1. (Chapter 1 Problem 5 of the text book)

a) **Modified G-S algorithm**

1. Initially, every element is unpaired.
2. For every unpaired m, let

$$W_m = \{\, w \mid w \in W \text{ and } m \text{ has not tried to pair}\,\}.$$

Let $w \in W_m$ with the largest $f_m(w)$.
If there is more than one instance of w with the largest $f_m(w)$, randomly pick one (or the first one in the row).
If w is unpaired, then put (m, w) into a pair.
Otherwise (there is a pair (m', w)):

- if $g_w(m) > g_w(m')$, then remove pair (m', w) and put (m, w) as a pair,
- otherwise mark w as one that m has tried to pair.

Prove that there is no *strong instability*: Consider the following properties:

i. Once a $w \in W$ is paired, w remains paired.

ii. For pairs (m_1, w) and (m_2, w), with (m_1, w) created earlier than (m_2, w), we have

$$g_w(m_1) \le g_w(m_2).$$

Since S is a matching and $|M| = |W| = n$, by Property (1), if there is an unpaired m then there must also exist an unpaired w such that m has not yet tried to pair with w. Therefore, every m will eventually pair with some w, and the algorithm returns a perfect matching S at termination.

Now assume, for contradiction, that there exists a strong instability with respect to S. Then there exists a pair $(m, w) \in (M \times W) \setminus S$ such that $(m, w'), (m', w) \in S$ and

$$f_m(w) > f_m(w') \quad \text{and} \quad g_w(m) > g_w(m').$$

This means that m must have tried to pair with w before it was paired with w'. However, since $(m, w) \notin S$, Property (2) implies that

$$g_w(m) \le g_w(m'),$$

a contradiction.

Therefore, S is stable.

b) **Counterexample.** Consider the following instance. There are two men $M = \{m_1, m_2\}$ and two women $W = \{w_1, w_2\}$, where each man and each woman is indifferent among their options.

There are two possible perfect matchings:

i. $S_1 = \{(m_1, w_1), (m_2, w_2)\}$. In this case, there is a weak instability since

$$f_{m_1}(w_2) = f_{m_1}(w_1) \quad \text{and} \quad f_{w_2}(m_1) = f_{w_1}(m_2),$$

which is a special case of

$$f_{m_1}(w_2) \geq f_{m_1}(w_1) \quad \text{and} \quad f_{w_2}(m_1) \geq f_{w_1}(m_2).$$

ii. $S_2 = \{(m_1, w_2), (m_2, w_1)\}$. Similarly, there is a weak instability since

$$f_{m_1}(w_2) = f_{m_1}(w_1) \quad \text{and} \quad f_{w_2}(m_1) = f_{w_1}(m_2),$$

which is a special case of

$$f_{m_1}(w_1) \geq f_{m_1}(w_2) \quad \text{and} \quad f_{w_1}(m_1) \geq f_{w_1}(m_2).$$

Thus, in this setting there is always a weak instability.

2. (Chapter 1 Problem 6 of the text book) Let

$$G_d = \{(s, p) \mid \text{ship } s \text{ visited port } p \text{ at day } d\}$$

denote the given schedule of the day each ship is visiting each port. (If the scahdule is given in a different structure, create this in $O(n \times m)$

Initially:

- $UP := \varnothing$ (every ship is unpaired).
- $US := \varnothing$ (every port is unpaired).
- $S := \varnothing$ (Truncated Schedule).

Algorithm:

1. **for** i = m to 1, **do** /*$O(m)$ times */
   1.1. fo reach (s, p) in G_d /*$O(n) times$*/
   1.2. **If** $s \in US$ and $p \in UP$ **then** /*$O(1)$*/ $S := S \cup \{(s, p, d)\}, US = US \cup \{s\}, UP = UP \cup \{p\}; / * O(1) * /$
2. **End for**
3. **Return** S.

Running time is $O(n \times m)$.

S denote each ship s should stay at port p from day d on.

Prove: Imagine that, by contradiction, ship s is not assigned to a port. Since there are n ships and n ports, there is a port p that is not assigned to a ship too. Since, by the set up, each ship is visiting all ports, there should be a (s, p, d). Thus in iteration d, ship s should have been matched with port p, since both of them are unmatched. which is a contradiction. Thus, the algorithm returns a perfect match. Imagine by contradiction, that ship s' must visit port p at day d', but ship s is matched with port p at day $d \leq d'$. By the question, $d \neq d'$, thus $d < d'$. since ship p is matched with port p at day d, the port p is not matched with any ships for day $d' > d$. Thus, ship s must have merged with port p at day d'. Which is a contradiction.

