4a),

```
import numpy as np

def min_u(A, A_row):
    A_col = sum(A)
    u = np.zeros((A_row + 1, A_col + 1))
    ans = 0

    for i in range(1, len(u)):
        for j in range(0, A_col + 1):
            if A[i - 1] == j:
                u[i, j] = A[i - 1]
            if u[i - 1, j] != 0:
                u[i, j] = u[i - 1, j]
                ans = u[i - 1, j] + A[i - 1]
                u[i, int(ans)] = ans

    A_col = sum(A)
    midLastRow = 0
    if A_col % 2  == 0:
        A_col += 2
        midLastRow = A_col /2
    else:
        midLastRow = (A_col+1) / 2

    lastRow = max(u[len(u) -1][:int(midLastRow)])
    print("residul is: ",sum(A) - 2 * lastRow )

for i in range(3,13):
    p10  =pow(10,i)
    arr = np.random.randint(p10, size=100)
    min_u(arr, len(arr))
    print(arr," &&&")


min_u(arr, len(arr))
```

```
package HW5;

import java.util.ArrayList;
import java.util.NoSuchElementException;
import java.util.Random;
import java.util.Scanner;

public class Heap<T extends Comparable<T> > {


	private ArrayList<T> items;
	public Heap()
	{
		items = new ArrayList<T>();
	}

	private void switchUp ()
	{
		int k = items.size() -1;
		while( k > 0)
		{
			int p = (k-1)/2;
			T item = items.get(k);
			T parent = items.get(p);

			if (item.compareTo(parent) > 0)
			{
				// swap two elements;
				items.set(k,  parent);
				items.set(p, item);

				k = p;
			}
			else
			{
				break;
			}
		}
	}

	public void switchDown()
	{
		int k = 0;
		int l = 2 * k + 1;
		while(l < items.size())
		{
```

```
                int max = l, r = l + 1;
                if ( r < items.size())
                {
                        if(items.get(r).compareTo(items.get(l)) > 0)
                        {
                                max ++;
                        }
                }
                if (items.get(k).compareTo(items.get(max)) < 0 )
                {
                        T temp = items.get(k);
                        items.set(k, items.get(max));
                        items.set(max, temp);
                        k = max;
                        l = 2 * k+1;
                } else {
                        break;
                }
        }
}

public void insertV( T item)
{
        items.add(item);
        switchUp();
}

public T delete()
            throws NoSuchElementException {
               if (items.size() == 0) {
                  throw new NoSuchElementException();
               }
               if (items.size() == 1) {
                  return items.remove(0);
               }
               T hold = items.get(0);
               items.set(0, items.remove(items.size()-1));
               switchDown();
               return hold;
            }

public boolean isEmpty() {
      return items.isEmpty();

   }
public String toString()
```

```
	{
		return items.toString();
	}

	public int size() {
	      return items.size();
	   }

	public static void main(String[] args) {

		// TODO Auto-generated method stub
		//Heap<Integer>  arr = new Heap<>();

//		Scanner input = new Scanner(System.in);
//		System.out.print("Enter the next it, 'quit' to stop: " );
//		String l = input.next();
//
//
//		while(!l .equals("quit"))
//		{
//			arr.insertV(Integer.parseInt(l));
//			System.out.println(arr);
//			System.out.print("Enter next integer, 'quit' to stop: " );
//			l = input.next();
//		}



		int  power = 3;
		while(power <= 12)
		{
			long exp = (long) Math.pow(10, power);

			Heap<Long> nums = new Heap<>();
			long[] array = new long[100];
			Random rand  = new Random();


			 for (int i = 0; i< array.length; i ++)
			 {
				double seed = rand.nextDouble();
				array[i] = (long)(seed* exp);
				nums.insertV(array[i]);

			 }
			 int n = array.length;
```

```
                    while( n!=1 )
                    {
                            long max1 = nums.delete();
                            long max2 = nums.delete();
            nums.insertV( max1 - max2 );

                            n -=1;
                    }
                    System.out.println("residual in power of " + power  +" is : "+  array[0]);
                    power +=1;
                }

        }



}
```

13. The problem of searching for cycles in graphs arises naturally in financial trading applications. Consider a firm that trades shares in n different companies. For each pair $i \neq j$, they maintain a trade ratio r_{ij}, meaning that one share of i trades for r_{ij} shares of j. Here we allow the rate r to be fractional; that is, $r_{ij} = \frac{2}{3}$ means that you can trade three shares of i to get two shares of j.

