**Program Description**

The program simulates a Job Assignment Problem wherein an **n** number of jobs and candidates are available and each job must be assigned only one candidate. Each candidate may demand a different salary for each job. The program utilizes the Hungarian Method to solve for the assignment with the least total cost such that one candidate is assigned to one job.

The program will ask for user input for the amount of jobs available which would be the **n** value for the **nxn** matrix followed by the input of each candidate’s salary demand an **n2** amount of times (n number of candidates multiplied by an n number of jobs. The program will then proceed to present the algorithm flow step-by-step attempting to find the most optimal assignment with the least total cost.

**Limitations**

Some matrices may require a solution that is beyond the scope of the algorithm such as matrices that have multiple optimal assignments and matrices that do not have at least one row containing only one zero. Other approaches are required for such occasions and are indicated by the program when such scenarios occur.

**References**

The Assignment Problem and the Hungarian Method. (n.d.). Retrieved from http://math.harvard.edu/archive/20\_spring\_05/handouts/assignment\_overheads.pdf

Assignment problem: Hungarian method 3. (n.d.). Retrieved October 13, 2018, from https://www.wiwi.uni-kl.de/bisor-orwiki/Assignment\_problem:\_Hungarian\_method\_3

And the works of Prof. G. Srinivasan, Department of Management Studies, IIT Madras:

N. (2009, August 31). Lec-16 Assignment Problem - Hungarian Algorithm. Retrieved October 13, 2018, from https://www.youtube.com/watch?v=BUGIhEecipE