3. (Chapter 2 Problems 1 and 3 of the text book.)

a) i. $n^2 < 3.6 \times 10^{13} \;\Rightarrow\; n < (3.6 \times 10^{13})^{1/2} \;\approx\; 6.0 \times 10^6$

ii. $n^3 < 3.6 \times 10^{13} \;\Rightarrow\; n < (3.6 \times 10^{13})^{1/3} \;\approx\; 3.3 \times 10^4$

iii. $100n^2 < 3.6 \times 10^{13} \;\Rightarrow\; n < \left(\frac{3.6 \times 10^{13}}{100}\right)^{1/2} \;\approx\; 6.0 \times 10^5$

iv. $n \log_2 n < 3.6 \times 10^{13} \;\Rightarrow\; n <$ solution of $n \log_2 n = 3.6 \times 10^{13} \;\approx\; 9 \times 10^{11}$

v. $2^n < 3.6 \times 10^{13} \;\Rightarrow\; n < \log_2(3.6 \times 10^{13}) \;\approx\; 45$

vi. $2^{2^n} < 3.6 \times 10^{13} \;\Rightarrow\; n < \log_2\Big(\log_2(3.6 \times 10^{13})\Big) \;\approx\; 5.49 \;\Rightarrow\; n = 5$

b) $f_2(n) = \sqrt{2n}$, $f_3(n) = n + 10$, $f_6(n) = n^2 \log n$, $f_1 = n^{2.5}$, $f_4(n) = 10^n$, $f_5(n) = 100^n$.

4. (Chapter 2 Problem 6 of the text book)

The algorithm runs for $\sum_{i=2}^{n} n * (n - i)$ iterations. In each iterations $(j - i)$ additions. in total:

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} (j - i) = \frac{n^3 - 1}{6}$$

if each addition takes k primitive operations, then the running time is:

$$\frac{n^3 - 1}{6} \times k$$

a)

$$\frac{n^3 - 1}{6} \times k \leq \frac{n^3}{6} \times 2k, \text{ for } n > 1$$

since $\frac{n^3}{6} \times 2k$ is $O(n^3)$, then the running time of the algorithm is $O(n^3)$

b)

$$\frac{n^3}{6} \times .5k \leq \frac{n^3 - 1}{6} \times k \text{ , for } n > 1$$

since $\frac{n^3}{6} \times .5k$ converges to n^3, then the running time of the algorithm is $\Omega(n^3)$

c) **for** $i = 1, 2, .., n$ **do**
$s = 0$
  for $j = i + 1, i + 2, .., n$ **do**
    $s = s + A[j]$
    Store s in $B[i, j]$
  endfor
end for

The algorithm perform 1 addition operation per each iteration

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} 1 = \frac{n^2 - n}{2}$$

The running time os $O(n^2)$

5. (Chapter 3 Problem 2 of the text book) **Answer 1**: Initialize: $Visited = \varnothing$ **While** there is a node s in $S \setminus Visited$, **do**: /*$O(n)$ times */ mark as s visited ($Visited = Visited \cup \{s\}$) **for** each neighbor u of s, **do**: /*$O(m)$ times */ **if** u is not marked $Visited$, **then**: $DFS(u, s)$ **end if End for End while Return** cycle **DFS(u,s)** $\{$ **for** each neighbor v of u, other than s, **do**: **if** $v \in Visited$, then $Cycle = 1$. **else**: $Visited = Visited \cup \{v\}$ $DFS(v, u)$ **end if end for**
$\}$ Running time is $O(m + n)$ (Similar to the original DFS) **Prove**: Imagine there is a cycle s_1, s_2, ..., s_m, s_1. then starting from s_i, $1 \leq i \leq m$, the DFS algorithm visit s_i again. Since s_i is already visited, then the algorithm returns a cycle. Imagine there is no cycle. Thus there is no node that can be visited two times starting from another point. Thus, DFS will not return a cycle. **Answer 2:** Define connected component

$$C_s = \{v | v \in N \text{ and there is a path between } s \text{ and } v\}$$

Also:

- $V(s) = |C_s|$
- $E(s) = \frac{\sum_{u \in C_s} deg(u)}{2}$

There is a cycle iff for each components of graph G, with more than 2 noted:

$$V(s) \leq E(s)$$

I use the following algorithm to find V(s) and E(s):

Algorithm (DFS-based cycle detection).

