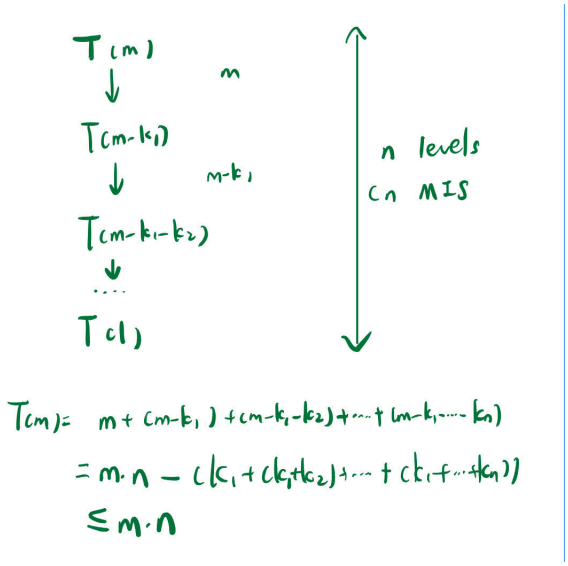

Figure 1: 3.b

$$T(n) \leq m \cdot n$$

which is

$$T(n) = O(mn)$$

Problem 4: Counting inversions [15 marks + 10 marks]

(a) (5 marks) There are n students and each student i has 2 scores x_i and y_i. Students i and j are friends if and only if $x_i < x_j$ and $y_i > y_j$. Devise an $O(n \log n)$ algorithm to count the number of pairs of friends.

(b) (10 marks) Let there be an array of 2D pairs $\big((x_1, y_1), \ldots, (x_n, y_n)\big)$. With a fixed constant y' a pair (i, j) is called *half-inverted* if $i < j$, $x_i > x_j$, and $y_i \geq y' > y_j$. Devise an algorithm that counts the number of half-inverted pairs. You will get full marks if your algorithm is correct of complexity no more than $O(n \log n)$.

(c) (**Bonus**: 10 marks) A pair (i, j) is inverted if $i < j, x_i > x_j$, and $y_i > y_j$. Devise an algorithm as efficient as possible that counts the number of inverted pairs.

Solution. **Please write down your solution to Problem 4 here.**

(a) The idea is that we first sort the array of (x_i, y_i) according to the value of x_i, then we count the inversion pair according to the value of y_i.

```
mergecount(A,B){
    ptr1=0;
    ptr2=0;
    count=0;
    new array L;
    while(ptr1<len(A) && ptr2<len(B)){
        if(y_ptr1 > y_ptr2){

        \\according to the sorted property of y_i
        \\ we could know that all pair after (x_ptr1,y_ptr1)
        \\ are friends with (x_ptr2,y_ptr2)
            count += len(A)-ptr1;
            add B[ptr2] to L;
            ptr2 ++;
        }
        if(y_ptr1 > y_ptr2){
            count += len(B)-ptr2;
            add A[ptr1] to L;
            ptr1 ++;
        }
    }
    if(ptr1 == len(A)-1){
        add the rest elements in B to L;
    }

    if(ptr2 == len(B)-1){
        add the rest elements in A to L;
    }
```

```
    return (count,L);
};


sortAndCount(L){
    if(len(l)==1){
        return L;
    }
    else{
        divide L into half;
        (r_A,A)← sortAndCount(A);
        (r_B,B)← sortAndCount(B);
        (r_L,L)← mergecount(A,B);
        return (r_A + r_B + r_L,L);
    }

}

\\main function
friend(L){
    sort(L);\\sort L according to x value
    sortAndCount(L);
}
```

The idea is that, having the x_i already in ascending order, we do a enhanced merge sort to get the number of friends in the array, given the definition of a friend pair $x_i < x_j$ **and** $y_i > y_j$. We could write the recurrence into

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

which is $O(n \log n)$.

(b) The idea is that, since we are given the requirement of an additional y', we could use this as a key to simplify our strategy.
As we usually do in mergesort, there will be two part of recursion, we denote them as A_1 and A_2, A_1 is the part where all the elements in it has their index smaller than that of elements in A_j. So what we want to achieve is to find the half-inverted pair in $O(n)$ time, so we would expect the return from recursion A_1 only contains those pair whose y value are **greater or equal than** y', and the return from recurstion A_2 only conatins those whose y are **less than** y', and **both return array are sorted in ascending order according to their** x **value**. The benefit of this is that we do not have to worry about y value, we only look at x value.
Now let's write some pseudo code.

```
mergecount(A,B,y'){
    ptr1=0;
    ptr2=0;
    counter=0;
    new Array L;
    length1 = len(A);
    length2 = len(B);

    while(ptr1<length1 && ptr2<length2){
```

```
            if(A[ptr1]<B[ptr2]){
                add A[ptr1] to L;
                ptr1 ++;
            }
            else if(A[ptr1]>B[ptr2]){
                if(A[ptr1][1]>y' && A[ptr1][1] > B[ptr2[1]]){
                    counter += (length-ptr1);
                }
                add B[ptr2] to L;
                ptr2 ++;
            }
            }

            else if(ptr1 == length1-1){
                add the rest of B to L;
            }
            else if(ptr2 == length2-1){
                add the rest of A to L;
            }

        //Now we have obtained our L with x in ascending order
        m = len(L);
        new Array M;//contain element whose y >= y'
        new Array N; // contain element whose y < y'

        for(i=0;i<m;i++){
            if(L[i][1]>=y'){
                add L[i][1] to M;
            }
            else{
                add L[i][1] to N;
            }
        }
        return (counter,M,N);

    sortAndCount(A,y'){
        if(len(A)==1){
            return A;
        }
        else{
            divide A into half;
            (c1,M1,N1)←sortAndCount(A1,y');
            (c2,M2,N2)←sortAndCount(A2,y');
            (c3,M3,N3)←mergecount(M1,N2,y');
            return(c1+c2+c3,M3,N3);
            }

    }
}
```

Problem 5: Randomized binary search [Bonus: 5 marks]

In the binary search algorithm, it is typical to take the middle element of the (sub-)array of length n as the pivot in each iteration. We can also choose uniformly at random an element in the (sub-)array as our pivot. Assume that randomly choosing such a pivot takes $\Theta(1)$ time. Prove that the expected running time of the randomized binary search algorithm is $O(\log n)$.

Solution. **Please write down your solution to Problem 5 here.**
Since we are randomly picking pivot everytime, in an array of length n, the probaility of each index being picked is $\frac{1}{n}$. Now we could write the recurrence relationship as

$$T(n) = p \cdot T(1) + p \cdot T(2) + \cdots + p \cdot T(n-1) + 1$$

the recurrence part is the mathematical expectation of the sub-problem, the 1 is the time to randomly pick a pivot.Now let's proceed with this relationship.

$$T(n) = \frac{1}{n}(T(1) + \cdots + T(n-1)) + 1$$
$$T(n-1) = \frac{1}{n-1}(T(1) + \cdots + T(n-2)) + 1$$

Taking away the fraction part of both equations, we get

$$nT(n) - (n-1)T(n-1) = T(n-1) + 1$$

which is

$$T(n) = T(n-1) + \frac{1}{n}$$

We recursively write this relationship into the form

$$T(n) = \frac{1}{n} + \frac{1}{n-1} \cdots + \frac{1}{2} + 1$$

So $T(n)$ is the nth harmonic number, which gives us

$$T(n) = \Theta(\log n)$$

.