Name: Tachyun Yang 2021-14284 Hw2

Problem 1 Suppose Vlam numbers are finite.

define $V_n = \{u_1, u_2, u_3, \ldots, u_{n-1}, u_n\}$

where V_n is the largest number and V_{n-1} is the second largest.

The sum of u_n and u_{n-1}

$\therefore u_n + u_{n-1} = u_{max}$ is an Vlam number because

① it is the sum of two Vlam number

② u_n, u_{n+1} is the largest number so there is no two other numbers in V_n that adds up to u_{max}.

So V_n isn't finite. V_n is infinite.

int start is the pointer for first element in arr[]
int end is the pointer for last element
int goal is the number we are looking for.
arr[] is the array list of the integers.

Problem 2

```
ternary search ( int start, int end, int goal, int arr[])
{
 int three  = (end-1)/3;
int three2  = end-(end-1)/3;

if (arr[three] == goal) return three;
if(arr[three2] == goal) return three2;
if (goal < arr[three]) return ternary search(start, three-1, goal, arr[]);
if( goal > arr[three2]) return ternary search( three2+1, end, goal, arr[]);
else return ternary search (three+1, three2-1, goal, arr[]);

}
```

Problem 3

Proof by counter example;

Assume that $f(x) = 3x$

$g(x) = x$

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{3x}{x} = 3$$

$$\left|\frac{f(x)}{g(x)}\right| \le 3$$

$|f(x)| \le 3|g(x)|$ for all x

$C = 3$, $k \le$ constant any x, eg $k = 1$

Therefore $f(x) = O(g(x))$.

However, is $2^{f(x)} = O(2^{g(x)})$?

$$\lim_{x\to\infty} \frac{2^{f(x)}}{2^{g(x)}} = \lim_{x\to\infty} \frac{2^{3x}}{2^{x}} = \lim_{x\to\infty} 2^{2x} = \infty$$ not a finite constant.

Hence, $2^{f(x)}$ is not $O(2^{g(x)})$.

b) $\log(n!) = \log(1 \cdot 2 \cdot \ldots \cdot n-1 \cdot n) = \log 1 + \log 2 \ldots \log n$

$n \log n = \log n + \log n \ldots + \log n$ for $n \geq 1$

$\therefore \log(n!) \leq n \log n \quad \therefore \log(n!) = O(n \log n)$

for $k \in \{0, 1, \ldots n-1\}$

$n-1 \geq k$

$-k + n - 1 \geq 0$

$-k^2 + kn - k \geq 0$

$n + kn - k^2 - k \geq n$

$n(1+k) - k(k+1) \geq n$

$(k+1)(n-k) \geq n$

for any k, $n \leq (n-k)(k+1)$

$\prod_{k=0}^{n-1} n \leq \prod_{k=0}^{n-1} (n-k)(k+1)$

$n^n \leq (1 \cdot n) \cdot (2 \cdot (n-1)) \cdot (3 \cdot (n-2)) \ldots (n \cdot 1)$

$n^n \leq (n!)^2$

$\log(n^n) \leq \log(n!)^2$

$n \log n \leq 2 \log(n!)$

Thus $\log(n!)$ is $\Omega(n \log n)$

Therefore

$\log(n!)$ is $\Theta(n \log n)$.

https://stackoverflow.com/questions/2095395/is-logn-%CE%98n-logn/2095472

problem 4

https://en.wikipedia.org/wiki/Stable_marriage_problem

Assuming all man and women is not matched,

mtow[i][j] a 2d array where each row represents every man and columns represents the women in declining preference.

wtom[i][j] 2d array vice versa from m to w.

GS algorithm (int mtow[][], int wtom[][]) {

while a free man m exists who still has to propose to a woman w

{

w = m's most desired w that he has not proposed to

for (every m in mtow)

{

if (w has no match) (m and w is engaged);

else if (w is engaged to m_2 but w prefers m over m_2)

{

m and w is engaged ;

m_2 is matchless ; }

else

{

m_2 and w remain engaged ;

m remains matchless ; }

}

returns match results.

}

For worst case complexity, assuming n men and women, The man looks for every single woman and is rejected. Therefore, time complexity is $O(n^2)$

b) When algorithm stops, all man and woman should be engaged and at worst case, a man proposes to everyone. And while being proposed to, w must engage with someone.

