Implementation of G-S Algorithm.

For concreteness, lets think of the procedure implementing the algorithm as taking as parameters:

- n, the number of men/women. Lets think of the men as being numbered $0, \ldots, n-1$ and the women as being numbered $0, \ldots, n-1$.

- MRanking[ ][ ], an $n \times n$ array whose $(i,j)^{th}$ entry MRanking[i][j] is a number between 0 and $n-1$. This entry specifies man i's rank of woman j. (If MRanking[i][j] = 0, j is the woman i likes best.)

- WRanking[ ][ ], where WRanking[i][j] is woman i's rank of man j.

The algorithm will create and use some additional data structures:

- MPrefs [ ] [ ] - an $n \times n$ array where MPrefs [i] [j] gives index of the woman whom Man i ranks j. So MPrefs [i] [0] is woman whom man i likes best.

- lastproposal [ ] - an integer array of size n where lastproposal [i] is the rank of woman to whom i made last proposal. Initialized to -1.

- FreeList - a linked list containing indices of all men who are currently free.

- Partner [ ] – an integer array of size n where Partner [i] is man to whom woman i is engaged, if she is engaged.

- Engaged [ ] – an ~~integer~~ boolean array of size n where Engaged [i] is true iff woman i is engaged.

- Instantiate arrays Mprefs, lastproposal, Partner, engaged

- For i ← 0 to n-1 do
    lastproposal [i] ← -1
    engaged [i] ← false
  endfor

- For i ← 0 to n-1 do
    For j ← 0 to n-1 do
      MPrefs [i] [MRanking [i][j]] ← j
    endfor
  endfor

- Initialize FreeList.

- For i ← n-1 down to 0 do
    FreeList. insert (i)
  endfor

```
While (FreeList is not empty) do
    m ← FreeList.head()
    lastproposal[m] ← lastproposal[m] + 1
    w ← MPrefs[m][lastproposal[m]]
    If (not engaged[w])
        engaged[w] ← true
        partner[w] ← m
        FreeList.remove()

    else
        m' ← partner[w]
        If WRanking[w][m] < WRanking[w][m']
            partner[w] ← m
            FreeList.remove()
            FreeList.Insert(m')
        endif
    endif
endwhile
```

Analyzing the Running time.

Since the number of pseudo-code step executions increases (generally) with n, let us bound the number of steps executed as a function of n.

We count each basic pseudo-code/ java code step execution as 1 step. To achieve machine/language/compiler independence, we will only be interested in bounding the total number of steps up to a multiplicative constant.

Doing the Counting, we obtain:

There exist constants $a > 0$, $b > 0$, $c > 0$ so that the running time (the ~~num~~ total no. of executions of basic pseudo-code steps) is bounded by $an^2 + bn + c$.

Define the worst-case running time for a given n to be maximum running time over all instances with n men and n women.

Clearly, worst-case running time(n)

$$\leq an^2 + bn + c.$$

We can also lower bound the running time.

For each n, worst case running time(n) is at least ϵn^2, for some constant $\epsilon > 0$.

In fact, we can make the following statement, because of the steps that create MPrefs[ ][ ]

For each n, running time on every instance of size n is at least n^2.

Our statements about the ~~effi~~ running time are an attempt to quantify efficiency - no. of executions of basic steps is a very good indicator of wall clock time that an implementation takes to execute.

But why not just do the following:
Take n to be in the range that we care about.
Sample a few instances with n men and women. Measure wall clock time for executions on those instances. This gives some estimate of how good the algorithm is.

Such an approach is indispensible if you are an algorithms expert in the context of a larger system.

But the approach has some inadequacies, which we should try to overcome <u>when</u> possible:

- What if the range of values of n we care about changes?
- What if the "distribution of realistic inputs" changes?
- Ultimately, the approach does not give an insight into why a certain algorithm is efficient and some other algorithm is not.