1. #include <iomanip>
2. #include <iostream>
3. **using** **namespace** std;
4. // User-defined Functions Prototypes
5. **int** matrixzerocount(**bool** (&zeromatrix)[9][9], **int** matrixsize);
6. **void** header(), pauseheader();
7. **void** createzeromatrix(**bool** (&zeromatrix)[9][9], **int** (&matrix)[9][9], **int** matrixsize);
8. **void** rowzerocount(**bool** (&zeromatrix)[9][9], **int** (&zerocount)[2][9], **int** matrixsize);
9. **void** colzerocount(**bool** (&zeromatrix)[9][9], **int** (&zerocount)[2][9], **int** matrixsize);
10. **void** matrixdisplay(**int** (&matrix)[9][9],**bool** (&zeromatrix)[9][9], **int**, **bool** (&tick)[2][9], **int** (&mark)[9][9], **bool**, **bool**, **bool**, **bool**);
11. // Main Method
12. **int** main(){
13. // cmd Console Recoloration
14. system("COLOR f0");
15. // Variables Initialization
16. **int** matrixsize, numvalue, salary[9][9], replicate[9][9], zerocount[2][9], mark[9][9];;
17. **bool** zeromatrix[9][9], tick[2][9];
18. // Program Start
19. **for** (**int** programinstance=1; ; programinstance++){
20. // Data Capture: Employees and Jobs Count
21. header();
22. **do**{
23. **if** (cin.fail()){
24. cin.clear();
25. cin.ignore(100,'\n');
26. }
27. cout << " Enter the number of open jobs: ";
28. cin >> numvalue;
29. }**while**(cin.fail() || numvalue < 2 || numvalue > 9);
30. matrixsize = numvalue;
31. // Data Capture: Salary Values
32. cout << endl;
33. **for** (**int** x=0; x<matrixsize; x++){
34. cout << "   Applicant 0" << x+1 << endl;
35. **for** (**int** y=0; y<matrixsize; y++){
36. **do**{
37. **if** (cin.fail()){
38. cin.clear();
39. cin.ignore(100,'\n');
40. }
41. **if** (y == 0){cout << " Candidate's Expected Salary for ";}
42. **else** {cout << "                                 ";}
43. cout << "Job #" << y+1 << ": ";
44. cin >> numvalue;
45. }**while**(cin.fail() || numvalue < 0 || numvalue >= 1000000);
46. salary[x][y] = numvalue;
47. }
48. cout << endl;
49. }
50. pauseheader();
51. system("pause");
52. system("cls");
53. // Replicate Matrix
54. **for** (**int** x=0; x<9; x++){
55. **for** (**int** y=0; y<9; y++){
56. replicate[x][y] = salary[x][y];
57. }
58. }
59. // Algorithm Simulation Start
60. cout << endl
61. << "     " << **char**(210) << **char**(196) << **char**(196) << **char**(216) << "          "
62. << **char**(216) << **char**(196) << **char**(196) << **char**(196) << **char**(196) << **char**(196) << **char**(183) << endl
63. << "     " << **char**(206) << " Algorithm's Steps " << **char**(206) << endl
64. << "     " << **char**(211) << **char**(196) << **char**(196) << **char**(196) << **char**(196) << **char**(196) << **char**(216)
65. << "          " << **char**(216) << **char**(196) << **char**(196) << **char**(208) << endl;
66. // Hungarian Algorithm Flow: Step 1
67. cout << endl
68. << **char**(196) << **char** (216) << **char**(196) << "\n  STEP 1\n" << **char**(196) << **char** (216) << **char**(196)
69. << "\n  Arrange the costs in a squared matrix.\n\n";
70. matrixdisplay(replicate,zeromatrix,matrixsize,tick,mark,**false**,**false**,**false**,**false**);
71. pauseheader();
72. system("pause");
73. // Hungarian Algorithm Flow: Step 2
74. **for** (**int** x=0; x<matrixsize; x++){
75. // Retrieve Row's Smallest Value
76. numvalue = replicate[x][0];
77. **for** (**int** y=1; y<matrixsize; y++){
78. **if** (replicate[x][y] < numvalue){
79. numvalue = replicate[x][y];
80. }
81. }
82. // Subtract Row's Smallest Value from Each Row Element
83. **for** (**int** y=0; y<matrixsize; y++){
84. replicate[x][y] -= numvalue;
85. }
86. }
87. cout << endl << endl << endl
88. << **char**(196) << **char** (216) << **char**(196)   << "\n  STEP 2\n" << **char**(196) << **char** (216) << **char**(196)