If M_1 and W_2 are both engaged but not to each other, it is not possible for both of them to prefer each other over their finalized partners. Because if W_2 preferred M_1 over her partner, she would have picked M_1 after she has engaged. However, she picked her current partner because she prefers her current one over M_1. Therefore, matching is stable.

c) Men = $\{M_1, M_2, M_3\}$
Women = $\{W_1, W_2, W_3\}$
Preferences
M to W = $\{\{W_1, W_2, W_3\}, \{W_3, W_1, W_2\}, \{W_1, W_3, W_2\}\}$
W to M_2 $\{\{M_1, M_2, M_3\}, \{M_2, M_1, M_3\}, \{M_3, M_2, M_1\}\}$

If W_3 wants to be with M_3, since M_3 wants W_1 more than W_3, W_3 can lie and engage with M_1 first. Then W_1 chooses M_2 over M_3 so M_3 becomes rejected and unengaged. When M_3 then chooses W_3, W_3 can misrepresent her preference and choose M_3 over M_1, leading her to a more favourable match.

http://www.cs.cornell.edu/courses/cs4820/2018fa/lectures/approx-onlinet.pdf

Problem 5

a) Show that the greedy function is arbitrarily bad.

Counter example: $W = 10$

object	weight	value	value/weight
A	4	10	2.5
B	3	6	2
C	2	4	2
D	2	3	1.5
E	1	0.5	0.5

According to Greedy algorithm, $\sum_{i=1}^{k} w_i < 10$ ordering all items in decreasing value/weight ratio:

A ($w_i = 4$, $\frac{v}{w} = 2.5$), B ($w_i = 3$, $\frac{v}{w} = 2$), C ($w_i = 2$, $\frac{v}{w} = 2$), D ($w_i = 2$, $\frac{v}{w} = 1.5$), E ($w_i = 1$, $\frac{v}{w} = 0.5$)

but k stops at object C because $w_i \leq W = 10$

hence total value stored is $10 + 6 + 4 = 20$.

However, this algorithm misses the opportunity of adding object E, where a C-approx algorithm will give max value of 20.5.

Hence, Greedy algorithm is arbitrarily bad.

https://people.seas.harvard.edu/~yaron/AM221-S16/lecture_notes/AM221_lecture17.pdf

b) Assuming OPT_a is the optimal solution from the original algorithm and OPT_b is the optimal solution for modified algorithm,

$OPT_b \leq OPT_A$ since b is modified and is a relaxation from a.

If the output of the greedy algorithm is $V(S)$,

$$OPT_b \leq OPT_a \leq V(S) + v_i \leq 2 \cdot \max(V(S), v_i)$$

Algorithm $\frac{1}{2}$ approximation is implied.

$$\text{greedy solution} \geq \frac{OPT}{2}$$

https://www.cs.princeton.edu/courses/archive/spring05/cos423/lectures/11approx-alg.pdf

Problem 6

a) Since one machine is required to run the job with the max t_j, and will have more t_js to add depending on how many jobs. The optimal make span L^* will always $\geq \max_{1 \leq j \leq n} t_j$.

Since total processing time is $\sum_{1 \leq j \leq n} t_j$ and assuming each machine perfectly splits up the load in equal time, $L^* = \frac{1}{m} \sum_{1 \leq j \leq n} t_j$. However, sometimes loads will be split inefficiently, therefore L^* will be increased.

$\therefore L^* \geq \frac{1}{m} \sum_{1 \leq j \leq n} t_j$.

b)

```
GreedyAlgorithm( int m, int n, int t_n[]) {
    for each machine i {
    L_i = 0 ;     - unload all makespan
    J(i) = φ ;    - J(i) is the subset of all jobs assigned to machine i.
    }
    for each job k {
        a = index of machine with lowest load ;
        J(a) = J(a) ∪ {k} ;   assign k to machine a
        L_a = L_a + t_k ;     adds time to total make spath
    }
}
```

If machine i has maximum load L_i and j is the last job assigned to i, before j was assigned i has the smaller load $= L_i - j$.

$L_i - t_j \leq L_a$ for all $a \leq m$. Therefore $(L_i - t_j) \leq \frac{1}{m} \sum_a L_a$

but $L^* \geq \frac{1}{m} \sum_a L_a$ and $t_j \leq L^*$

So $L_i = L_i - t_j + t_j \leq 2L^*$ and is a 2 approximation algorithm.

c)

```
Greedy Algorithm( int m, int n, int t_n[]) {
    Sort t_n in descending order;
        for each machine i {
        L_i = 0;      - unload all makespan
        J(i) = φ;     - J(i) is the subset of all jobs assigned to machine i.
        }
      for each job k {
            a = index of machine with lowest load;
            J(a) = J(a) ∪ {k};   assign k to machine a
            L_a = L_a + t_k;     adds time to total make span
      }
}
```

If machine i has maximum load T; and j is the last job assigned to i, before j was assigned i has the smaller load $= L_i - j$.

then same as last time,

$$L_i = (L_i - t_j) + t_j$$
$$\leq L^*$$

However, if there is more than m jobs, job m+1 must be $t_j \geq t_{m+1}$ because job is in decending order. Therefore, $t_j \leq t_{m+1} \leq \frac{1}{2} L^*$

Thus $L_i = (L_i - t_j) + t_j \leq \frac{3}{2} L^*$.

a $\frac{3}{2}$ approximation algorithm.

If job count is less than m, schedule is optimal.