# CSE 60111, Complexity and Algorithms

## Spring Semester, 2013

## Solution 4

**1.**

**Ideas:** We combine the idea of divide and conquer and prune and search to solve this problem. First, we select the middle element $k_{\lfloor \frac{h}{2} \rfloor}$ among the k integers $k_1$, $k_2$, …, $k_h$. Second, we select the $k_{\lfloor \frac{h}{2} \rfloor}$ smallest element $a_x$ from S, and use $a_x$ to divide S into two subsets S1 and S2. We know that all the $k_i$ ($i<\lfloor \frac{h}{2} \rfloor$) smallest element in S can be found only in S1; and all the $k_j$ ($j>\lfloor \frac{h}{2} \rfloor$) smallest element in S can be found only in S2. So we can recursive solve the multiple selection problem on S1 and S2.

**Pseudo-code:**

***MultipleSelection****(S, $k_1$, $k_2$, …, $k_h$)*

***Input:*** *S: a set of n real numbers $a_1$, $a_2$, …, $a_n$.*
*$k_1$, $k_2$, …, $k_h$: the rank of the elements that should be selected from S*

***Output:*** *the elements of S whose ranks are given by $k_1$, $k_2$, …, $k_h$ .*

***Begin***

$k_{\lfloor \frac{h}{2} \rfloor}$ = ***Selection****($k_1$, $k_2$, …, $k_h$, $\lfloor \frac{h}{2} \rfloor$);*

$a_x$ = ***Selection****(S, $k_{\lfloor \frac{h}{2} \rfloor}$);*

*Divide S into two sets according to $a_x$, such that all elements in S1 are smaller than $a_x$, and all elements in S are larger than $a_x$.*

*K1 = {$k_i$ | $i<\lfloor \frac{h}{2} \rfloor$}, and K2 = {$k_i$ | $i>\lfloor \frac{h}{2} \rfloor$}.*

***MultipleSelection****(S1, K1);*
***MultipleSelection****(S2, K2);*

***End***

**Time Complexity**: According to the divide and conquer process, we have the following recurrence relation:

T(n,k)=T(n1,k1)+ T(n2,k2)+ an= O(nlogk), where n1+n2 =n, and k1+k2 = k.

You can use guess and verification to see this is true.

**Correctness**: We can see using an appropriate element to probe can narrow the searching space.

**2.**

**a.**

**Ideas:** We use the idea of prune and search. To be specific, first we find the median element x in S. Second, we compute the sum of weights for all elements that are smaller than x, and denote the value of the sum as Ws. If $Ws < \frac{1}{2}$ $and$ $Ws + w_x \geq 1/2$, we know that x is the weighted median. If $Ws < \frac{1}{2}$ $and$ $Ws + w_x < 1/2$, we know x cannot be the weighted, and consequently, any element that is smaller than x cannot be the weighted median either, this means we can prune away these elements, and recursively find our target element in the remaining data. If $Ws \geq \frac{1}{2}$, we know the weighted median should be smaller than x, and we can also prune away the elements that are larger than x.

**Pseudo-code:**

*y =* ***WeightedSelection****(x1, x2, …, xn, w1, w2,…, wn, W)*

***Input:*** *x1, x2, …, xn: n elements.*
*w1, w2,…, wn: the weight of each element.*
*W: the target weight, should be initialized as half of the sum of weights for all elements.*

***Output****: y: the element among x1, x2, …, xn, such that* $\sum_{x_i<y} w_i < W$ $and$ $\sum_{x_i \leq y} w_i \geq W$

***Begin***

*x =* ***Selection****(x1, x2, …, xn, n/2);*
*Ws =* $\sum_{x_i<x} w_i$*;*
***If*** $Ws < W$ $and$ $Ws + w_x \geq W$
***Return*** *x;*
***Else if*** $Ws < W$
*S1 =* $\{x_i \mid x_i > x\}$*;*
*W1 =* $\{w_i \mid x_i > x\}$*;*
***WeightedSelection****(S1, W1, W -* $\sum_{x_i \leq x} w_i$*);*
***Else***
*S2 =* $\{x_i \mid x_i \leq x\}$*;*
***WeightedSelection****(S2, W2, W);*
***End If***

***End***

**Time Complexity**: T(n) = T(n/2) + an = O(n)

**Correctness**: Pay attention to what target weight should be used as input, since it is not always half of the sum of all weights as we pruning data away.

**b.**

**Ideas:** Since the objective function we want to minimize in 2-D case can be decomposed into the sum of two objective functions in 1-D with respect to each of the 2 dimensions. And a key observation is the objective function in 1-D can be minimized by selecting the weighted median point along the 1-D dimension as the target point p. We use the case in the x-coordinate as an example to illustrate this.

Specifically, suppose the weighted median in the x-coordinate is x*, and the optimal

selection point is x, we will prove x cannot be different from x* by comparing the two resulted values by substituting x* and x into our objective function respectively. The difference between these two values is

$$D = \sum_{i=1}^{n} w_i|x - x_i| - \sum_{i=1}^{n} w_i|x^* - x_i|$$

$$= \sum_{i=1}^{n} w_i(|x - x_i| - |x^* - x_i|)$$

If x < x*,

$$D = \sum_{i=1}^{n} w_i(|x - x_i| - |x^* - x_i|)$$

$$= \sum_{x_i<x^*} w_i(|x - x_i| - |x^* - x_i|) + \sum_{x_i\geq x^*} w_i(|x - x_i| - |x^* - x_i|)$$

$$= \sum_{x_i<x^*} w_i(|x - x_i| - |x^* - x_i|) + \sum_{x_i\geq x^*} w_i(x^* - x)$$

$$\geq \sum_{x_i<x^*} w_i(x - x^*) + \sum_{x_i\geq x^*} w_i(x^* - x)$$

$$= (x^* - x)\left(\sum_{x_i\geq x^*} w_i - \sum_{x_i<x^*} w_i\right)$$

$\geq 0$

Similarly, if x > x*, we can also see D is not smaller than zero. This means the optimal solution x can only be x*.