89. << "\n  Reduce all rows by the least value of that row to get at least one 0 in every row.\n\n";
90. matrixdisplay(replicate,zeromatrix,matrixsize,tick,mark,**false**,**false**,**false**,**false**);
91. pauseheader();
92. system("pause");
93. // Hungarian Method: Step 3
94. **for** (**int** x=0; x<matrixsize; x++){
95. // Retrieve Column's Smallest Value
96. numvalue = replicate[0][x];
97. **for** (**int** y=1; y<matrixsize; y++){
98. **if** (replicate[y][x] < numvalue){
99. numvalue = replicate[y][x];
100. }
101. }
102. // Subtract Column's Smallest Value from Each Row Element
103. **for** (**int** y=0; y<matrixsize; y++){
104. replicate[y][x] -= numvalue;
105. }
106. }
107. cout << endl << endl << endl
108. << **char**(196) << **char** (216) << **char**(196) << "\n  STEP 3\n" << **char**(196) << **char** (216) << **char**(196)
109. << "\n Reduce all columns by the least value of that column to get at least one 0 in every column.\n\n";
110. matrixdisplay(replicate,zeromatrix,matrixsize,tick,mark,**false**,**false**,**false**,**false**);
111. pauseheader();
112. system("pause");
113. // Generate Zero Matrix
114. createzeromatrix(zeromatrix,replicate,matrixsize);
115. // Determine Each Row's Zero Count
116. rowzerocount(zeromatrix,zerocount,matrixsize);
117. // Check if Exhaustive Search Would Be Used
118. numvalue = 0;
119. // Check if All Rows Have 2 or More Zeros
120. **for** (**int** x=0; x<matrixsize; x++){
121. **if** (zerocount[0][x] >= 2){
122. numvalue++;
123. }
124. }
125. **if** (numvalue == matrixsize){
126. // Unsolveable Matrix
127. cout << endl << endl << endl
128. << **char**(196) << **char** (216) << **char**(196) << "\n  CAN NOT SOLVE GIVEN MATRIX\n"
129. << **char**(196) << **char** (216) << **char**(196) << endl;
130. matrixdisplay(replicate,zeromatrix,matrixsize,tick,mark,**false**,**false**,**false**,**false**);
131. cout << "\n   Solution to given matrix is beyond the scope of the current program's algorithm.\n"
132. << "   Either multiple assignments combinations are currently possible or an advanced approach must be followed.\n\n";
133. pauseheader();
134. system("pause");
135. }
136. **else**{
137. **do**{
138. // Initialize Row and Column Zero Counts
139. rowzerocount(zeromatrix,zerocount,matrixsize);
140. colzerocount(zeromatrix,zerocount,matrixsize);
141. // Initialize "ticks"
142. **for** (**int** x=0; x<2; x++){
143. **for** (**int** y=0; y<matrixsize; y++){
144. tick[x][y] = **false**;
145. }
146. }
147. // Initialize "marks"
148. **for** (**int** x=0; x<matrixsize; x++){
149. **for** (**int** y=0; y<matrixsize; y++){
150. mark[x][y] = 1;
151. }
152. }
153. **do**{
154. // Re-determine Each Column's Zero Count
155. rowzerocount(zeromatrix,zerocount,matrixsize);
156. // Hungarian Method: Step 4a
157. **for** (**int** x=0; x<matrixsize; x++){
158. **if** (zerocount[0][x] == 1){
159. **for** (**int** y=0; y<matrixsize; y++){
160. **if** (zeromatrix[x][y] == **true**){
161. mark[x][y] = 2;
162. **for** (**int** z=0; z<matrixsize; z++){
163. **if** (zeromatrix[z][y] == **true** && z != x){
164. zeromatrix[z][y] = **false**;
165. mark[z][y] = 3;                                          }
166. }
167. **break**;
168. }
169. }
170. }
171. }
172. cout << endl << endl << endl
173. << **char**(196) << **char** (216) << **char**(196) << "\n  STEP 4a\n" << **char**(196) << **char** (216) << **char**(196)
174. << "\n If a row has exactly one 0 then mark it and cross out all 0 in the corresponding column.\n\n";
175. matrixdisplay(replicate,zeromatrix,matrixsize,tick,mark,**false**,**false**,**true**,**false**);
176. pauseheader();
177. system("pause");
178. // Re-determine Each Column's Zero Count
179. colzerocount(zeromatrix,zerocount,matrixsize);
