Homework #4

(Due: Dec 14)

Group Number: 19

Group Members		
Name	SBU ID	% Contribution
Kalantri Chaitanya	111446728	33.33
Karhadkar Rohan	111406429	33.33
Paratkar Neel	111483570	33.33

Collaborating Groups

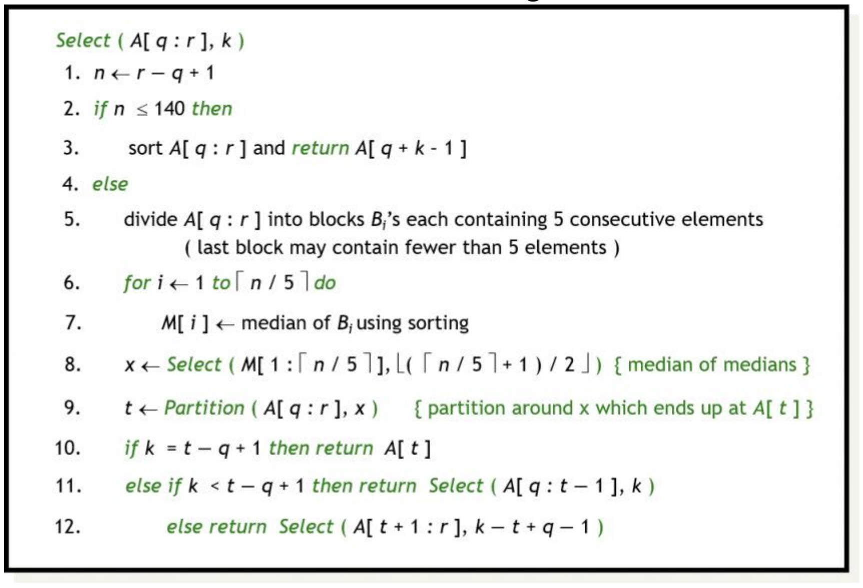
Group Number	Specifics (e.g., specific group member? specific task/subtask?)
52	Question 3 - Neha

External Resources Used

	Specifics (e.g., cite papers, webpages, etc. and mention why each was used)
1.	http://math.mit.edu/~goemans/18434S06/setcover-tamara.pdf
2.	https://people.seas.harvard.edu/~yaron/AM221-S16/lecture_notes/AM221_lecture21.pdf
3.	http://www.geeksforgeeks.org/set-cover-problem-set-1-greedy-approximate-algorithm/
4.	
5.	

1 (a)

This is the normal deterministic select algorithm.



```
Select ( A[ q : r ], k )
 1. $n \leftarrow r - q + 1$
 2. if $n \leq 140$ then
 3.     sort A[ q : r ] and return A[ q + k - 1 ]
 4. else
 5.     divide A[ q : r ] into blocks $B_i$'s each containing 5 consecutive elements
            ( last block may contain fewer than 5 elements )
 6.     for $i \leftarrow 1$ to $\lceil n / 5 \rceil$ do
 7.         $M[ i ] \leftarrow$ median of $B_i$ using sorting
 8.     $x \leftarrow$ Select ( $M[ 1 : \lceil n / 5 \rceil ], \lfloor ( \lceil n / 5 \rceil + 1 ) / 2 \rfloor$ ) { median of medians }
 9.     $t \leftarrow$ Partition ( A[ q : r ], x )    { partition around x which ends up at A[ t ] }
10.     if $k = t - q + 1$ then return A[ t ]
11.     else if $k < t - q + 1$ then return Select ( A[ q : t − 1 ], k )
12.         else return Select ( A[ t + 1 : r ], k − t + q − 1 )
```

It has been modified slightly to be able to run it in parallel as shown below:

```
Par-Select (A[q : r ], k)
1. n <- r - q + 1
2. if n <= 140 then
3.         sort A[q : r] and return A[q + k - 1]
4. else
5.         spawn
6.                 divide A[q : r] into blocks $B_i$'s each containing 5 consecutive elements
                   (last block may contain fewer than 5 elements)
7.         sync
8.         parallel-for i <- 1 to $\lceil n/5 \rceil$ do
9.                 M[i] <- median of $B_i$ using sorting
10.        x <- Par-Select(M[1 : $\lceil n/5 \rceil$ ], $\lfloor (\lceil n/5 \rceil + 1) / 2 \rfloor$ {median of medians}
11.        t <- Par-Partition (A[q : r], x) {partition around x which ends up at A[t]}
12.        if k = t - q + 1 then
13.                return A[t]
14.        else if k < t - q + 1 then
15.                return Par-Select( A[q : t - 1], k)
16.        else
17.                return Par-Select(A[t + 1 : r], k - t + q - 1)
```