A *trading cycle* for a sequence of shares $i_1, i_2, \ldots, i_k$ consists of successively trading shares in company i_1 for shares in company i_2, then shares in company i_2 for shares i_3, and so on, finally trading shares in i_k back to shares in company i_1. After such a sequence of trades, one ends up with shares in the same company i_1 that one starts with. Trading around a cycle is usually a bad idea, as you tend to end up with fewer shares than you started with. But occasionally, for short periods of time, there are opportunities to increase shares. We will call such a cycle an *opportunity cycle*, if trading along the cycle increases the number of shares. This happens exactly if the product of the ratios along the cycle is above 1. In analyzing the state of the market, a firm engaged in trading would like to know if there are any opportunity cycles.

Give a polynomial-time algorithm that finds such an opportunity cycle, if one exists.

$n_1 \ldots n_n$ companies.

pair ($i \neq j$)

ratio ij.

one share i k j.

ration can be $r_{ij} = \frac{2}{3}$

3 share i,
2 show j

$r_{ij} = \frac{2}{3}$

Solution: By hint. use the key fact of reducing the negative cycle detection problem. that could relates products to sums; $\log(xy) = \log x + \log y$. for value. i, j.
For Grape G. with direct edge (i,j). The cost of the pair we say $\log r_{ij}$. for the edge (i,j). By given: the cycle(g) increases the number of shares when the ratio along the cycle is above 1, which is $\prod_{(i,j)} r_{ij} > 1$. In other words. in this cycle: $\sum_{(i,j)} \log r_{ij} > 0$, then in the cycle(g). $-\sum_{(i,j)\in g.} r_{ij} < 0$.

Thus. This opportunity exist if & only if by the proof of detecting negative cycles. which proofed by Bellman - Ford algorithm.

Algorithm: Bell-man algorithm on the book. p.288.

corness: on the book. by Bellmann-algorithm. plus $\log x + \log y > 0$ $\log xy$ $xy > 0$

Complexity: Base on Bell man algorithm, the complexity is $O(|V| \cdot |E|)$. n is the number of nodes. E is numbers of edges.

10. Suppose you are given a directed graph $G = (V, E)$, with a positive integer capacity c_e on each edge e, a source $s \in V$, and a sink $t \in V$. You are also given a maximum s-t flow in G, defined by a flow value f_e on each edge e. The flow f is *acyclic:* There is no cycle in G on which all edges carry positive flow. The flow f is also integer-valued.

Now suppose we pick a specific edge $e^* \in E$ and reduce its capacity by 1 unit. Show how to find a maximum flow in the resulting capacitated graph in time $O(m + n)$, where m is the number of edges in G and n is the number of nodes.

⇒ the maximum flow of the new graph either not change. or decreases by 1. if the flow value one the edge, which reduce one, was smaller than the capacity before change. the maximum flow will not change.

1. The f is a acyclic and given s-t flow of G_1, in the maximum flow of G_1. if one of the edge, e, reduce it capacity by 1 unit, G_1 becomes a residual graph with capacity next to each edge by the definition of the residual graph. that is $C_e - f(e)$ leftover. which is 1 unit of capacity on which we could try pushing flow forward. In a Augumenting paths in a residual graph (G_f). we push flow from s to t in G_f. let p be a simple path in G_f., that is p does not visit any node more than once.

2. Because all capacities in the flow network G_1 are integers. then the Ford-Fulkerson algorithm terminates in at most C iterations of the while loop. here. $C = 1$. By the running time of Ford-Fulkerson, algorithm in number of n nodes and m edges in G_1. we can use $O(m+n)$ to simplify the bounds

correctness: on the book. p 350.

16. Back in the euphoric early days of the Web, people liked to claim that much of the enormous potential in a company like Yahoo! was in the "eyeballs"—the simple fact that millions of people look at its pages every day. Further, by convincing people to register personal data with the site, a site like Yahoo! can show each user an extremely targeted advertisement whenever he or she visits the site, in a way that TV networks or magazines couldn't hope to match. So if a user has told Yahoo! that he or she is a 20-year-old computer science major from Cornell University, the site can present a banner ad for apartments in Ithaca, New York; on the other hand, if he or she is a 50-year-old investment banker from Greenwich, Connecticut, the site can display a banner ad pitching Lincoln Town Cars instead.

