# Assignment 1
# Algorithm Design and Analysis

bitjoy.net

October 7, 2015

I choose problem 1,3,4,7,8.

## 1 Divide and Conquer

### 1.1 Algorithm in natural language

Assume the two databases are A and B, A(i) and B(i) are the $i^{th}$ smallest value each contains.

First we compare the median of A and B. Let $k = \lfloor n/2 \rfloor$, so A(k) and B(k) are the median of A and B. If A(k) < B(k) (as all values are distinct, no A(k) == B(k) case; the case A(k) > B(k) is the same if we exchange A and B), then the median of combined 2n values must be in A[k,n] or B[1,k]. Because A(k) is greater than the first $k-1$ elements in A, B(k) > A(k), so the last $k$ elements of B are also greater than the first $k-1$ elements of A. As a result, the median of 2n values couldn't lie in A[1,k-1], neither B[k+1,n].

So, take A' = A[k,n] as the new A, B' = B[1,k] as the new B, but we can't delete the databases, the $i^{th}$ smallest value in A' is the $(i+k)^{th}$ smallest value in A, the $i^{th}$ smallest value in B' is the $i^{th}$ smallest value in B, recursively we will get the median of combined 2n values.

### 1.2 Algorithm in pseudo-code

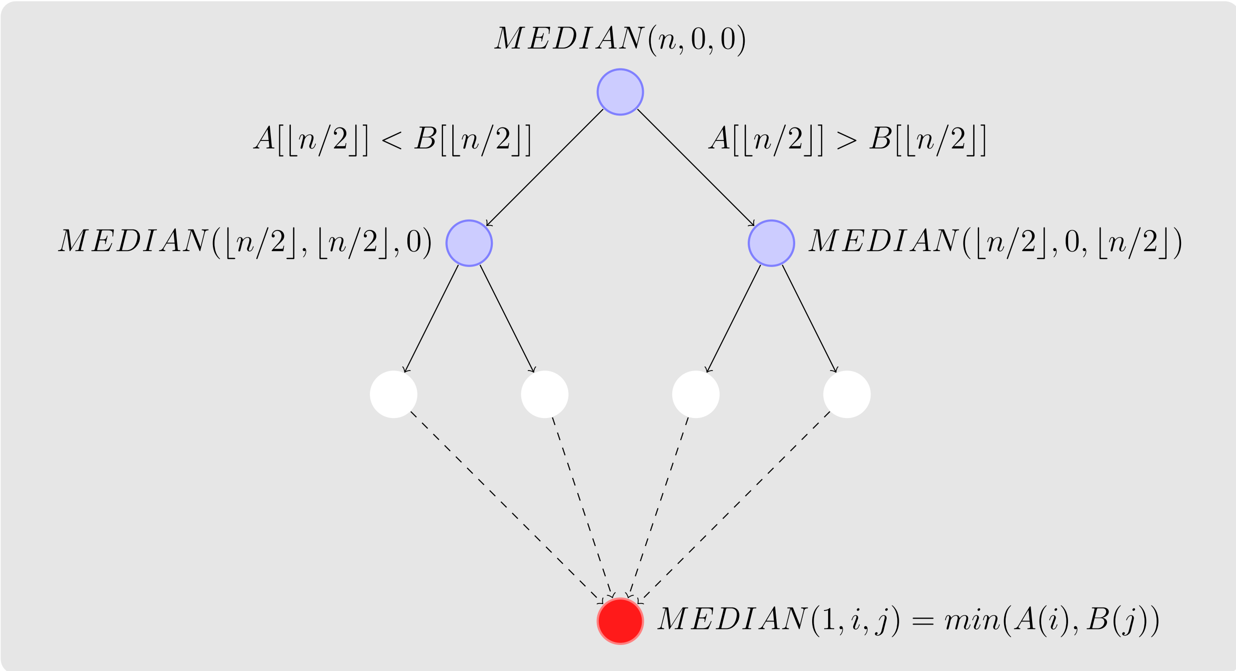
We define algorithm MEDIAN(n,a,b) that input integers n,a and b and output the median of the union of the two parts A[a+1,b+n] and B[b+1,b+n].