While there exists an unvisited node, **do**:
  Pick s from the set of unvisited nodes.
  $V(s) := 1$, $N(s) := 0$.
    DFS$(s, V(s), N(s))$
  $E(s) := N(s)/2$
  If $V(s) \leq N(s)$, **then** there is a cycle; **break**.

End while{no cycles detected}

DFS($s, V(s), N(s)$) {
For each neighbor v of s:
Update $N(s) := N(s) + 1$.
If v is not visited, **then**
$V(s) := V(s) + 1$
$DFS(v, V(s), N(s))$.
end if
End for }

Analysis: Running time is as much as DFS, which is O(m+n) since each node is visited only once and each edges is visited at most one time.

6. (Chapter 3 Problem 10 of the text book)

1. **Initialize**: $N := 0, l = 1, Visited = \varnothing$, $Max = 0$ /*$O(1)$ */
2. Apply BFS algorithm from point v. /* Running time is $O(m + n)$ */

- $L_0 = s$
- $L_1 = \{$all nodes adjacent to $s\}$
- For $i > 1$:
  - $L_i = \{$nodes not in $L_0 \cup L_1 \cup .. \cup L_{i-1}$ and adjacent to a node in $L_{i-1}\}$

3. If $w \in L_l$, then $Max = l$. If w is not in any L_l, there is no path between v and w so return 0. so far.

4. **Do** $mDFS(v, 1)$ /* Running time is $O(m)$ */
$mDFS(s, l)$:

- **for** neighbor u_l of s in $L(l)$, **do**
  - **if** $l \leq Max$ **then do**:
    - **if** $u_l = w$, **then** $N = N + 1$
    - **else** $mDSF(u_l, l + 1)$
    - **end if**
  - **end if**
- **end for**

Return N.

The algorithm might visit some nodes more than one time, but it runs at most as much as number of all edges in L_{i-1} to L_i, for $i \leq Max$, which is less than the total number of edges, m. Thus the running time is $O(m)$. Total running time is $O(m + n)$

Prove:
BFS find the shortest path, Max.
mDFS(v, 1) find all path from v to w. All shortest paths consist of one edges between L_{i-1} to L_i, for $i \leq Max$. Since all of these edges are tried in the algorith, it returns the number of shortest path.

7. (Chapter 3 Problem 11 of the text book)

Algorithm (Infection Spread).

Initialize:
$G := \big((C_i, C_j, t_k) \mid t_x \leq t_k \leq t_y\big)$ sub-sequence of sorted triples)
$m' := |G| \leq m$.
I := $\{C_0\}$.

For $q = 1$ to m': /* m iterations */
  Pick (c_q, c'_q, t_q) as the q-th element of G (i.e., with the q-th lowest t_k).
   If $c_q \in$ Infected and $c'_q \notin$ Infected, **then** /* at most n times */
     I := I $\cup \{c'_q\}$;
   Else if $c'_q \in$ Infected and $c_q \notin$ Infected, **then** /* at most n times */
     I := I $\cup \{c_q\}$;
   End if
End for
If $c_b \in$ Infected, **then** print "C_b **is infected**".
Else print "C_b **is not infected**".
End if
Each triples is checked ones, and each node is marked infected at most 1 time. Thus the running time is $O(m + n)$

Prove by induction: At iteration 0 (time t_x, once only c_0 is infected), machine c is infected iff $c \in I$.

By induction, assume that at iteration q, if c is infected, it is in I_q. Imagine for contradiction that c' is infected, but it is not in I_{q+1} at iteration $q+1$. if c' is in I_q, then since $I_q \subset I_{q+1}$, then c' in I_{q+1}. which is a contradiction. If c' is not in I_q, then it is infected before at t_{q+1}. If it is infected at t_{q+1}, then it should have exchanged with an infected c''. If did that, by the algorithm it should have been added to I_{q+1}. Which is contradiction. By induction, assume that if c is in I_q, it is infected. Then, at iteration $q+1$, c' is added to I_q if and only iff, c' is not already infected and exchange data with $c \in I_q$. If exchange data with $c \in I_q$, it is infected. Thus, c' in I_{q+1} at iteration $q+1$ is also infected.