CSE548, AMS542: Analysis of Algorithms, Fall 2017 Date: Dec 14

Homework 4

Group Number: 52

Name	SBU ID	% Contribution
Rahul Bhansali	111401451	33.33%
Kiranmayi Kasarapu	111447596	33.33%
Neha Indraniya	111499447	33.33%

Collaborating Groups

Group Number	Specifics Number (e.g., specific group member? specific task/subtask?)
18, 21	Question 1
26	Question 2
61	Question 2 & 3
42	Question 2

External Resources Used

	Specifics (e.g., cite papers, webpages, etc. and mention why each was used)
1.	https://www.cs.cmu.edu/~avrim/451f11/lectures/lect0908.pdf (Running time of deterministic Quick Select)
2.	https://www.cs.jhu.edu/~mdinitz/classes/ApproxAlgorithms/Spring2015/notes/lec4.pdf (Vertex cover as set cover problem) for Question 2

Task 1. [80 Points] Selection in Parallel Given an array of n distinct numbers and an integer k ∈ [1, n], this task asks you to select and return the k-th smallest number in the array efficiently in parallel.

(a) [20 Points] Parallelize the deterministic selection algorithm shown in Figure 1 which is taken from lecture 6 (slide 2). Write down the pseudocode of the parallel version. Analyze its work, span and parallelism.

Here is the parallelized implementation of the deterministic selection algorithm:

Par-Select(A[q:r],k):

1. $n = r - q + 1$
2. *if* $n <= 140$ *then*
3. $A[q : r]$ *and return* $A[q + k + 1]$
4. *else*
5. *in parallel divide* $A[q : r]$ *into blocks* B_i*'s each containing* 5 *consecutive elements* (*last block may contain fewer than* 5 *elements*)
6. *parallel for* $i <- 1$ *to* $ceil(n/5)$ *do*
7. $M[i] <- Median\ of\ B_i$ *using sorting*
8. $x <- Par - Select(M[1 : ceil(n/5)], floor((ceil(n/5) + 1)/2)$
9. $t <- Par - Partition(A[q : r], x)$
10. *if* $k == t - q + 1$ *then return* $A[t]$
11. *else if* $k < t - q + 1$ *then return* $Par - Select(A[q : t - 1], k)$
12. *else return* $Par - Select(A[t + 1 : r], k - t + q - 1)$

Work $T_1(n)$**:**

The runtime of the serialized algorithm of Deterministic select using Median of Medians was covered in class and its equation is as follows:

$$T(n) = T(n/5) + T(3n/4) + \theta(n) \quad \text{if } n >= 140$$
$$= \theta(1) \quad \text{if } n < 140$$

The runtime obtained after solving using Akra-Bazzi recurrence is $\theta(n)$ which is the work.

Span $T_\infty(n)$**:**

The Span can be given by the following recurrence:

$T(n) = T(n/5) + T(3n/4) + \theta((log\ n)^2)$ *where* $T(n/5)$ *is from Line* 8 *and* $T(3n/4)$ *is from Line* 11 *or* 12.

Now, as both $T(n/5) + T(3n/4)$ are positive, we can write:

$T(n) > T(3n/4) + \theta((log\ n)^2)$

Using Master's Theorem,

$T(n) = \Omega((log\ n)^3)$

Therefore, the span is $\Omega((log\ n)^3)$

Parallelism:

$$\frac{T_1(n)}{T_\infty(n)} = \frac{\theta(n)}{\Omega((log\ n)^3)} = O(\frac{n}{(log\ n)^3})$$

(b) [[20 Points] Consider the parallel randomized quicksort algorithm shown in Figure 2 which is taken from lecture 12 (slide 86). It sorts an array of n distinct numbers in increasing order of value. How will you modify it so that it returns the k-th smallest number in the array instead of sorting them, where k ∈ [1, n]. Write down the pseudocode of the parallel selection algorithm you obtain. Give high-probability bounds on its work, span and parallelism.