MEDIAN($n, a, b$)

```
if n == 1
    return min(A(a+k),B(b+k))
k = ⌊n/2⌋
if A(a + k) < B(b + k)
    return MEDIAN(k, a + k, b)
else
    return MEDIAN(k, a, b + k)
```

To find the median of 2n elements in A and B, we just call MEDIAN(n,0,0).

### 1.3 Subproblem reduction graph




### 1.4 Correctness of the algorithm

We can use ***loop invariant*** to prove it.

**Initialization:** At the beginning, we call MEDIAN(n,0,0) to find the median of the union of A[1,n] and B[1,n], say it's $M_1$. As described in the section 1.1, we know $M_2 = MEDIAN(\lfloor n/2 \rfloor, \lfloor n/2 \rfloor, 0)$, the median of the union of $A[\lfloor n/2 \rfloor, n]$ and $B[1, \lfloor n/2 \rfloor]$, is also the median of the union of A[1,n] and B[1,n], that's to say $M_2 == M_1$.

**Maintenance:** We take $A' = A[\lfloor n/2 \rfloor, n]$ as the new A, $B' = B[1, \lfloor n/2 \rfloor]$ as the new B, after calling $M_3 = median(\lfloor n/4 \rfloor, \lfloor n/4 \rfloor, 0)$, we can say $M_3 == M_2$, so $M_3 == M_1$.

**Termination:** After calling MEDIAN m(m is big enough) times, only A(i) and B(j) left, the median of them is $M_m = min(A(i), B(j))$, as $M_m == M_{m-1} == ... == M_1$, $M_m$ is the final global median.

### 1.5 Complexity of the algorithm

Let T(n) be the number of queries asked by our algorithm, each time we call the function, we ask 2 queries in line 4, after that, half elements are "eliminated", so we have $T(n) = T(\lfloor n/2 \rfloor) + 2$. Therefore $T(n) = 2\lfloor logn \rfloor = O(logn)$.

## 3 Divide and Conquer

### 3.1 Algorithm in natural language

In the textbook, we can find the inversions while merging two sub-sorted array, because their conditions are the same. When $i < j$ and $a_i > a_j$, $a_j$ should be in front of $a_i$ and $(a_i, a_j)$ is a inversion, so we do two things in one merge. But when counting *significant inversions*, we can't do these in the same time, because $a_i > a_j$ doesn't mean $a_i > 3a_j$.

Don't worry, we can do it in two merges, one for sorting, one for finding *significant inversions*.The new algorithm is very similar to the one in textbook, so let's go straight to section 3.2 to see the pseudo-code.