180. // Hungarian Method: Step 4b
181. **for** (**int** x=0; x<matrixsize; x++){
182. **if** (zerocount[1][x] == 1){
183. **for** (**int** y=0; y<matrixsize; y++){
184. **if** (zeromatrix[y][x] == **true**){
185. mark[y][x] = 2;
186. **for** (**int** z=0; z<matrixsize; z++){
187. **if** (zeromatrix[y][z] == **true** && z != x){
188. **if** (mark[y][z] == 1){
189. mark[y][z] = 3;
190. }
191. zeromatrix[y][z] = **false**;
192. }
193. }
194. **break**;
195. }
196. }
197. }
198. }
199. cout << endl << endl << endl
200. << **char**(196) << **char** (216) << **char**(196) << "\n  STEP 4b\n" << **char**(196) << **char** (216) << **char**(196)
201. << "\n Then do similar procedure for all columns until all zeroes have either been marked or crossed.\n\n";
202. matrixdisplay(replicate,zeromatrix,matrixsize,tick,mark,**false**,**false**,**true**,**false**);
203. // Check if All Zeroes Have Either Been Marked or Crossed
204. numvalue = 0;
205. **for** (**int** x=0; x<matrixsize; x++){
206. **for** (**int** y=0; y<matrixsize; y++){
207. **if** (mark[x][y] == 1 && replicate[x][y] == 0){
208. numvalue++;
209. }
210. }
211. }
212. **if** (numvalue != 0){
213. cout << "      " << numvalue << " zero";
214. **if** (numvalue == 1){cout <<"es";}
215. cout << " remain unmarked or uncrossed.\n\n";
216. }
217. **else**{
218. cout << "      No zeroes remain unmarked or uncrossed.\n\n";
219. }
220. pauseheader();
221. system("pause");
222. }**while**(numvalue != 0);
223. // Hungarian Method: Step 4c
224. cout << endl << endl << endl
225. << **char**(196) << **char** (216) << **char**(196) << "\n  STEP 4c\n" << **char**(196) << **char** (216) << **char**(196);
226. // Optimal Assignment is Possible
227. **if** (matrixzerocount(zeromatrix,matrixsize) == matrixsize){
228. numvalue == 1;
229. **break**;
230. }
231. // Optimal Assignment is not Possible
232. cout << "\n The number of assigned zeros (n=" << matrixzerocount(zeromatrix,matrixsize)
233. << ") is not enough to complete an optimal assignment.\n\n";
234. pauseheader();
235. system("pause");
236. // Hungarian Method: Step 5
237. **for** (**int** x=0; x<2; x++){
238. **for** (**int** y=0; y<matrixsize; y++){
239. tick[x][y] = **false**;
240. }
241. }
242. // Re-determine Matrix's Row and Column Zero Count to Reflect Changes Made in Step 4b
243. rowzerocount(zeromatrix,zerocount,matrixsize);
244. colzerocount(zeromatrix,zerocount,matrixsize);
245. // 'Tick' Unassigned Rows
246. numvalue = 0;
247. **for** (**int** x=0; x<matrixsize; x++){
248. **if** (zerocount[0][x] == 0){
249. numvalue++;
250. tick[0][x] = **true**;
251. }
252. }
253. // Check if Any Rows Were Ticked
254. **if** (numvalue == 0){
255. // All Possible Ticking is Done
256. **break**;
257. }
258. cout << endl << endl << endl
259. << **char**(196) << **char** (216) << **char**(196) << "\n  STEP 5a\n" << **char**(196) << **char** (216) << **char**(196)
260. << "\n Tick all the unassigned rows.\n\n";
261. matrixdisplay(replicate,zeromatrix,matrixsize,tick,mark,**true**,**false**,**true**,**false**);
262. pauseheader();
263. system("pause");
264. // Hungarian Method: Step 5b
265. **for** (**int** x=0; x<matrixsize; x++){
266. **if** (tick[0][x] == **true**){
267. **for** (**int** y=0; y<matrixsize; y++){
268. **if** (replicate[x][y] == 0){
269. tick[1][y] = **true**;
270. }
271. }
272. }
273. }
274. cout << endl << endl << endl
275. << **char**(196) << **char** (216) << **char**(196) << "\n  STEP 5b\n" << **char**(196) << **char** (216) << **char**(196)
276. << "\n For every ticked row, if there is a zero then cross the corresponding column of that zero.\n\n";
277. matrixdisplay(replicate,zeromatrix,matrixsize,tick,mark,**true**,**false**,**true**,**false**);
