Problem 1)

1) To construct a preference list that ensures no more than $O(n)$ rounds, each student should be matched with their top choice college and each college should rank the student who prefers it the most as its top choice.

→ For students (1 to n):

student i should have a list which looks like [College(i), College($i+1$) ... College(n), College(1)... College($i-1$)]

→ For colleges (1 to n)

College i should have a preference list which looks like

[Student(i), Student($i+1$) .... Student (n), Student (1) .... Student ($i-1$)]

- Each student proposes to their top choice college and thus each college receives a proposal which are accepted since all colleges are unmatched.
- This result in just one round of proposals making the run time $O(n)$.

~~Problem 1)~~ Problem 1)

2) To design a list so the number of rounds in the algorithm is $\Omega(n^2)$ we must cause the maximum ~~possible~~ possible number of ~~propo~~ proposals before a stable matching is found.

To construct such a list:

Students: Each student should have the same list: [College 1, College 2, ... College n]

Colleges: Each college ranks the students in reverse order:

College 1: [Student(n), Student($n-1$), ... Student(1)]
College 2: [Student($n-1$), Student($n-2$), ... Student 1, Student(n)]

⋮

College n: [Student(1), Student(n), Student($n-1$), Student($n-2$)...]

Explanation:

- Every student proposes to college 1 in the first round but only student (n) gets matched.
- In round 2, all ~~all~~ unmatched students propose to college 2 but only student ($n-1$) gets matched
- Student 1 gets matched only at the end with college n, in the last round of proposal.

~~[illegible]~~

- Student 1 makes n proposals, Student 2 makes $(n-1)$ proposals ..... student n makes 1 proposal.

$$n(n-1)(n-2)\ldots 1 = \frac{n(n+1)}{2} \text{ which is } \Omega(n^2)$$

Problem 2)

1) We can modify the Gale-Shapely algorithm to handle ties.

→ Initialization: all students and colleges start unmatched

→ While there is an unmatched student as who has not proposed to every college:

→ S proposes the the highest ranked colled which it hasn't yet proposed to (college c) [any college if there is a tie]

→ If college c is free it gets matched with S.

→ else if c prefers s over its current partner S' or there is a tie, c pairs with s and S' becomes free.

→ else if c prefers its current partner over s, s gets rejected.

- This algorithm definitely terminates because each student proposes to each college at most 1 time.

- There cannot be a strong instability as:

→ Suppose there was a strong instability (s, c) where s is matched to c' and s' is matched to c.

→ This would mean s prefers c to C' and c prefers s to S'.

→ However if that is the case S must have proposed to c befor c' and since they aren't match c must have rejected S in favor of S' or some other student c preferred over s.

→ this contradicts c prefering s over s'.

∴ No pair will strictly prefer each other over their current match as any such pair would already be match or would be indifferent which does not qualify as strong instability.

Problem 2)

2) Counter Example to show that there does not always exist a perfect matching with no weak instabilities.

→ Student list

: S1 prefers C_1 and C_2 equally

: S2 prefers C_1 over C_2

→ College list:

: C1 prefers S_2 over S_1

: C_2 prefers S_2 and S_1 equally

If S_1 matched with C_1 and S_2 matched with C_2

- (S_2, C_1) is a weak instability

If S1 is matched with C_2 and S_2 is matched with C_1

- (S_1, C_1) is a weak instability

Therefore, each matching has atleast 1 weak instability and it happens because of the introduction of ties in preferences.

Problem 3)

1) The algorithm ends up having 3 nested loops:

→ outer loop from 1 to n.

→ middle loop for $(i+1)$ to n which runs at most n times.

→ the loop for computing the maximum which runs atmost n times for each pair (i, j)

The total number of operations is bounded by $n \times n \times n = n^3$. and the running time is $O(n^3)$. In the worst case we need to scan upto n elements, for each i and j, to find maximum. giving us 3 nested loops with a worst case of n iterations each.

2) For $i=1$ and $j=n$, which always occurs, it requires scanning all n elements for the maximum and it happens $\frac{n(n-1)}{2}$ times for all pairs $(i<j)$. So lower bound is $\frac{n(n-1)}{2} n$

$= \frac{n^3 - n^2}{2}$ which is $\Omega(n^3)$. Therefore the algorithm has $O(n^3)$ and $\Omega(n^3)$ and thus $\Theta(n^3)$.

3) We can try to come up with a faster algorithm by not recomputing the maximum from scratch each time but using the logic that: $B[i,j] = \max(B[i, j-1], A[j])$ and we can thus calculate maximum for $B[i,j]$ incrementally using previous value of $B[i, j-1]$

→ We need to set $B[i,i] = A[i]$ for all i

→ We need to iterate over each i from 1 to n.

→ for each i we need to iterate over j from $i+1$ to n using relation $B[i,j] = \max(B[i,j-1], A[j])$

∴ The inner loop only performs one comparison to calculate the maximum per iteration

Analysis and Verification :

- Outer loop runs n times .
- Inner loop : for each i, runs $n-i$ times . Total iterations are : $\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$

- Each inner loop iteration requires on one comparison, and is constant time .

So, $T(n) = O(n^2)$

$$\lim_{n \to \infty} \frac{T(n)}{f(n)} = \lim_{n \to \infty} \frac{n^2}{n^3} = \lim_{n \to \infty} \frac{1}{n} = 0$$

∴ Suggested ~~algorithm~~ algorithm is faster than original algorithm .

Problem 4)

We need to minimize the worst case number of drops :

- With two jars, we can use the first one to eliminate large sections of the ladder, and then use second one to do a linear search of the remaining section.
- We can define a step size S. We can drop for S, then $2S$, $3S$ .... and so on until it breaks.
- If the first jar breaks at ik, we know highest safe rung is between ik and $(i-1)k$
- We can do a linear search in these k rungs.
- This means at most i drops for first jar and $k-1$ drops for second jar.

$$n \approx ik \quad \therefore \quad i \approx n/k$$

$$f(n) \approx \frac{n}{k} + k$$

$$\frac{d}{dk}\left(\frac{n}{k}+k\right) = -\frac{n}{k^2} + 1 = 0 \Rightarrow k \approx \sqrt{n}$$

$\therefore$ optimal k size is $\sqrt{n}$ steps

Strategy will be to :

→ drop first jar from $\sqrt{n}$, $2\sqrt{n}$, $3\sqrt{n}$ .... until it breaks

→ use second jar to do linear search in last $\sqrt{n}$ range

$f(n)$ will be $\approx 2\sqrt{n}$

$$\lim(n \to \infty)\ f(n)/n = \lim(n \to \infty) \frac{2\sqrt{n}}{n} = \lim(n \to \infty) \frac{2}{\sqrt{n}} = 0$$

$\therefore$ This strategy grows slower than $O(n)$.