## 3.2 Algorithm in pseudo-code

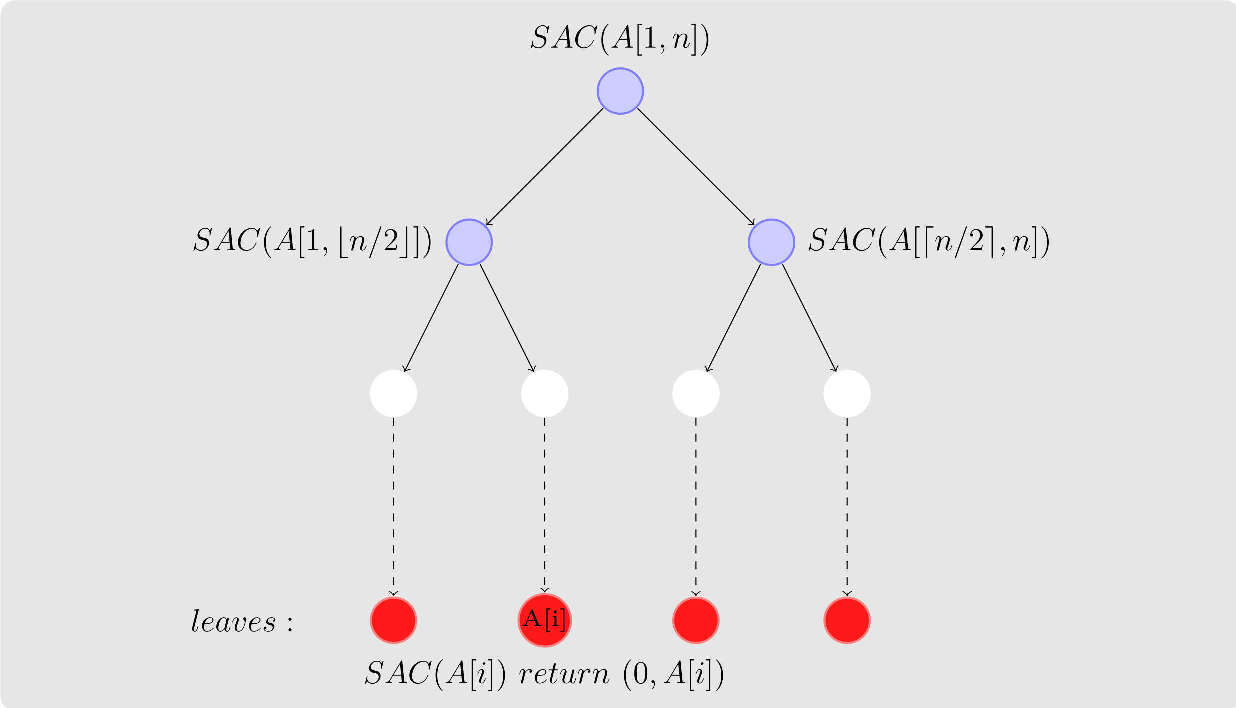
We define algorithm SORT-AND-COUNT(A) that input an unsorted array A and output the number of *significant inversions* in original A and the sorted array A.

```
MERGE-AND-COUNT(L, R)
RC = 0; i = 0; j = 0;
for k = 0 to ||L|| + ||R|| − 1
    if L[i] > R[j]
        A[k] = R[j]
        j++
    else
        A[k] = L[i]
        i++
i = 0; j = 0;
for k = 0 to ||L|| + ||R|| − 1
    if L[i] > 3R[j]
        RC = RC + length[L] - i
        j++
    else
        i++
return (RC,A)
```

```
SORT-AND-COUNT(A)
if A has one element
    return (0,A)
else
    Divide A into two sub-sequences L and R
    (RC_L,L) = SORT-AND-COUNT(L)
    (RC_R,R) = SORT-AND-COUNT(R)
    (C,A) = MERGE-AND-COUNT(L,R)
    return (RC = RC_L + RC_R + C,A)
```

## 3.3 Subproblem reduction graph

SAC(A) is short for SORT-AND-COUNT(A).




### 3.4 Correctness of the algorithm

As we can see in section 3.2, the new MERGE-AND-COUNT just add an extra merge based on the old MERGE-AND-COUNT in the textbook, so it is obvious that new algorithm can sort array correctly.

As for finding all *significant inversions*, suppose we are going to merge $L[1, n_1]$ and $R[1, n_1]$ which are already sorted. If $L[i] > 3R[j]$, then $(L[i], R[j])$ is a *significant inversion*, as all $L[i+1, n_1]$ is greater than $L[i]$, so $L[i+1, n_1]$ together with $R[j]$ are *significant inversions* too. Thus, the number of *significant inversions* is $n_1 - i$.

So, recursively we can find all *significant inversions*.

### 3.5 Complexity of the algorithm

Let T(n) be the time of my algorithm, as there are an extra merge in the new MERGE-AND-COUNT algorithm, so the MERGE-AND-COUNT time is O(2n), we get $T(n) = 2T(n/2) + O(2n)$, thus $T(n) = O(nlgn)$.

