Discrete Mathematics Problem Set 2

Department of Computer Science and Engineering
2021-16988 Jaewan Park

Problem 1

Proof. Suppose that there are a finite number($= n$) of Ulam numbers.

$$u_1,\ u_2,\ u_3,\ \cdots,\ u_{n-1},\ u_n$$

Let's consider an integer $N = u_{n-1} + u_n$. Since there is no Ulam number larger than u_n, N should be given as another sum of two Ulam numbers.

However, Since u_{n-1} and u_n are the two largest Ulam numbers, there cannot be a sum of two other Ulam numbers that gives N. This is a contradiction, thus there are infinitely many Ulam numbers.

□

Problem 2

The pseudocode is:

Algorithm 1 Search Problem

```
1: procedure TERNARY SEARCH ($x$: integer, $a_1, a_2, \cdots, a_n$: increasing integers)
2:     $i := 1$                                  ▷ left endpoint of interval
3:     $j := n$                                  ▷ right endpoint of interval
4:     while $i < j$
5:         $p := \lfloor (i * 2 + j)/3 \rfloor$   ▷ 1/3 point of interval
6:         $q := \lfloor (i + j * 2)/3 \rfloor$   ▷ 2/3 point of interval
7:         if $x \le a_p$ then $j := p$
8:         else if $x > a_q$ then $i := q + 1$
9:         else $i := p + 1$, $j := q$
10:    if $x = a_i$ then location := $i$
11:    else location := 0
12:    return location        ▷ returns the index $i$ where $a_i = x$, or 0 if $x$ is not found
```

Up to three comparisons, $i < j$, $x \le a_p$, $x > a_q$ are made at each stage in the while loop.

Let's consider the worst case where x is at the midpoint. After the first step the size of the list is 3^{k-1}, and after each step it decreases similarly until it becomes 3^1. When the size is 3^1, the last step occurs and gives $i = j = {}^{3^k+1}/_2$. After that, a comparison $i < j$ gives false and the while loop breaks. Finally the comparison $x = a_i$ gives true and returns the location.

The loop occurs k times with three comparisons per step, and there are two extra copmarisons after the final step. Thus $3k + 2 = \log_3 n + 2$ comparisons are made in total. The time complexity is $\Theta(\log n)$.

Problem 3

(a) False / Counterexample : $f(x) = x$, $g(x) = -x^2$

For $c = 1$, $k = 1$, the following is true.

$$x > k \Rightarrow |x| \leq c\left|-x^2\right|$$

Thus $f(x) = O(g(x))$. However, we cannot find c, k that makes $2^{f(x)} = O\big(2^{g(x)}\big)$.

If such c, k exists, the following should be true.

$$x > k \Rightarrow |2^x| \leq c\left|2^{-x^2}\right|$$

Then $c \geq 2^{x^2+x}$ for all $x > k$. However since $2^{x^2+x} \to \infty$ when $x \to \infty$, there is no such fixed c, k.

(b) True

Proof. For any integer n, we can obtain the following.

$$\begin{aligned} e^x &= \sum_{k=0}^{\infty} \frac{x^k}{k!} \\ &= 1 + x + \frac{x^2}{2} + \cdots + \frac{x^n}{n!} + \cdots \\ &\geq \frac{x^n}{n!} \end{aligned}$$

Substituting n for x gives

$$e^n \geq \frac{n^n}{n!}, \;\; n! \geq \left(\frac{n}{e}\right)^n$$

$$\therefore \log n! \geq \log\left(\frac{n}{e}\right)^n = n \log n - n$$

Also, since $n! = n \times (n-1) \times \cdots \times 1$, we can obtain

$$\log n! = \log\left(n \times (n-1) \times \cdots \times 1\right) \leq \log\left(n \times n \times \cdots \times n\right) = n \log n$$

$$n \log n - n \leq \log n! \leq n \log n$$

$$\therefore 1 - \frac{1}{\log n} \leq \frac{\log n!}{n \log n} \leq 1$$

Since $\lim\limits_{n\to\infty}\left(1 - \frac{1}{\log n}\right) = \lim\limits_{n\to\infty} 1 = 1$, we can obtain

$$\lim_{n\to\infty} \frac{\log n!}{n \log n} = 1$$

Therefore an integer N exists where

$$\forall \epsilon \in \mathbb{R}, \;\; n > N \Rightarrow \left|\frac{\log n!}{n \log n} - 1\right| < \epsilon$$

