L.68 ----組み合わせの配列を格納した配列（2 次元配列）を生成----

(本番では 28C14 を想定して計算するが)

たとえば 5C3 なら

[[0,1,2],[0,1,3],[0,1,4],[0,2,3],[0,2,4],

[0,3,4],[1,2,3],[1,2,4],[1,3,4],[2,3,4]]

のように 0 から始まる組み合わせの配列を生成

L.79 --コンビネーションに対応する着座表を作成--

座席の位置(行・列)のみ格納した「小さな配置図」の中から、

組み合わせ配列にしたがって、「人が座る位置」だけ取り出した配列を作成

L.85 --不快度の計算の下準備--

いちいち大きな座席表を作って計算するわけにもいかないので、

各行・各列に着席している人数をそれぞれ

total_people_row, total_people_column という配列に格納

その後のループは、

その行・列にいる人の数と、その人達の向いている方向により 4 通り場合分け

sum_eyesight に視界の広さ×人数分

sum_counted_people に視界に入る人の数×人数分

を足していく

不快度は、視界に入る人の数の総和 / 視界の広さの総和 と定義する

すなわち、不快度 = sum_counted_people / sum_eyesight

※ 大きな座席図

[[2. 0. 2. 2. 0. 2. 2. 0. 2. 2. 0.]

[0. 3. 0. 0. 3. 0. 0. 3. 0. 2. 3.]

[2. 0. 2. 2. 0. 2. 2. 0. 2. 2. 0.]

[2. 0. 2. 2. 0. 2. 2. 0. 2. 2. 0.]

[0. 3. 0. 0. 3. 0. 0. 3. 0. 2. 3.]

[2. 0. 2. 2. 0. 2. 2. 0. 2. 2. 0.]]

※ 小さな座席図（座席がある場所の行・列）

[[0. 1.]

[1. 0.]

[1. 2.]

[2. 1.]

[0. 4.]

[1.　3.]
[1.　5.]
[2.　4.]
[0.　7.]
[1.　6.]
[1.　8.]
[2.　7.]
[0. 10.]
[2. 10.]
[3.　1.]
[4.　0.]
[4.　2.]
[5.　1.]
[3.　4.]
[4.　3.]
[4.　5.]
[5.　4.]
[3.　7.]
[4.　6.]
[4.　8.]
[5.　7.]
[3. 10.]
[5. 10.]]

※ 「人が座る位置」の配列

上記の「小さな座席図」配列から、
各コンビネーションで指定されたインデックスに対応する組を取り出す
例：
コンビネーションが[0,1,4,6,17,24]と与えられた場合
小さな座席図 配列の[0,1,4,6,12,24]番目を取り出すので
[[0. 1.]
 [1. 0.]
 [0. 4.]
 [1. 5.]
 [5. 1.]
 [4. 8.]]
以上が「人が座る位置」の配列である

※ 実際のコードでは、

大きな座席図：ARRANGEMENT, min_arrangement, max_arrangement

小さな座席図：seat_position

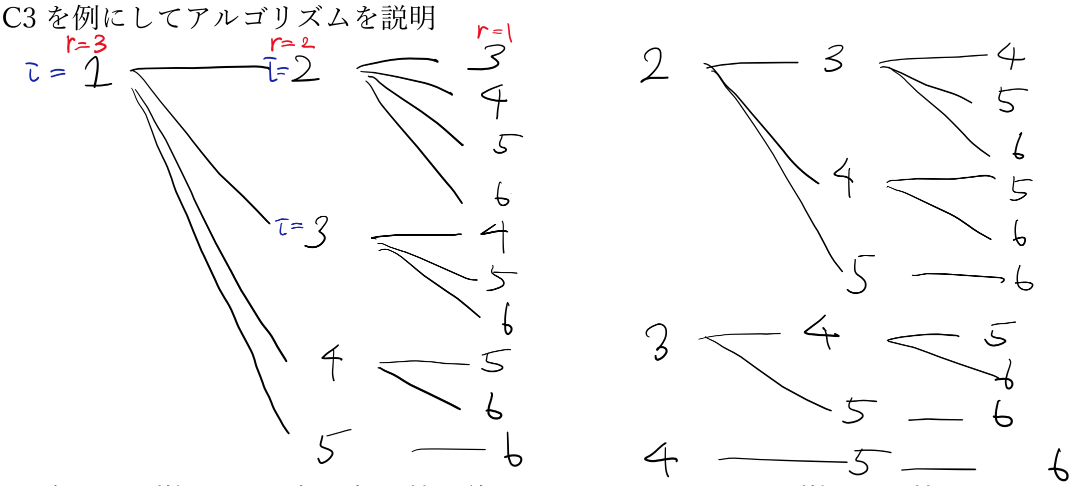
人が座る位置の配列：(ループ内)sat_seat_position, min_seat_position,
max_seat_position