We'll modify the algorithm as follows:

Par-PRQ(A[q:r], k):

1. $n = r - q + 1$
2. *if* $n <= 30$ *then*
3. *sort* $A[q : r]$ *using any sorting algorithm then return* $A[q + k + 1]$
4. *else*
5. *select a random element x from* $A[q : r]$
6. $t <- par - Partition(A[q : r], x)$
7. *if* $k == t - q + 1$ *then return* $A[t]$
8. *else if* $k < t - q + 1$ *then return* $Par - PRQ(A[q : t - 1], k)$
9. *else return* $Par - PRQ(A[t + 1 : r], k - t + q - 1)$

Let's have a look at the:

Work $T_1(n)$:

We are using the reference (External Resource 1).

This is a deterministic quickSelect algorithm and its expected running time is $\theta(n)$.

Proof:

Let $T(n,k)$ denote the expected time to find the kth smallest in an array of size n, and let $T(n) = max_k\, T(n,k)$. We will show that $T(n) < 4n.$
First of all, it takes $n-1$ comparisons to split the array into two pieces in Line 6.
These pieces are equally likely to have size 0 and $n-1$, or 1 and $n-2$, or 2 and $n-3$, and so on up to $n-1$ and 0.
The piece we recurse on will depend on k, but since we are only giving an upper bound, we can imagine that we always recurse on the larger piece. Therefore we have:

$$\begin{aligned} T(n) &\leq (n-1) + \tfrac{2}{n}\sum_{i=n/2}^{n-1} T(i) \\ &= (n-1) + avg[T(n/2), ..., T(n-1)]. \end{aligned}$$

Assume inductively that $T(i) \leq 4i \quad for\, i < n$. Then,

$$\begin{aligned} T(n) &\leq (n-1) + avg[\,4(n/2),\ 4(n/2+1), ..., 4(n-1)] \\ &\leq (n-1) + 4(3n/4) \\ &\leq 4n \end{aligned}$$

Therefore, the complexity of the serial algorithm is $O(n)$.

Span $T_\infty(n)$:

The analysis is similar to the method used above. The difference is that Line 6 was taking $\theta(n)$ time to execute in serially, but its parallel version $par-Partition()$ executes in $\theta((log\ n)^2)$ as taught in class.
So, to compute the span, we'll replace the serial time of Lines 1-6 by the span of Lines 1 to 6. which gives us the following equation:

$$T(n) \leq (log\ n)^2 + \tfrac{2}{n}\sum_{i=n/2}^{n-1} T(i)$$

$$= (log\ n)^2 \ + \ avg[T(n/2), ..., T(n-1)].$$

$$T(n) \ \leq (log\ n)^2 \ + \ c\ (log\ (n))^2 \quad , where\ c\ is\ a\ constant$$

$$T(n) \ = \ O\ (log\ n)^2$$

Therefore, $T_\infty(n) \ = \ O\ (log\ n)^2$

Comment on above Work and Span times:
As taught in class, the expected runtime of RandomizedQuickSort turns out to be the observed runtime w.h.p in n which is $\Theta(n\ log\ n)$.
As above algorithm has been derived by modifiying the RandomizedQuickSort algorithm, so we can say that its observed runtime will also be the expected runtime wh.h.p. in n i.e.:

$T_1(n) \ = \ O(n)\ w.h.p.\ in\ n$

$T_\infty(n) \ = O\ (log\ n)^2 \ \ w.h.p\ in\ n$

Parallelism:

$$\frac{T_1(n)}{T_\infty(n)} = \ O\left(\frac{n}{(log\ n)^2}\right)$$

(c) [40 Points] Design a parallel selection algorithm that can return the smallest k numbers in an array of n distinct numbers in sorted order using Θ (nk) extra space, Θ (nk) work and Θ log2 n span. Write down the pseudocode and show details of your analyses of work, span and space usage

Approach:

1. We'll create k copies of the array A of size n and call this array A_{new}. This can be done in parallel.
2. We'll run a parallel for loop and in it call **Par-PRQ(A[q:r],index)** k times with indices for **index** varying from 1 to k and store the result in an array $answer$.
3. So, finally we'll have the smallest k numbers in the $answer$ array.