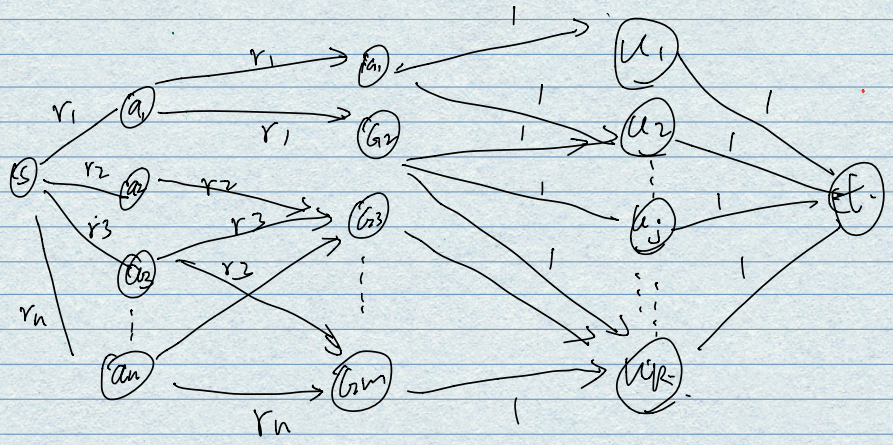
But deciding on which ads to show to which people involves some serious computation behind the scenes. Suppose that the managers of a popular Web site have identified k distinct *demographic groups*

$G_1, G_2, \ldots, G_k$. (These groups can overlap; for example, G_1 can be equal to all residents of New York State, and G_2 can be equal to all people with a degree in computer science.) The site has contracts with m different *advertisers*, to show a certain number of copies of their ads to users of the site. Here's what the contract with the i^{th} advertiser looks like.

- For a subset $X_i \subseteq \{G_1, \ldots, G_k\}$ of the demographic groups, advertiser i wants its ads shown only to users who belong to at least one of the demographic groups in the set X_i.
- For a number r_i, advertiser i wants its ads shown to at least r_i users each minute.

Now consider the problem of designing a good *advertising policy*—a way to show a single ad to each user of the site. Suppose at a given minute, there are n users visiting the site. Because we have registration information on each of these users, we know that user j (for $j = 1, 2, \ldots, n$) belongs to a subset $U_j \subseteq \{G_1, \ldots, G_k\}$ of the demographic groups. The problem is: Is there a way to show a single ad to each user so that the site's contracts with each of the m advertisers is satisfied for this minute? (That is, for each $i = 1, 2, \ldots, m$, can at least r_i of the n users, each belonging to at least one demographic group in X_i, be shown an ad provided by advertiser i?)

Give an efficient algorithm to decide if this is possible, and if so, to actually choose an ad to show each user.



we can convert it to max-flow problem, by ford-fulkerson's algorithm.
on the graph. per advertiser i → vertex a_i.
per user i, → a vertex a_i.
per group i → a vertex G_i
one source s. and sink t.

note: let f → flow

on the directed edges on the graph.
from $s \to a_i$ with weight r_i → no more than r_i ads to be shown for advertiser i. we say r_i is the upper bound can be served for the advertiser. Every advertiser i and group j. → edge from a_i to G_i. with weight r_i.

- Every user $-i$ and group j add edges from G_i to u_i, with weight 1. if $G_i \in u_i$ → flow of ads can move from group to user who are in this group. and every user

can be shown a single ad. on capacity 1.

- each user i, one edge from u_i to d, with weight 1.

algorithm: For the max-flow, if the max-flow is $\sum_i r_i$ say R. for every advertiser i, at least r_i users can be assigned to each advertiser-i.

Run ford-fulkerson's algorithm on the graph as the max-flow R.

For each advertiser i,

f on s and a_i will be r_i as max flow is R.

For each or group i and user j

for each advertiser-i:

for every and group j:

$f = \text{flow}(a_i, g_i)$ // f from a_i to g_i by fulkerson algorithm.

there will be at least f unassigned users such that $(G_i, u_k) =$ assign these f users to advertiser i and mark them assigned.

runtime of ford-fulkerson $= O(m,n)$ m is edges, n is notes.

where the $\sum_i r_i$ is the sum of all r_i's. the maximum outgoing flow at any edge is R.

E is the number of advertisers(a) + number of users(u) is a part of f sum of # of group that per ad target.

the complexity: $O((a + u + \sum_{i=1}^{a} |x_i| + \sum_{i=1}^{u} |u_i|) * \sum_i r_i)$.

Programming Assignment

For this problem you need to append your code to your submission pdf.