Here, number of items smaller than x is

$>= 3(\lfloor \lceil n/5 \rceil 1/2 \rfloor - 1)$

$>= 3n/10 - 6$

Number of items larger than x is
>= $3(\lfloor \lceil n/5 \rceil 1/2 \rfloor - 1)$
>= 3n/10 - 6

Number of items in any recursive calls (lines 15 and 17) is
<= n - (3n/10 - 6)
= 7n/10 + 6

The recurrence relation of this deterministic select algorithm is as follows:

$$T(n) = \begin{cases} \theta(1), & if\ n < 140, \\ T\lceil n/5 \rceil + T(7n/10 + 6) + \theta(n), & if\ n >= 140. \end{cases}$$

But, 7n/10 + 6 <= 7.5n/10 for n >= 140
Here, if we run the code on a single processor, we'll get the equation of work as follows:

$$T(n) = \begin{cases} \theta(1), & if\ n < 140, \\ T(n/5) + T(3n/4) + \theta(n), & if\ n >= 140. \end{cases}$$

As, we had proved in the class that the work of the above deterministic select algorithm is $\theta(n)$ using Akra-Bazzi Theorem:

From $(1/5)^p + (3/4)^p = 1$ we get p < 1.

Hence, T(n) = $\theta(n^p(1 + \int_1^n \frac{u}{u\hat{}(p+1)}\, du))$
$= \theta(n^p(1 + \int_1^n \frac{1}{u\hat{}(p)}\, du))$
$= \theta((\frac{1}{1-p})n - (\frac{p}{1-p})n^p)$
$= \theta(n)$

Similarly, if we run this algorithm on many processor, we will get the same equation:

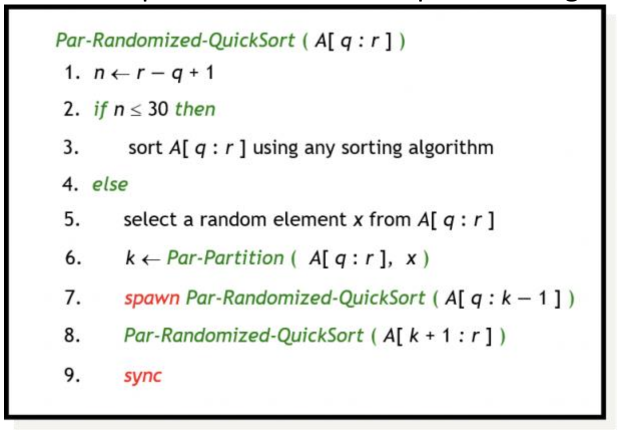
$$T(n) = \begin{cases} \theta(1), & if\ n < 140, \\ T(n/5) + T(3n/4) + \theta(log\hat{}2\ n), & if\ n >= 140. \end{cases}$$

Note that the parallel partition algorithm in the line 11 takes $O(\log^2 n)$ time to run in parallel.
Also, the depth of this recursion tree will be O(log n).
So, the total time complexity will be $O(\log^3 n)$.
The span we found out was $O(\log^3 n)$.

Therefore, parallelism will be given by P = $\frac{T_1}{T\infty}$
$= \frac{\theta(n)}{O(\log 3\ n)}$
$= \Omega(\frac{(n)}{(\log 3\ n)})$

Which suggests that the parallelism that we get will be at-least the quantity mentioned above.

1 (b)
This is the parallel randomized quick sort algorithm:



```
Par-Randomized-QuickSort ( A[ q : r ] )
1. n ← r − q + 1
2. if n ≤ 30 then
3.      sort A[ q : r ] using any sorting algorithm
4. else
5.      select a random element x from A[ q : r ]
6.      k ← Par-Partition (  A[ q : r ],  x )
7.      spawn Par-Randomized-QuickSort ( A[ q : k − 1 ] )
8.      Par-Randomized-QuickSort ( A[ k + 1 : r ] )
9.      sync
```

It has been modified slightly to not sort the complete array but to only get the kth smallest number in the array as shown below:

```
1. Par-Randomized-QuickSort-Findk (A[j + 1 : r], k)
2. n <- r – q + 1
3. if (n <= 30) then
4.        sort A[q : r] using any sorting algorithm
5.        return A[q + k - 1]
6. else
7.        select a random element x from A[q : r]
8.        j <- Par-Partition(A[q : r], x)
9.        if(k - 1 == j)
10.               return A[j]
11.       else if(k - 1 < j)
12.               spawn
13.                       Par-Randomized-QuickSort-Findk (A[q : j – 1, k])
14.               Sync
15.       Else
16.               spawn
17.                       Par-Randomized-QuickSort-Findk (A[j + 1 : r, k – j – 1])
18.               Sync
```

We have already proved in class that the run time of quick sort algorithm is $\theta(n\ log\ n)$
Although this algorithm does not completely sort the array, it compares the selected element with all other elements and in the worst case, it will have to run through the entire array to put the element in its proper position. And, in the average depth this recursion will reach is $\theta(n)$.
So, the worst case time complexity in this case would be $\theta(n\ log\ n)$.