The Pseudo Code is as follows:

1. $par-SmallestKnumbers(A[1:n],\ k)$:

#Initialize in parallel an array A_{new} of size kXn which contains k copies of the array A

2. *parallel for* $i = 1$ *to k do* :

3. *parallel for* $j = 1$ *to n do* :

4. $A_{new}[i][j] = A[j]$ *#create a copy of array A k times*

5. *parallel for* $i = 1$ *to k do* :

6. $answer[i] = Par-PRQ(A_{new}[i],\ i)$ *#par* $-$ *PRQ() is the same algo used in* $1(b)$

7. *return answer*

Let's analyze the:

Space Complexity:

Here we are using:

1. A_{new} of size kXn = $\theta(nk)$
2. *answer* of size k = $\theta(k)$

Therefore, overall the space complexity is $\theta(nk) + \theta(k) = \theta(nk)$

Work $T_1(n)$**:**

Lines 2 to 4	$\theta(nk)$
Lines 5 to 6	$\theta(kn)$
Line 7	$\theta(1)$

Comments:
Line 5: is called k times
Line 6 calls $Par-PRQ(A_{new}[i],\ i)$ and as seen in $1(a)$ the runtime of the serialized algorithm of deterministic select is $\theta(n)$
Therefore runtime of Line 5-6 is $\theta(kn)$

Overall,

$$T_1(n) = \theta(nk) + \theta(kn) + \theta(1) = \theta(nk)$$

Span $T_\infty(n)$**:**

Lines 2 to 4	$\theta(log\ k\ +\ log\ n)\ =\ \theta(log\ n)$
Lines 5 to 6	$\theta(log\ k\ +\ (log\ n)^2)\ =\ \theta((log\ n)^2)$
Line 7	$\theta(1)$

Comments:
Lines 6 calls $Par-PRQ(A_{new}[i],\ i)$ and as seen in $1(b)$ runs in $\theta((log\ n)^2)$ span time.

Overall:

$$T_\infty(n)\ =\theta(log\ n)\ +\ \theta((log\ n)^2)\ =\ \theta((log\ n)^2)$$

Parallelism:

$$\frac{T_1(n)}{T_\infty(n)}\ =\ \theta\left(\frac{nk}{(log\ n)^2}\right)$$

Task 2
(a) [50 Points] 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.

In the given algorithm, we know that each type of chocolate can occur in at most k boxes. So lets for analysis purpose assume that each chocolate can be present in exactly k number of boxes.

Lets look at an example.
Lets say we have 6 different type of chocolates a, b, c, d, e, f and we want to choose in this order. There are 7 boxes as shown below and each box has price pi as listed below them.

1	2	3	4	5	6	7
ac	bde	acb	be	def	acf	ef
50	20	30	25	35	40	20

For choosing chocolate a, we pick up box 3 b3, as this has smallest price among all the boxes which chocolate a.
After first iteration, the price array looks something like this.

1	2	3	4	5	6	7
ac	bde	acb	be	def	acf	ef
20 (30 already paid for B3)	20	0 (30 paid for itself)	25	35	10 (30 already paid for B3)	20

Next we want to choose chocolate b, it comes for free as the minimum prices of all the boxes that contain chocolate b is 0, which is nothing but B3, and we already paid for it.
Similarly for chocolate c, the minimum price is), which is nothing but box 3, B3.

Now when we choose chocolate d, we pick up box 2, B2, which has minimum price as 20.
Now the array looks as shown below.

1	2	3	4	5	6	7
ac	bde	acb	be	def	acf	ef
20 (30 already paid for B3)	0 (20 paid for itself)	0 (30 paid for itself)	25	15 (20 paid for B2)	10 (30 already paid for B3)	20

Picking chocolate e comes for free as B2, has smallest price 0.