## 4 Divide and Conquer

### 4.1 Algorithm in natural language

Given the complete binary tree T, let t be the root of T, $t_L$ and $t_R$ be the left and right child of t.

If $t < t_L$ and $t < t_R$, t is one of *local minimum* node; if not, choose one of child that less than t, say $t_L$(or $t_R$), check if $t_L$'s 2 children are less than t, if so, $t_L$ is the *local minimum* node; if not, recursively check one of $t_L$'s child.

If we can't find a *local minimum* node among T's internal nodes, assume we reach X which is greater than its left child $X_L$, and $X_L$ is a leaf node, as a result, $X_L$ is the *local minimum* node.

So, we can always find a *local minimum* node.

## 4.2 Algorithm in pseudo-code

We define algorithm FIND-LOCAL-MINIMUM(X) that input a node X and output one of *local minimum* node in X's subtree.

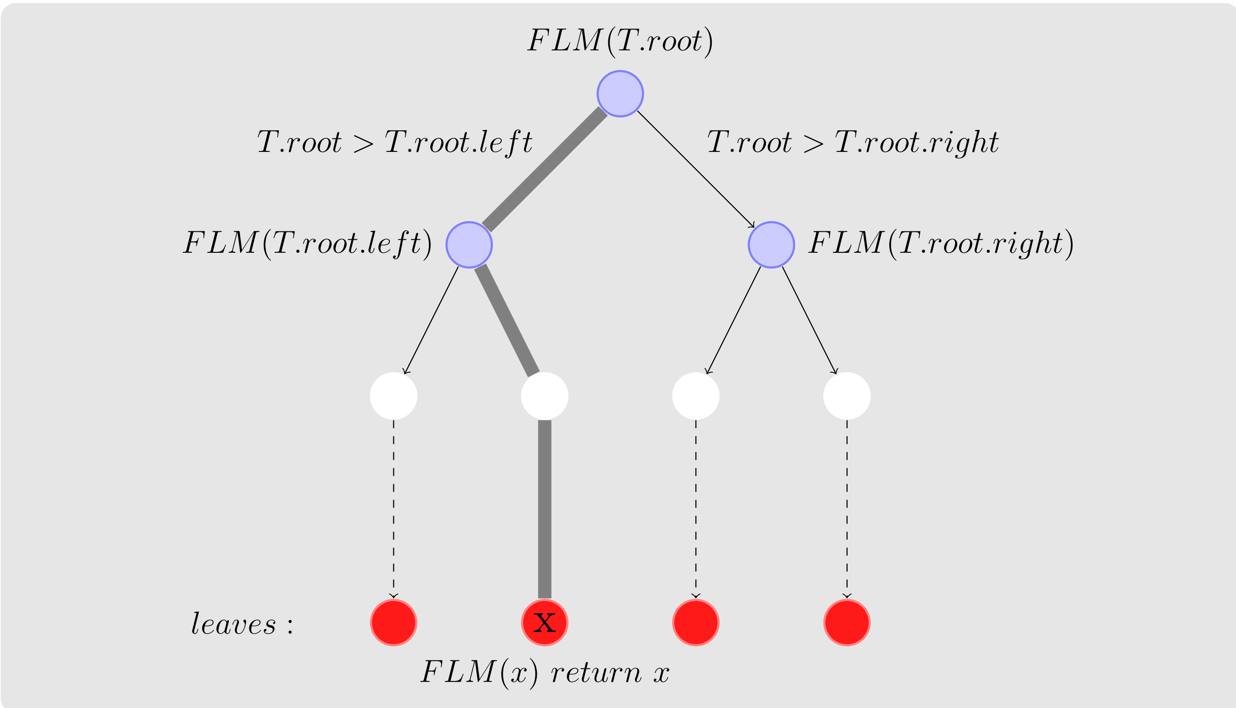
FIND-LOCAL-MINIMUM($X$)

```
if X has children
    Let X_L and X_R be the left and right child of X
    if X < X_L and X < X_R
        return X
    elseif X > X_L
        return FIND-LOCAL-MINIMUM(X_L)
    elseif X > X_R
        return FIND-LOCAL-MINIMUM(X_R)
else
    return X
```

To find the *local minimum* node of T, we just call FIND-LOCAL-MINIMUN(T.root).