So, work here is $\theta(n\ log\ n)$.

Note that the parallel partition algorithm in the line 8 takes $O(\log^2 n)$ time to run in parallel.
Also, the depth of this recursion tree w.h.p. will be $O(\log n)$ as proved in the class.
So, the total time complexity will be $O(\log^3 n)$.
The span we found out was $O(\log^3 n)$.

Therefore, parallelism will be given by P = $\frac{T1}{T\infty}$

$$= \frac{\theta(n \log n)}{O(\log 3\ n)}$$

$$= \Omega(\frac{(n \log n)}{(\log 3\ n)})$$

1 (c)

```
Par-Selection(A[q : r], k)
1. n <- r - q + 1
2. if n <= 30
3.      sort A[q : r]
4.      return A[q + k - 1]
5. Create temporary array flag[1 : n][1 : 2k + 1] and each element to zero.
6. Create array B of size 2k + 1
7. Randomly put 2k + 1 elements from A to B
8. parallel-for i <- 1 to n do
9.      parallel-for j <- 1 to 2k + 1 do
10.          if A[i] > B[j] then
11.               flag[i][j] <- true
12. Create an array B[1 : k] and count[1 : n]
13. parallel-for i <- 1 to n do
14.     count[i] <- Parallel-Sum(flag[i])
15.     if count[i] <= k then
16.          replace A[i] with max(B[1 : 2k + 1])
17. Sort(B[1 : 2k + 1]
18. return B[1 : k]
```

If we want to run this algorithm on a single processor, the recurrence relation will be:

$$T(n) = \begin{cases} \theta(1), & if\ n < 30, \\ \theta(nk) + \theta(nk) + \theta(k \log k), & if\ n >= 30. \end{cases}$$

So, work here is $\theta(nk)$.

Note that the parallel sum algorithm in the line 14 takes $O(\log^2 n)$ time to run in parallel.
So the recurrence relation for parallel will be:

$$T(n) = \begin{cases} \theta(1), & if\ n < 30, \\ \theta(log\ n) + \theta(log\ k) + \theta(log\ n) + \theta(log\verb|^|2\ n), & if\ n >= 30. \end{cases}$$

So, the span will be equal to $\theta(log\verb|^|2\ n)$

Therefore, parallelism will be given by $P = \frac{T_1}{T\infty}$

$= \frac{\theta(nk)}{\theta(\log 2\ n)}$

$= \theta(\frac{(nk)}{(\log 2\ n)})$

Task 2. [50 Points] Chocolates

Prove that the algorithm shown in Figure 3 is a k-approximation algorithm for selecting boxes of the minimum total cost that include at least one chocolate of each type. In other words, prove that I will not have to spend more than a factor of k more money than someone using an optimal algorithm for selecting the boxes.

Solution:

In order to solve this problem, I have used the Set Cover method.

What is the set cover problem?
Principle: "You must select a minimum number [of any size set] of these sets so that the sets you have picked contain all the elements that are contained in any of the sets in the input (wikipedia)." Additionally, you want to minimize the cost of the sets.

Initially, define the parameters to be used:
Valid instances: Universe U, $|U| = n$
Family of sets $F = \{ S_1, S_2, \ldots, S_m\}$, S_i is subset of U for all i.
Costs $p_1, p_2, \ldots, p_k$

Feasible Solutions: A set I is subset of |m| such that $U_{i \in I}\, S_i = U$
Objective function: Minimizing |I|
Greedy algorithm: In each iteration, pick a set which covers most uncovered elements, until all the elements are covered.