Now when picking chocolate f, we'll choose box 6, B6 as this has the smallest price 10, among all boxes which have chocolate f.
So the price array looks like the following

1	2	3	4	5	6	7
ac	bde	acb	be	def	acf	ef
20 (30	0 (20 paid	0 (30 paid	25	5 (20 paid	0 (30	10 (10 paid

already paid for B3)	for itself)	for itself)		for B2, 10 paid for B6)	already paid for B3, 10 paid for itself)	for B6)

We will choose boxes, which have price 0 at the end, so in this example, we will choose B2, B3 and B6. The total price is 20 + 30 + 40 = 90
But the optimal price is when we choose boxes B2 and B6, for which optimal cost is 20 + 40 = 60.

We'll have to prove that the price we pay, is at most k times the optimal price.
For analysis purpose, let assign weights for each chocolate in the following way:

Whenever we pick a new Box, say Bi, we assign each chocolate in that box, that we have not seen so far with a cost equal to price of the box Pi divided by number of chocolates in the box, not picked so far.
Then, we go ahead and subtract this cost from at most k boxes in which chocolate we are choosing is present. We will just maintain that we have paid Pi for the remaining k-1 boxes. We will add this cost to Bi's chocolates, when any of the K-1 boxes cost become 0 later.

In the above example. The cost assigned to each chocolate will look as follows after each iteration.

	a	b	c	d	e	f
1st iter	30/3 = 10	30/3 = 10	30/3 = 10			
4th iter				20/2 = 10	20/2 = 10	
6th iter	30/3 = 10	30/3 = 10	30/3 = 10			10/1 = 10
Final costs	20	20	20	10	10	10

In the first iteration when we pick up Box3, for choosing chocolate a, its price is 30. But it has chocolate b and c, which are not seen before, so we divide this 30 among all 3. So each gets 30/3 = 10 cost. We go and subtract 30 from boxes B1 and B6(subtracting from k boxes), which has chocolate "a", we will just maintain a record that we subtracted from B1 and B6 for box B3. We will add this cost for chocolates in B3 if the price of B1 or B6 ever goes to 0. We can do this, because we maintained a record for what box, we subtracted what amount.
In the 2nd & 3rd iteration, we pick the same box B3, for choosing chocolates "b" & "c". But we do not pay any cost as it is already paid for when choosing "a", so their costs remain the same.

When choosing chocolate d, we have chosen B2, which has “b”, “d” and “e”. But we have already paid for “b” when choosing B3. So we divide B2’s cost among d and e only, since both of them are being picked for the first time. So each gets a cost of 20/2 = 10.
For picking chocolate e next, we pick up B2, and its cost is already minimum, so no extra cost added.
Now when we pick chocolate f, we pick box 6 B6, but we already paid 30 when we are picking B3. The number of chocolates not seen so far in B6 is only one. So chocolate “f” gets a cost of 10/1. The 30 we paid earlier for B3, will get distributed to the a, b and c again. So the final cost of each of a, b, c, d, e, f are 20, 20, 20, 10, 10, 10 respectively.

We can clearly induce that, for a chocolate that is being pickup for the 1st time, we give at most k times price of the box we picked divided by the number of chocolates in that box Bi, we have not picked so far.
So cost of each chocolate can be written as

$$c_x \leq \frac{k * P_i}{\left| Bi - (B1 \cup B2 \cup B3 ... B_{i-1}) \right|}$$

If we see the above example, we can see the cost we paid for the boxes picked, is summation of all costs of each chocolate we assigned the way above.
Therefore

$$C = \sum_{x=1}^{m} c_x$$

In the optimal solution, lets say we have chosen a set of boxes S. All the boxes together cover all chocolates atleast once, they may be covered more than once also.
Therefore

$$C^* = \sum_{B_i \in S} p_i = \sum_{B_i \in S} \sum_{x \in B_i} c_x^* \geq \sum_{x=1}^{m} c_x^*$$