## 4.3 Subproblem reduction graph

FLM(X) is short for FIND-LOCAL-MINIMUM(X).



### 4.4 Correctness of the algorithm

We can use ***loop invariant*** to prove it.

**Initialization:** At the beginning, if $T.root < T.root.left$ and $T.root < T.root.right$, T itself is a *local minimum* node.

**Maintenance:** Otherwise, claim that at any point in the execution of the algorithm, the parent (if any) of X has a greater value than X itself. Thus, X only need to compare with $X_L$ and $X_R$, if $X < X_L$ and $X < X_R$, X is a *local minimum* node, if not, go to line 6 or line 8 recursively.

**Termination:** If algorithm doesn't return among internal nodes, it reaches leaf node X, so the parent of X has a greater value than X, thus, X is a *local minimum* node.

So, the algorithm can always find a *local minimum* node.

### 4.5 Complexity of the algorithm

As we can see in the section 4.3, each time we go to one of X's subtree and do 3 probes, as the longest path is the height of the tree, say $log_2 n$. so we do at most $3log_2 n$ probes, so the complexity is $O(logn)$.

## 7 Divide and Conquer

### 7.1 Implementation of the Sort-and-Count algorithm

I implemented the Sort-and-Count algorithm in Python3.

```python
# -*- coding: utf-8 -*-

import time

INF = 100001
inversions = 0

def MergeAndCount(A, p, q, r):
    global INF
    global inversions
    L = A[p:q+1]
    L.append(INF) #add a sentinel card
    R = A[q+1:r+1]
    R.append(INF) #add a sentinel card
    i = 0
    j = 0
    for k in range(p, r + 1):
        if L[i] < R[j]:
            A[k] = L[i]
            i = i + 1
        else:
            A[k] = R[j]
            j = j + 1
            if L[i] != INF:
                inversions = inversions + len(L) - i - 1

def SortAndCount(A, p, r):
    if p < r:
        q = int((p + r) / 2)
        SortAndCount(A, p, q)
```

```
        SortAndCount(A, q + 1, r)
        MergeAndCount(A, p, q, r)

if __name__ == "__main__":
    Q5 = open('Q5.txt', encoding = 'utf-8')
    data = [ int(x) for x in Q5 ]
    Q5.close()
    start = time.clock()
    SortAndCount(data, 0, len(data) -1 )
    end = time.clock()
    print("number of inversions:%d\ntime:%f s"%(inversions,end-start))
```

The number of inversions in Q5.txt is 2500572073, running time is 1.658 s

## 7.2 Quick-Sort version

Yes! Quick-Sort can also count inversions. Typical Quick-Sort is unstable, so it can't count inversions, once we make it stable, it can.

```
# -*- coding: utf-8 -*-

import time

inversions = 0

def Partition(A, p, r):
    global inversions
    tmp = [0] * (r-p+1)
    pivot = A[p]
    k = 0
    for i in range(p+1, r+1):
        if A[i] < pivot:  #less than pivot
            tmp[k] = A[i]
            inversions = inversions + i - k - p
            k = k + 1
    tmp[k] = pivot
    ans = k + p
    k = k + 1
    for i in range(p+1, r+1):
        if A[i] > pivot:  #greater than pivot
            tmp[k] = A[i]
            k = k + 1
    k = 0
    for i in range(p, r+1): #copy back
        A[i] = tmp[k]
        k = k + 1
    return ans

def QuickSortAndCount(A, p, r):
    if p < r:
        q = Partition(A, p, r)
        QuickSortAndCount(A, p, q-1)
        QuickSortAndCount(A, q + 1, r)

if __name__ == "__main__":
    Q5 = open('Q5.txt', encoding = 'utf-8')
    data = [ int(x) for x in Q5 ]
    Q5.close()
    start = time.clock()
```

```
    QuickSortAndCount(data, 0, len(data) -1 )
    end = time.clock()
    print("number of inversions:%d\ntime:%f s"%(inversions,end-start))
```

The number of inversions in Q5.txt is 2500572073, running time is 2.266 s. it is slower than Merge-Sort version, although the complexity is still $O(nlgn)$, it has to scan the array 3 times in Partition step, and it isn't a in-place sort.