Goal:
Find a set $I \subseteq \{ 1, 2,.. m\}$ that minimizes $\sum p_i\, (_{i \in I})$, such that $U(_{i \in I})\, S_i = U$

Algorithm 1: Greedy algorithm for Min Set Cover
Let C represent the set of elements covered so far
Let cost effectiveness, or α , be the average cost per newly covered node

```
1: C ← ∅
2: while not all elements in the universe are covered do
3:      Let α = p(S) / |S-C|
4:      S set that covers the most elements which is not covered from the U and whose cost
        effectiveness is smallest
5.      For each e∈S-C, set price(e) = α
6:      C ← C ∪ {S}
7: end while
8: return picked sets
```

Note: According to the problem, there will be a state wherein the person can pick all the boxes containing his/her favorite chocolate. This will happen when, after subtracting the values of the individual boxes prices which contain the favorite chocolate it reduces to zero.

Algorithm 2: Greedy algorithm for the k-approximation:
This is a kind of weight approximation. As in the weights are no more in the same cost.
Let, OPT= Optimal cost
C = Derived cost

(i) We know $\sum_{e \in U} price(e)$ = cost o f the greedy algorithm = $c(S_1) + c(S_2) + \ldots + c(S_m)$
Because of the nature in which we distribute costs of elements.

(ii) We will show price(e_k) less than equal to OPT/ (n-k+1), where e_k is the kth element covered.
So, OPT = $c(O_1) + c(O_2) + \ldots + c(O_p)$
Now, assume the greedy algorithm has covered the elements in C so far. Then we know the uncovered elements, or |U-C|, are at most the intersection of all the optimal sets intersected with the uncovered elements:
$|U-C| \leq |O_1 \cap (U-C)| + |O \cap (U-C)| + \ldots + |O_p \cap (U-C)|$

In the greedy algorithm, we select a set with cost effectiveness α, where

$\alpha \leq c(Oi) / |O_i \cap (U-C)|$, i= 1…p .
We know this because the greedy algorithm will always choose the set with the smallest cost effectiveness, which will wither be smaller than or equal to a set the optimal algorithm chooses.

which will either be smaller than or equal to a set that the optimal algorithm chooses.

Algebra:
$c(O_i) \geq \alpha\, |O_i \cap (U-C)|$

OPT = $\sum$ $O_i >= \alpha \sum |Oi \cap (U - C)| >= \alpha\, |(U-C)|$

A <= OPT / |U-C|
Therefore, the price of the k-th element is:
α <= OPT / (n- (k-1))
= OPT / (n-k+1)

Putting (i) and (ii) together, we get the total cost of the set cover:

$\sum_{k=1}^{n}$ price(e_k) <= $\sum_{k=1}^{n} OPT * \frac{1}{n-k+1}$ = OPT(1+ ½ + … 1/n)
= OPT * H_n = OPT * log(n)

According to the problem, there will be a state wherein the person can pick all the boxes containing his/her favorite chocolate. This will happen when, after subtracting the values of the individual box prices which contain the favorite chocolate it reduces to zero. Hence, at this moment the person can pick all the boxes at the same time.
That means if we consider there are k boxes, then the person can at max chose k boxes together.

Hence, when we are selecting k boxes at a time, there will be some other elements along with the favorite chocolate which will get added as well.

Therefore, there exists a set in OPT which covers at least $n_t−1$ / k uncovered elements. Because the greedy algorithm always chooses the set which covers most uncovered elements, the greedy algorithm covers at least ($n_t−1$ / k) uncovered elements at iteration t.

Therefore, $n_k <= n_{t-1} – n_{t-1}/k = (1-1/k)\ n_{t-1}$
Now, by induction, $n_t <= (1-1/k)^t\ n$

Consider t = k * ln(n/k)

$n_t <= (1-1/k)^{k*\ln(n/k)} * n <= e^{-\ln(n/k)} * n <= (k/n) *n = k$

The greedy algorithm covers the remaining k elements using at most k sets, so the greedy algorithm uses at most (k + k*ln(n/k)) = k (1+ln(n/k)) sets overall.

Finally, we can say that the expressed equation is in upper bounded by k; and which can be return as OPT * k >= C

Hence, proved that I will not have to spend more than a factor of k more money than someone using an optimal algorithm for selecting the boxes.

Solution3.a

Given: $P = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{m_i m_j}{\sqrt{(c_1 d^2_{ij} + c_2 r_i r_j)}}$

Also m_i = m and c$_1$= 0 so,

$$P = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{m^2}{\sqrt{(c_2 r_i r_j)}}$$

$$P = \frac{m^2}{\sqrt{c_2}} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{\sqrt{(r_i r_j)}}$$

Here we can see that the only part that we are concerned with during computation is $\frac{1}{\sqrt{(r_i r_j)}}$.

Lets divide the masses into $log_{1+\varepsilon}\ n$ groups. We will classify the masses based on their radii r.