We define c_x^* as cost of each chocolate in optimal solution. This cost will be cost of box Bi divided by chocolates not seen so far.

$$c_x^* = \frac{P_i}{\left| Bi - (B1 \cup B2 \cup B3 ... B_{i-1}) \right|}$$

In the optimal solution, suppose for chocolate c , we have chosen Bj box, if we have chosen the same box in our algorithm, then it must have been the minimum priced box present at that iteration, and we are paying k*Pj’s price in our solution. In the optimal solution, we’ll be paying only Pj price for it.
Therefore

$$c_x = c_x^* * k$$

This will hold for all chocolate types that are present in Bj. In the optimal solution, we do not pay any extra cost for it as we already paid for this box for chocolate c, In our solution also, we will not pay any extra cost as k*pj is already paid when choosing chocolate m.

Suppose we have chosen a different box other than Bj in the optimal solution lets say Bj', its cost will be $P_{j'}$ in which chocolate c is present, we are still paying k*Pj amount for it in our solution. It means in optimal solution, we have chosen something larger than Pj as Pj is the minimum priced box for chocolate m.
Therefore,

$$c_x \leq c_x^* * k \qquad (\text{because pj < pj'})$$

So combining the equations, we can write as follows

$$C = \sum_{x=1}^{m} c_x$$

$$C \leq \sum_{x=1}^{m} c_x^* * k = k \sum_{x=1}^{m} c_x^*$$

$$\leq k * C^*$$

$$\Rightarrow \frac{C}{C^*} \leq k$$

Therefore, this algorithm is a k-approximation algorithm.

Alternate Approach:

We can clearly this problem as a generalized set cover problem. We are saying that we have a universal set U of m objects(m chocolates here). We are given there are n boxes, these n boxes can be considered as n subsets of these m objects. Now each box is given as price Pi. We can project this as weight for each subset. In the subset problem, we studied in class, we considered the weight of each subset to be 1. Here we will consider it as some wi.
It is also given that, each chocolate will be present in atleast one box, and at most k boxes. So we can say that each object m, will be present in atleast one set and in atmost k sets. Now we have to find a solution which minimizes the cost while selecting a subset of sets which span all m objects.
We have studied the vertex-cover problem in the class, and we can transform vertex-cover problem as a special case of set-cover problem we studied in class, where weights of each subset is considered as 1.
The universe U is the edge set E, and the family of sets is F = {Su | u ∈ V } where Su = {{u, v} | {u, v} ∈ E}.
We know that vertex-cover is 2-approximation algorithm. We reason that, since one edge is incident upon 2 vertices, we proved in class that it is a 2-approx algorithm. In this case, it means that each element appears in exactly 2 sets.
Our original problem now reduces to frequency related algorithm, as each element can appear in at most k boxes. For analysis purpose we will assume that each element(or chocolate) will appear in exactly k sets/boxes. So we can construct a graph, with vertices as boxes and edges as chocolates. Here our edge is not just incident upon 2 vertices, but it will be incident on k vertices, we'll have to imagine it hypothetically. If a particular chocolate appears in only one box, then we will just add a self-loop(or self-edge) to that vertex.

In each iteration, we are choosing an uncovered chocolate by preference and select all the sets that contain this element and we repeat until all elements are covered. Basically we'll remove all incident chocolates/edges on the vertex which has the smallest price.
This is an k-approximation for the same reason that our algorithm for vertex cover was a 2-approximation. Informally, for every two chocolates ci, cj ∈ U considered by this algorithm, there are no sets which cover both ci and cj (or else whichever was covered first would have caused this set to be included, so the algorithm would not consider the second element). The Optimal has to be at least as large as the number of iterations of this algorithm. On the other hand, in each iteration this algorithm only picks k vertices/boxes/sets. Hence it includes at most k * Optimal solution sets.