When $\epsilon = 0.5$,

$$n > N \Rightarrow 0.5 \leq \frac{\log n!}{n \log n} \leq 1.5$$

$$\therefore |\log n!| \leq 1.5|n \log n|, \;\; |\log n!| \geq 0.5|n \log n|$$

Therefore $\log n! = O(n \log n)$ and $\log n! = \Omega(n \log n)$, so $\log n! = \Theta(n \log n)$.

$\square$

Problem 4

(a) The pseudocode is:

Algorithm 2 Stable Matching Problem

```
procedure GALE-SHAPLEY ALGORITHM ($m_1, \cdots, m_n$: unengaged men, $w_1, \cdots, w_n$: unengaged women)
    while an unengaged man exists
        for $m$ := all men who are unengaged
            $w$ := most preferred woman whom $m_i$ has not yet proposed
            if $w$ is engaged then
                $m'$ := man who is engaged to $w$
                if $w$ prefers $m$ to $m'$ then
                    $m' \leftarrow$ unengaged
                    $(m,\ w) \leftarrow$ engaged
            else
                $(m,\ w) \leftarrow$ engaged
```

Let's consider the worst case where all men and women have the same order of preference. Without loss of generality, assume the order is $m_1 < m_2 < \cdots < m_n$ and $w_1 < w_2 < \cdots < w_n$.

Consider the kth repeat of the **while** loop. The **for** loop repeats $n-k+1$ times, and makes two comparisons each in the **if** statement except the first case, which makes one. (Since w is equal for all m_i, w is always engaged to the previous man except the first case.) The **while** loop repeates n times, from $k = 1$ to $k = n$. Therefore the total number of comparisons is

$$\sum_{k=1}^{n} (1 + 2(n-k)) = n^2 - n + 1$$

The time complexity is $\Theta(n^2)$.

(b) The proof can be done in two steps.

(1) The Gale-Shapley algorithm guarantees perfect matching.

Proof. Suppose there is a case where the output is not a perfect matching. Then at least one man and woman who are unengaged must exist.

Applying this algorithm gives two observations. First, if a woman is once engaged, she stays or only gets engaged to better men and never becomes unengaged. Second, if a man is once engaged, he stays or only gets engaged with worse women.

According to these observations, the situation results in a contradiction. For the man and woman to remain unengaged, the man should have proposed to the woman before and got rejected. To become rejected, there should have been another better man who proposed at that time or later, and the woman should have been engaged to that man. However, the first observation states that a woman once engaged never becomes unengaged. Therefore this is a contradiction, and the algorithm guarantees perfect matching. □

(2) The output of the Gale-Shapley algorithm is stable.

Proof. Suppose there is a case where the output is not stable. There should exist a man m engaged to a woman w' and a woman w engaged to a man m' but both prefer each other over their pairs.

m should have proposed to w prior to w'. If m and w got engaged to each other at that time, w cannot get engaged to a man worse than m, according to our first observation. However, the result shows that w is engaged to m', who she prefers less than m.

Therefore m and w wouldn't have got engaged to each other at that time. This indicates there was a man better than m who proposed to w at that time, and the two got engaged to each other. However, since m' is worse than m, he is also worse than this man who got engaged with w. This is a contradiction, since w got engaged to someone worse than her previous pair. Therefore the output of the algorithm is always stable.