コンビネーション配列：(ループ内)each_combination, combination_list[min_index],
combination_list[max_index]

全コンビネーションを格納する配列：combination_list

関数 generate_combination_list()について

6C3 を例にしてアルゴリズムを説明



組み合わせの樹形図は一個一個の枝に着目すると、一回り小さな樹形図を持っている

一個内側の階層に入ると何が変わるかというと、まず nCr の r が変わる。

外側では **3 個**とる組み合わせを探していたが、内側では **2 個**とる組み合わせを探すことになる。そのさらに内側では **1 個**とる組み合わせを探すことになる。

また、一個内側の階層ではスタート地点の数字も変わる。

一番外側の **1 の枝**の内側の階層では、**2,3,4,5 の枝**があり、

その内側階層の **2 の枝**には、**3,4,5,6 の枝**があり

その内側階層の **3 の枝**には、**4,5,6 の枝**があり

その内側階層の **4 の枝**には、**5,6 の枝**があり

一番外側の **2 の枝**の内側の階層では、**3,4,5 の枝**があり、

………

そこで、スタート地点の数字を i とおくと「その一個内側の i」を j として、

j は i+1 から n-r+1 までの間を動く。

また、一番内側の階層では r=1 となり、そこではもはや組み合わせなど考えずに、1 個ずつ数字をとっていけばよい。例えば 1 の枝の 2 の枝の内側では、3,4,5,6 を順に取るだけである。

そこで、

一番内側の **r=1** の階層まで入って**「1 個ずつ取る」**を実行(48~52 行目)してから

一個外側の階層の**スタート地点の数字 j** をくっつける(57~59 行目)

↑その数字 **j を i+1~n-r+1 まで**動かして実行（56~60 行目）

すると、枝から伸びる組み合わせ総数がわかったので、**さらに外側**の階層の組み合わせの計算に使うことができる

………

（＊最後のページに 6C3 の例を示した）

という処理を実装した。56 行目で自分自身を関数として呼び出す**再帰呼び出し**により、ど

んどん「内側の関数」に入っていくことができ、一番内側(r=1)に入ると、**値が確定して return** できるので、「外側の関数」に出ていくことができる。これで一番外側の関数に出てきたときに、ようやく求める値が確定する。
※**一番外側の 1,2,3,4** の根本部分は、更に外側に **i=0** からスタートする枝が存在し、そこからのびた **j=1,2,3,4 の枝**であると解釈して書かれている。

＊ 6C3 の処理の流れ

引数(n,r,i)=(6,3,0)の状態でスタート

j=1,2,3,4 についてループを回そう

j=1 のとき

(n,r,i)=(6,2,2)

j'=2,3,4,5 についてループを回そう

j'=2 のとき

(n,r,i)=(6,1,3)

r=1 になったので、配列に j''=3,4,5,6 をそれぞれ格納。

[[3],[4],[5],[6]]

そこに j'=一個外側を加えて

[[**2**,3],[**2**,4],[**2**,5],[**2**,6]]

j'=3 のとき

(n,r,i)=(6,1,4)

配列に j''=4,5,6 を格納。[[4],[5],[6]]

そこに j'を加えて[[**3**,4],[**3**,5],[**3**,6]]

j'=4 のとき

同様にして[[**4**,5],[**4**,6]]

j'=5 のとき

同様にして[[**5**,6]]

一個内側の階層の中身がすべてわかったので、そこに j=1 をつける

[[**1**,2,3],[**1**,2,4],[**1**,2,5],[**1**,2,6],[**1**,3,4],[**1**,3,5],[**1**,3,6],[**1**,4,5],[**1**,4,6],[**1**,5,6]

j=2 のとき（略）

j=3 のとき（略）

j=4 のとき（略）

j=1,2,3,4 についてすべてわかったので、それを全部合わせれば求める組み合わせになる