Task-3

(a) [15 Points] Suppose mi = m for all i ∈ [1, n], and c1 = 0, but c2 > 0. Give an algorithm for approximating P within a factor 1+ ε of the exact value in O $log_{1+\varepsilon}{}^{2}(n)$ **time, where > 0. You are allowed to spend up to Θ (n) time for preprocessing the input before you compute the approximate value of P. Give pseudocode and show your analyses of approximation ratio and running time.**

Solution:

P can be calculated by:-

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

It is given that m_i =m and c1=0 so,

$$P = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{mm}{\sqrt{(0 + c_2 r_i r_j)}}$$

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

So, now we need to optimize on $\sqrt{(r_i r_j)}$. As this is in denominator, we know that only the value which will be lower will give significant values.
Hence, here we can divide the masses in $log_{1+\varepsilon}(n)$ groups based on the range of "r".
So, each group will have k_i radius in the range of $(r_{max} - r_{min}) / log_{1+\varepsilon}(n)$.

Pseudocode:-

```
1 getGroups(Array masses)
```

```
 2    n=size(masses)
 3    groups= new Array[log n]
 4      max_radius=max(masses.getRadius)
 5      minr_radius=min(masses.getRadius)
 6      range=max_radius-min_radius
 7      s=range/log n
 8      for mass in masses:
 9      groups[ (mass.getradius-minr_radius)/s].count ++  // adding this to the
appropriate group
10 Return groups

11 calculateP(Array masses)
12      groups=getGroups(masses)
13      P=0
14      for i=0 to log(n)-1
15              for j=i to log(n)
16                  temp =
                      (groups[i].count*groups[j].count)
                     /sqrt(groups.getradius[i]*groups.getradius[j])
17                   P=P+(temp)
19 return P
```

Complexity:

Here in preprocessing we are dividing these groups based on radius range, as we have one loop there hence the complexity will be O(n).

In calculateP function the we have two loops each of them runs for log(n) times hence complexity will be $log_{1+\varepsilon}{}^{2}(n)$

Approximation ratio:

The mass m with original radius r_i lies in one of the $log_{1+\varepsilon}(n)$ groups, so it it's new radius will be the mean of that group, this will be r+ ε so the term $\sqrt{}(r_i r_j)$ will at most have value $\sqrt{}(r_i + \varepsilon r_i)(r_j + \varepsilon r_j)$

= $\sqrt{}(r_i r_j)(1+\varepsilon)^2$

= $\sqrt{}(r_i r_j)(1+\varepsilon)^2$

= $(1+\varepsilon)\sqrt{}(r_i r_j)$

We know that without approximation it should have returned $\sqrt{(r_i r_j)}$, so from the above equation we will have $(1+\varepsilon)$ approximation ratio.

(b) [15 Points] Suppose c1 = 0 and c2 > 0 as in part 3(a), but not all objects have the same mass. Give an (1 +ε)-approximation algorithm for this case. Give pseudocode and show your analyses of approximation ratio and running time.
Solution:
For this we will have

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

So, now we need to optimize on $m_i m_j / \sqrt{(r_i r_j)}$. Now, the value in numerator and denominator both vary and we need to write our approximation algorithm accordingly. Hence, here we will divide our planets to $log_{1+\varepsilon}(n)$ groups based on the range of ratio of mass/ $\sqrt{}$ (radius). So, each group will have k_i ratios in the range of ($ratio_{max} - ratio_{min}$)/ $log_{1+\varepsilon}(n)$.

Pseudocode:-

```
1 getGroups(Array masses)
2        n=size(masses)
3       groups= new Array[log n]
4       max_radius=max(masses.mass/sqrt(masses.getRadius))
5       minr_radius=min(masses.mass/sqrt(masses.getRadius))
6       range=max_radius-min_radius
7       s=range/log n
8       for mass in masses:
9             groups[ (mass.mass/sqrt(mass.getRadius)-minr_radius)/s].count ++
               // adding this to the appropriate group
10 Return groups

11 calculateP(Array masses)
12          groups=getGroups(masses)
13          P=0
14          for i=0 to log(n)-1
15              for j=i to log(n)
16                   temp=(groups[i].count*groups[j].count)*
                            (groups.getmass[i]*groups.getmass[j])
```

/ sqrt (groups.getradius[i]*groups.getradius[j])

17 P=P+(temp)

28 return P

Complexity:

Here in preprocessing we are dividing these groups based on radius range, as we have one loop there hence the complexity will be O(n).