□

(c) Let's consider a case where $n = 3$ and the order of preferences is the following.

$$m_1 : (w_3, w_1, w_2),\ m_2 : (w_1, w_3, w_2),\ m_3 : (w_1, w_2, w_3)$$
$$w_1 : (m_1, m_2, m_3),\ w_2 : (m_2, m_1, m_3),\ w_3 : (m_2, m_1, m_3)$$

The original match results in $(m_1, w_3), (m_2, w_1), (m_3, w_2)$. However, if w_1 misrepresents her preferences as (m_1, m_3, m_2) she can get a better result. The result is $(m_1, w_1), (m_2, w_3), (m_3, w_2)$.

Problem 5

(a)

Proof. Let's consider a case where there are two items i and j, where $v_1 = 2$, $w_1 = 1$ and $v_2 = W$, $w_2 = W$. The goal of the algorithm is to maximize the total value. Since $2/1 > W/W$, the greedy algorithm will take only the first item, while the optimal input is taking the second item.

If this algorithm is a c-approximation algorithm, c should satisfty the following.

$$2 \geq cW,\ c \leq \frac{2}{W}$$

However when $W \to \infty$, $\frac{2}{W} \to 0$, so a fixed positive real number c doesn't exist. Therefore the algorithm is arbitrarily bad.

□

(b)

Proof. Suppose the items are sorted in increasing value/weight order. Let's say the items selected by the greedy algorithm are $1, 2, \cdots, k$. We can think of a more general problem:

$$\text{when } \sum_{i=1}^{n} w_i x_i \leq W, \text{ maximize } \sum_{i=1}^{n} v_i x_i \ \ (\forall i,\ 0 \leq x_i \leq 1)$$

The knapsack problem is a typical case of this problem, when $x_i = 0$ or 1 for all i. Therefore the optimal input of the knapsack problem is also a possible input of this problem, and the optimal result of this problem should be larger than the optimal result of the knapsack problem. Let's call each OPT$'$ and OPT.

Since the items are sorted, if we lessen x_i and greaten x_j when $i < j$, the result will become smaller. For example let's say $x_i = p$, $x_j = q$ initally, and we change the value to $x_i = p'$, $x_j = q'$ to match the whole weight. Then we can obtain

$$pw_i + qw_j = p'w_i + q'w_j,\ \frac{w_j}{w_i} = \frac{p - p'}{q' - q}$$

Since $i < j$ and the items are sorted, $\dfrac{v_i}{w_i} > \dfrac{v_j}{w_j}$. Therefore $\dfrac{v_j}{v_i} < \dfrac{w_j}{w_i} = \dfrac{p-p'}{q'-q}$, and the sum becomes smaller.

$$(pv_i + qv_j) - (p'v_i + q'v_j) = (p-p')v_i + (q-q')v_j > 0$$

Therefore if we lessen x_i and greaten x_j when $i < j$, the result will become smaller. The optimal input will give more weight on items with smaller i. Therefore the following is the optimal input of this new problem.

$$x_i = \begin{cases} 1 & (1 \le i \le k) \\ \dfrac{W - (w_1 + \cdots + w_k)}{w_{k+1}} & (i = k+1) \\ 0 & (k+1 < i \le n) \end{cases}$$

Since we know $\mathrm{OPT}' > \mathrm{OPT}$,

$$\begin{aligned} \mathrm{OPT}' &= v_1 + v_2 + \cdots + v_k + \frac{W - (w_1 + \cdots + w_k)}{w_{k+1}} v_{k+1} \\ &> \mathrm{OPT} \end{aligned}$$

Also, since $\dfrac{W - (w_1 + \cdots + w_k)}{w_{k+1}} < 1$ we can obtain $v_1 + v_2 + \cdots + v_{k+1} > \mathrm{OPT}' > \mathrm{OPT}$.

Let's say the item with maximum value is l. Then $v_l \ge v_{k+1}$, and this gives

$$v_1 + v_2 + \cdots + v_k + v_l \ge v_1 + v_2 + \cdots + v_k + v_{k+1} > \mathrm{OPT}$$

Therefore, either $v_1 + v_2 + \cdots + v_k > \dfrac{\mathrm{OPT}}{2}$ or $v_l > \dfrac{\mathrm{OPT}}{2}$.