**Pseudo-code:**

*p = **PostOfficeLocation2D**(p1, p2, …, pn, w1, w2, …, wn)*

***Input**: p1, p2, …, pn: n 2-D points*

*w1, w2, …, wn: weights of points*

***Output**: p: the point we select such that the objective function is minimized*

***Begin***

*x = **WeightedSelection**(p1.x, p2.x, …, pn.x, w1, w2, …, wn, $\frac{1}{2}\sum_{i=1}^{n} w_i$);*

*y = **WeightedSelection**(p1.y, p2.y, …, pn.y, w1, w2, …, wn, $\frac{1}{2}\sum_{i=1}^{n} w_i$);*

*p.x = x;*

*p.y = y;*

***Return** p;*

***End***

**Time Complexity**: Linear time.

**Correctness**: See the ideas part.

**3.**

**a.**

**Ideas**: We consider two cases with respect to i. If $i \geq n/2$, we just use the standard selection algorithm. Otherwise, we do the following. First, we pair up the elements and compare the two elements within each pair. Second, suppose P is a set of elements consisting of the smaller element of each pair. We recursively solve our selection problem on P (namely, selecting the i-th smallest element from M). Say x is returned. Third, we take all such pairs of elements that the smaller element in the pair is not larger than x. Actually, we will have i such pairs. Finally, we recursively solve our selection problem on the i pairs of elements.

**Pseudo-code:**

```
x = AnotherSelection(a1, a2, …, an, i)
Input:  a1, a2, …, an: n elements
        i: the order of the target element
Output:x: the i-th smallest element
Begin
    If i ≥ n/2
        x = Selection(a1, a2, …, an, i);
        Return x;
    End If
    Pair up the elements;
    Initialize the array P;
    For each pair of elements (a, b)
        P.add(min(a, b));
    End For
    x = AnotherSelection(P, i);
    Initialize the array Q;
    For each pair of elements (a, b)
        If min(a, b) < x
            Q.add(a);
            Q.add(b);
        End If
    End For
    x = Selection(Q, i);
    Return x;
End
```

Number of Comparisons used by the algorithm AnotherSelection consists of three parts, if $i < n/2$. First, comparing each pair of elements takes $O(\lfloor n/2 \rfloor)$ time. Second, recursively solving the problem on the set P takes $U_i(\lceil n/2 \rceil)$ time, since we don't know whether i is smaller than half of $\lceil n/2 \rceil$ (the size of our sub-problem) or not. Finally, recursively solving the problem on the set P takes $T(2i)$ time, since we know for sure i is not smaller than half of 2i (the size of our sub-problem).

**Correctness**: Pay attention to the relation between i and the size of the sub-problems.

**b.**

If $i < \frac{n}{2}$, we have

$$U_i(n) = \left\lfloor \frac{n}{2} \right\rfloor + U_i\left(\left\lceil \frac{n}{2} \right\rceil\right) + T(2i)$$

$$= \left\lfloor \frac{n}{2} \right\rfloor + \left(\left\lfloor \frac{n}{4} \right\rfloor + U_i\left(\left\lceil \frac{n}{4} \right\rceil\right) + T(2i)\right) + T(2i)$$

$$= \cdots$$

$$= n + O(T(2i)\log(n/i))$$

**c.**

If i is a constant smaller than n/2,

$$U_i(n) = n + O\left(T(2i)\log\left(\frac{n}{i}\right)\right)$$

$$= n + O(logn)$$

**4.**

**Ideas**: The ideas are basically the same as that for solving the standard 2-D linear programming problem discussed in class, except that now we have the property that the vertices of common intersection area within each group are in sorted order now, and we can use this property to speed up our algorithm for solving the standard 2-D linear programming problem in three places.

First, we can easily separate the upper half-planes and lower half-planes within each group in O(log(n/g)) time.

Second, we can use the selection in sorted arrays algorithm (which takes O(glog(n/g)) time) to select the intersection point we want to use to probe, instead of using the general selection algorithm (which takes O(n) time).

Third, once we know the probe element, since within each group, the vertices of common intersection area within each group are already in sorted order, we can easily find the probe element in each group in O(log(n/g)) time, and prune away the unnecessary data.

**Pseudo-code:**

***SortedGroups2DLinearProgramming****(A1, A2, …, Ag)*

***Begin***

> ***%*** *Note that you should separate the upper half-planes and lower half-planes, and address these two types of half plans separately as discussed in class, but more efficiently as described above. We only give a high level description here to let you what we are doing differently now.*
>
> *X =* ***SelectionInSortedArrays****(A1, A2, …, Ag, n/2);*
>
> ***%*** *Data pruning is performed within each group as described above.*
>
> *[A1', A2', …, Ag'] =* ***DataPrune****(A1, A2, …, Ag, X);*
>
> ***SortedGroups2DLinearProgramming****(A1', A2', …, Ag');*

***End***

**Time Complexity:** We can write our recurrence relation as follows.

T(n) = T(n/b) + O(glog(n/g)), where b > 1, and is a constant, because we can always prune some portion of the data away.

We can stop the recursive process until we have a constant number of elements within each group, this means we have to do O(log(n/g)) times recurrence. So our total running time is $O(g \times \left(\log\left(\frac{n}{g}\right)\right)^2)$.

**Correctness**: We essentially follow the same process for solving the standard 2-D linear programming problem, with just a few changes to speed up the process by using the property that intersection vertices within each group are already sorted.