In calculateP function the we have two loops each of them runs for log(n) times hence complexity will be $log_{1+\varepsilon}{}^{2}(n)$

Approximation ratio:

We know that for any mass of mass value m_i and radius r_i lies in one of the $log_{1+\varepsilon}(n)$ groups, so it's ratio will be the mean of that group, this will be deviating from original by at most 1+ ε factor so the term ratio of the masses $g_i = \frac{(1+\varepsilon)m_i}{\sqrt{(r_i)}}$

So we now have $g_i * g_j = \frac{(1+\varepsilon)m_i}{\sqrt{(1+\varepsilon)}\,(r_i)} * \frac{(1+\varepsilon)m_j}{\sqrt{(1+\varepsilon)}\,(r_j)}$

$= \frac{(1+\varepsilon)^2 m_i m_j}{(1+\varepsilon)\sqrt{r_i r_j}} = \frac{(1+\varepsilon)\, m_i m_j}{\sqrt{r_i r_j}}$

We know that the actual solution should be $\frac{m_i m_j}{\sqrt{r_i r_j}}$, so this is in the ratio of $(1+\varepsilon)$.

Hence, approximation ratio= $(1+\varepsilon)$

(c) [20 Points] Repeat part 3(b) assuming c1 is also positive (i.e., c1 > 0).

Solution:

For this we will have

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

Here we have $c_1 d^2{}_{ij} + c_2 r_i r_j$ in denominator, based on this we can divide the masses in log(n) subgroups, such that the masses which are closer are in one subgroup. So for any two groups $g_i \; and \; g_j$ we will have the distance of d between them. This distance will be taken from the centres of groups. The radius of the group will be εd.

In each group for mass, we can take the sum of masses from each group.

Here, we know that, the distance d >> r, so we can easily say that:-

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

m=log (n)

$$P = \sum_{i=1}^{log(n)-1} \sum_{j=i+1}^{log(n)} \frac{g_i g_j}{\sqrt{(c_1 d^2_{ij})}}$$

Pseudocode:-

```
1 getGroups(Array masses)
2     n=size(masses)
3     groups= new Array[log n]

4      for mass in masses:
5           add to nearest Group

6 calculateP(Array masses)
7     groups=getGroups(masses)
8      P=0
9      for i=0 to log(n)-1
10       for j=i to log(n)
11          temp=(groups[i].mass*groups[j].mass)/sqrt(d[i,j]*d[i,j])
12          P=P+(temp)
13 return P
```

Complexity:

Here in preprocessing we are dividing these groups based on distance range, as we have one loop there hence the complexity will be O(n).

In calculateP function the we have two loops each of them runs for log(n) times hence complexity will be $log_{1+\varepsilon^2}(n)$

Approximation ratio:

Here we have approximated the distance in the denominator by the factor of ε, so for any real distance d(i,j), we will have approximated distance of (1+ ε)d(i,j)

$$P = \sum_{i=1}^{log(n)-1} \sum_{j=i+1}^{log(n)} \frac{g_i g_j}{\sqrt{(c_1 d^2_{ij})}}$$

As, we have taken the total mass in this did not approximate it, let us concentrate on denominator from above expression. As mentioned the distance is at most (1+ ε) times

the original distance i.e. $\sqrt{}(c_1 d^2_{ij}) = \sqrt{}(c_1 * ((1+\varepsilon) originald^2_{ij}) = (1+\varepsilon)$
$\sqrt{}(c_1 * (originald^2_{ij})$
Hence, it is in ratio of $(1+\varepsilon)$ times actual P, hence approximation ratio will be $(1+\varepsilon)$