The modified greedy algorithm for the knapsack problem selects either $1, 2, \cdots, k$ or only l. Therefore the result given by the modified algorithm is always larger than $\dfrac{\mathrm{OPT}}{2}$, so this is a $1/2$-approximation algorithm. □

Problem 6

(a)

Proof. Let's say $\max_{1 \le j \le n} t_j = t_J$, and job J is assigned to machine K. This gives $L_K \ge t_J$. Since L^* is a makespan, it should be the maximum of all loads. Therefore we can obtain the following.

$$L^* \ge L_K \ge t_J = \max_{1 \le j \le n} t_j$$

Also, the sum of all loads should be equal to the sum of all job times.

$$\sum_{k=1}^{m} L_k = \sum_{j=1}^{n} t_j$$

Since L^* is a makespan, it is larger than all other loads. Therefore we can obtain the following.

$$L^* = \frac{1}{m} \sum_{k=1}^{m} L^* \ge \frac{1}{m} \sum_{k=1}^{m} L_k = \frac{1}{m} \sum_{j=1}^{n} t_j$$

□

(b) The pseudocode is:

Algorithm 3 Load Balancing

```
procedure GREEDY ALGORITHM ($m$: integer, $t_1, \cdots, t_n$: time)
    for $j := 1$ to $n$
        $min :=$ MAXIMUM INTEGER
        for $k := 1$ to $m$
            if $L_k < L_{min}$ then $min := k$
        $L_{min} := L_k + t_j$
    $makespan :=$ MINIMUM NUMBER
    for $k := 1$ to $m$
        if $L_k > makespan$ then $makespan := L_k$
    return $makespan$
```

Proof. Let's say machine p has the maximum load. Also let's say q the last job assigned to K. Then before q was assigned, p would have had the smallest load. Therefore L_p at that step would have been same or smaller than the average load. Also, if J is the job that takes the longest time, $t_q \leq t_J$. This gives

$$L_p = (L_p - t_q) + t_q \leq \frac{1}{m}\sum_{j=1}^{n} t_j + t_J$$

From **(a)**, we know $L^* \geq \frac{1}{m}\sum_{j=1}^{n} t_j$ and $L^* \geq t_J$ where L^* is the makespan. This gives

$$L_p \leq \frac{1}{m}\sum_{j=1}^{n} t_j + t_J \leq 2L^*$$

Therefore the greedy algorithm is a 2-approximation algorithm.

□

(c)

Proof. Let's say machine p has the maximum load. we can consider two possible situations.

First, if p has only one job assigned, it is optimal. Therfore this gives

$$L_p \leq \max_{1\leq j\leq n} t_j \leq 1.5 \max_{1\leq j\leq n} t_j \leq 1.5L^*$$

Second, if p has more than one job assigned, there are at least $m+1$ jobs. Considering the first $m+1$ jobs, in the optimal case there should be a machine that has at least two jobs assigned. Thus optimal makespan is same or larger than the sum of the time of these two jobs. Also, since the jobs are sorted by time in decreasing order, all of the first $m+1$ jobs take longer than t_{m+1}. This gives

$$L^* \geq 2t_{m+1}$$

Now let's consider the situation in **(b)**. Since p has more than one job assigned, $q \geq m+1$. From that the jobs are sorted, we can obtain

$$t_q \leq t_{m+1}$$

Therefore, the makespan that the modified algorith gives satisfies the following.

$$L_p = (L_p - t_q) + t_q \leq \frac{1}{m}\sum_{j=1}^{n} t_j + t_{m+1} \leq L^* + \frac{1}{2}L^* = \frac{3}{2}L^*$$

Therefore the greedy algorithm is a $^3/_2$-approximation algorithm.

$\square$