# 8 Divide and Conquer

Here is my implementation of Find-Closest-Pair in Python3.

```
# -*- coding: utf-8 -*-
"""
Created on Tue Oct  6 16:09:40 2015

@author: czl
"""
import copy
import math

INF = 10000000 # max of the **square** of the distance

class Point:
  x = 0
  y = 0
  def __init__(self, x, y):
    self.x = x
    self.y = y

# calculate the square of the distance of point i and point j
def GetDistanceSquare(i, j):
  return (i.x - j.x) * (i.x - j.x) + (i.y - j.y) * (i.y - j.y)

def FindClosestPair(s, e):
    global INF
    if e - s < 3: # if less than 3 points, just brute force
        local_min1 = [INF,0,0]
        for i in range(s, e):
            for j in range(i + 1, e + 1):
                if GetDistanceSquare(dataX[i], dataX[j]) < local_min1[0]:
                    local_min1[0] = GetDistanceSquare(dataX[i], dataX[j])
                    local_min1[1] = dataX[i]
                    local_min1[2] = dataX[j]
        return local_min1
    else: # else divide and conquer
        m = int((s + e) / 2)
        l = FindClosestPair(s, m)
        r = FindClosestPair(m + 1, e)
        local_min2 = []
        if l[0] < r[0]:
            local_min2 = copy.deepcopy(l)
        else:
            local_min2 = copy.deepcopy(r)

        Y = []
        median = dataX[m]

```

```
        # collect points within the 2local_min2[0] strip
        # already sorted by y
        for i in dataY:
            if i.x >= median.x - local_min2[0] and i.x <= median.x + local_min2[0]:
                Y.append(i)
        for i in range(0, len(Y)):
            for j in range(i + 1, i + 7): # only calculate next 7 points
                if j >= len(Y):
                    break
                if GetDistanceSquare(Y[i], Y[j]) < local_min2[0]:
                    local_min2[0] = GetDistanceSquare(Y[i], Y[j])
                    local_min2[1] = Y[i]
                    local_min2[2] = Y[j]
        return local_min2

if __name__ == "__main__":
    Q8 = open('Q8.txt', encoding = 'utf-8')
    dataX = []
    for line in Q8:
        v = line.split()
        dataX.append(Point(int(v[0]), int(v[1])))
    Q8.close()
    dataY = copy.deepcopy(dataX)
    dataX.sort(key = lambda p: p.x) # pre-sorted by x
    dataY.sort(key = lambda p: p.y) # pre-sorted by y
    ans = FindClosestPair(0, len(dataX) - 1)
    print('The Closest Pair is (%d,%d)--(%d,%d)\nThe distance is %f'%(ans[1].x, ans[1].y, ans[2].x, ans[2].y, math.sqrt(ans[0])))
```

Example input(Q8.txt, each line has 2 numbers indicate a point (x,y)):
43 67
82 35
81 37
70 98
71 94
24 61
21 34
5 2
67 29
42 76

Example output:
The Closest Pair is (81,37)–(82,35)
The distance is 2.236068

Let $T(n)$ be the running time of each recursive step and $T'(n)$ be the running time of the entire algorithm. At the beginning, we sort the data by x and by y, so $T'(n) = T(n) + O(nlgn)$, and

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{if } n > 3 \\ O(1) & \text{if } n \leq 3 \end{cases}$$

Thus, $T(n) = O(nlgn)$ and $T'(n) = O(nlgn)$.