Question 4 (20 pts). *In this programming question, we will be considering the* NUMBER PARTITION *problem. As input, the number partition problem takes a sequence $A = (a_1, a_2, \ldots a_n)$ of non-negative integers, and outputs a sequence $S = (s_1, s_2, \ldots s_n)$ of signs $s_i \in \{-1, +1\}$ such that the* residue

$$u = \left| \sum_{i=1}^{n} s_i a_i \right|$$

complexity:

is minimized. Another way to view the problem is to split the set (or multi-set) of numbers given by A into two subsets A_1 and A_2 whose sums are as equal as possible. The absolute value of the difference of the sums is the residue.

(a.) NUMBER PARTITION *can be solved exactly in time polynomial in n and B. Find and implement a dynamic programming algorithm that has worst-case running time that is a polynomial function of n and B. As always, justify correctness, formulate a DP recurrence and analyze the running time.*

(b.) A deterministic heuristic for NUMBER PARTITION *is the Karmarkar-Karp algorithm. This KK algorithm uses* differencing*: repeatedly take the two largest elements from A, call them a_i and a_j, and replace the larger by $|a_i - a_j|$ while replacing the smaller by zero, until there is only one non-zero element left. This final non-zero element corresponds to an achievable residue, but it may not be the best possible residue, that is why we refer to this as a heuristic. (Each difference operation corresponds to putting a_i and a_j in different sets: if they were, weight 75 and 71, then it is as if we had just one element of weight 4 around.) For example, if A is initially $(10, 8, 7, 6, 5)$, then the KK algorithm proceeds as follows:*

better, but not max as 2 may not

$(10, 8, 7, 6, 5) \to (2, 0, 7, 6, 5) \to (2, 0, 1, 0, 5) \to (0, 0, 1, 0, 3) \to (0, 0, 0, 0, 2)$.

→ prove this algo?

Prove that there is a partition that achieves the residue computed by the KK algorithm. Then implement the KK algorithm (you need not compute the partition, just the residue), and justify your algorithm's running time. You should strive to obtain an $O(n \log n)$ implementation.

(c.) Run several experiments for each of your implementations head-to-head on sets of 100 integers chosen uniformly starting from the range $[1, 1000]$ and working up to the range $[1, 10^{12}]$ by powers of 10. Make sure to use 64-bit integers, and make sure your random number generator is working correctly on ranges this large. Point out where your pseudopolynomial-time algorithm from part (a) runs too slowly to be useful any more, and describe any differences you see in results obtained by the KK heuristic versus the exact algorithm.

4a). Algorithm:

```
A ← Array of input.
A_row ← the length of row A.
A_col ← the length of colms.
u ← a matrix by A_row. x A_col.
ans ← 0.
for i in 1 to len(u):
    for j in 0 to A_col+1:
        if A[i-1] == j:
            u[i,j] = A[i-j]
        if u[i-1, j] != 0
            u[i,j] = u[i-1,j]
            ans = u[i-1,j] + A[i-1]
            u[i, int(ans)] = ans
A_col = sum(A)
if A_col is even:
    midlast = Acol / 2
else: midlast = (A_col+1)/2
lastRow = max(u[len(u)-1][:int(midlast)])
return sum(A) - 2* lastRow.
```

complexity: general and track the matrix use $O(n^2)$. find residul. and calculate by constant time. thus. complexity is $O(n^2)$.

terminate: As the question is constructed by a matrix, AS we partition the law row the the matrix. and back track. there is a base case that reach to the first row of the matrix., which the row with only one and the first element of the. Array. that will be terminated.

4b).

```
l = leng of array.
Algorithm.      i = 1
while  i != l:
    m1 = heap.pop( )
    m2 heap.pop( )
    heap.insert.( |m1 - m2| ). //absolute value of |m1 - m2|
    i += 1
```

Termination: As the input of the array with length l: every loop. the Array length will be reduce one. when i equal. l, that is the array remine one elemat that is the residual by KK algorithm, it terminates.

correctness: By the definition of Heap and KK algorithm. The elements in Heap always sorted. for every loop. that is insert the difference of the first two maximum numbers recurssively. when the loop reach the base case. that is the residual of KK algorithm. It also very clear if we proof by induction.

4(c). For the pseudopolynomial-time algorithm from part (a). after the random number in 5,000,000 the speed goes very slow. However, use KK heuristic, the residual getting larger and larger with the random range getting bigger. and the running time is very short because of the complexity $O(n \log n)$.