PSEUDOCODE:

FUNCTION:

```
1. createGroups(array : m):
2.      Num = sizeof(m)
ε3.     Groups : Array of size $log_{1+\varepsilon}\ n$ all initialized to 0
4.      Max = maximum radius in m
5.      Min = minimum raidus in m
6.      Range = max - min
7.      d = range / $log_{1+\varepsilon}\ n$
9.      for m' in m:
10.         groups[(m' - min)/s]++
11. Return groups
```

The above function runs in O(n) time. Lines 4 and 5 also run in O(n) time.

FUNCTION:

```
CalculateP(array: m):
1.      Gr = createGroups(m)
2.      temp = 0
3.      For i in range 0 to $log_{1+\varepsilon}\ n$:
4.          for j in range i to $log_{1+\varepsilon}\ n$:
5.              temp = temp + (gr[i] * gr[j] / sqrt(gr[i].radius * gr[j].radius)
6.      P = temp
```

We can see that he above runs in O(($log_{1+\varepsilon}\ n$)^2)

As masses are same the only thing that varies is the radii. We have distributed the radii based on their deviation from the mean radius. Hence they will vary atmost by a factor of (1+ε). Thus putting this in the above equation we can see that $\sqrt{r1\ *\ r2}$ = $\sqrt{(1+\varepsilon)^2(r'1\ *\ r'2\ )}$ = (1+ε) $\sqrt{r'1\ *\ r'2}$

Hence we can see that the final solution will be of (1+ε) factor of original answer P.

Solution 3b.

Given: $P\ = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{m_i m_j}{\sqrt{(c_1 d_{ij}^2 + c_2 r_i r_j)}}$

As c1 = 0, we get :

$$P\ =\ \frac{1}{\sqrt{c_2}} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{m_i m_j}{\sqrt{(r_i r_j)}}$$

We can now divide the space into $log_{1+\varepsilon}\ n$ sections and each section will have the value of $\frac{m_i m_j}{\sqrt{(r_i r_j)}}$ where 0 <= i,j < $log_{1+\varepsilon}\ n$

Like in above problem we can consider the division $\frac{m_i m_j}{\sqrt{(r_i r_j)}}$ analogous to the radius.

FUNCTION:

```
1. createGroups(array : m):
2.      Num = sizeof(m)
ε3.     Groups : Array of size $log_{1+\varepsilon}\ n$ all initialized to 0
4.      Max = maximum $\frac{m_i m_j}{\sqrt{(r_i r_j)}}$ in m
5.      Min = minimum $\frac{m_i m_j}{\sqrt{(r_i r_j)}}$ in m
6.      Range = max - min
7.      d = range / $log_{1+\varepsilon}\ n$
9.      for m' in m:
10.             groups[(m' - min)/s]++
11. Return groups
```

The above function runs in O(n) time. Lines 4 and 5 also run in O(n) time.
We can calculate p using below pseudocode:
FUNCTION:

```
CalculateP(array: m):
1.      Gr = createGroups(m)
```

```
2.  temp = 0
3.  For i in range 0 to $log_{1+\varepsilon}\ n$:
4.      for j in range i to $log_{1+\varepsilon}\ n$:
5.          temp = temp + (gr[i].r * gr[j].r)
6.  P = temp
```

Where gr[i].r = $\frac{m_i m_j}{\sqrt{(r_i r_j)}}$ for that group.

We can see that the above runs in O(($log_{1+\varepsilon}\ n$)^2) time.
We can see that the ratio $\frac{m_i m_j}{\sqrt{(r_i r_j)}}$ varies from the mean by a factor of (1+ ε) . Thus the solution obtained will vary from optimal solution by a factor of (1+ ε) .

Solution 3c.

Given: $P = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{m_i m_j}{\sqrt{(c_1 d^2_{ij} + c_2 r_i r_j)}}$

For this problem, lets divide the entire space into $log_{1+\varepsilon}\ n$ sections divided by huge distances. We can use the following approximation:
As these sections are separated by huge distances, d >> r let the mean radius of each masses in a section be of form εd. As d >> r then $c_1 d^2_{ij} >> c_2 r_i r_j$.

$$P = \sum_{i=1}^{logn-1} \sum_{j=i+1}^{logn} \frac{m_i m_j}{\sqrt{(c_1 d^2_{ij})}}$$

Here dij will be d + εd = d(1+ε)
Hence :

$$P = \sum_{i=1}^{logn-1} \sum_{j=i+1}^{logn} \frac{m_i m_j}{(1+\varepsilon)\sqrt{(c_1 d^2_{ij})}}$$

Thus we can see that the optimal solution is of the factor (1+ ε) of the solution obtained from the approximate solution.