278. pauseheader();
279. system("pause");
280. // Hungarian Method: Step 6
281. **for** (**int** x=0; x<matrixsize; x++){
282. **if** (tick[1][x] == **true**){
283. **for** (**int** y=0; y<matrixsize; y++){
284. **if** (zeromatrix[y][x] == **true**){
285. tick[0][y] = **true**;
286. }
287. }
288. }
289. }
290. cout << endl << endl << endl
291. << **char**(196) << **char** (216) << **char**(196) << "\n  STEP 6\n" << **char**(196) << **char** (216) << **char**(196)
292. << "\n For every crossed column if there is a zero such that its row is unticked then tick it.\n\n";
293. matrixdisplay(replicate,zeromatrix,matrixsize,tick,mark,**true**,**false**,**true**,**false**);
294. pauseheader();
295. system("pause");
296. // Hungarian Method: Step 7
297. cout << endl << endl << endl
298. << **char**(196) << **char** (216) << **char**(196) << "\n  STEP 7\n" << **char**(196) << **char** (216) << **char**(196)
299. << "\n Draw lines through unticked rows and ticked columns.\n\n";
300. matrixdisplay(replicate,zeromatrix,matrixsize,tick,mark,**true**,**false**,**true**,**false**);
301. **for** (**int** x=0 ;x<matrixsize; x++){
302. **if** (tick[0][x] == **true**){
303. tick[0][x] = **false**;
304. }
305. **else**{
306. tick[0][x] = **true**;
307. }
308. }
309. matrixdisplay(replicate,zeromatrix,matrixsize,tick,mark,**false**,**true**,**true**,**false**);
310. cout << "     These are the minimal lines required to pass every zero in the Matrix.\n\n";
311. pauseheader();
312. system("pause");
313. // Hungarian Method: Step 8
314. numvalue = 0;
315. // Retrieve Highest Value from Entire "replicate" Matrix
316. **for** (**int** x=0; x<matrixsize; x++){
317. **for** (**int** y=0; y<matrixsize; y++){
318. **if** (replicate[x][y] > numvalue){
319. numvalue = replicate[x][y];
320. }
321. }
322. }
323. // Retrieve Smallest Value from "Untraced" Cells
324. **for** (**int** x=0; x<matrixsize; x++){
325. **if** (tick[0][x] == **false**){
326. **for** (**int** y=0; y<matrixsize; y++){
327. **if** (tick[1][y] == **false**){
328. **if** (replicate[x][y] < numvalue){
329. numvalue = replicate[x][y];
330. }
331. }
332. }
333. }
334. }
335. cout << endl << endl << endl
336. << **char**(196) << **char** (216) << **char**(196) << "\n  STEP 8\n" << **char**(196) << **char** (216) << **char**(196)
337. << "\n Find the smallest value of the matrix such that neither it's row nor column are marked. Call it theta.\n\n";
338. matrixdisplay(replicate,zeromatrix,matrixsize,tick,mark,**false**,**true**,**false**,**false**);
339. cout << "\n     The theta is: " << numvalue << ".\n\n";
340. pauseheader();
341. system("pause");
342. // Hungarian Method: Step 9
343. cout << endl << endl << endl
344. << **char**(196) << **char** (216) << **char**(196) << "\n  STEP 9\n" << **char**(196) << **char** (216) << **char**(196)
345. << "\n For all entries of the matrix, if there is no line then subtract theta.\n If there are 2 lines passing through then add theta.\n\n";
346. matrixdisplay(replicate,zeromatrix,matrixsize,tick,mark,**false**,**true**,**false**,**false**);
347. **for** (**int** x=0; x<matrixsize; x++){
348. **for** (**int** y=0; y<matrixsize; y++){
349. **if** (tick[0][x] == **true** || tick[1][y] == **true**){
350. **if** (tick[0][x] == **true** && tick[1][y] == **true**){
351. replicate[x][y] += numvalue;
352. }
353. }
354. **else**{
355. replicate[x][y] -= numvalue;
356. }
357. }
358. }
359. cout << "\n               Changes into:\n";
360. matrixdisplay(replicate,zeromatrix,matrixsize,tick,mark,**false**,**true**,**false**,**false**);
361. pauseheader();
362. system("pause");
363. // Hungarian Method: Step 10
364. cout << endl << endl << endl
365. << **char**(196) << **char** (216) << **char**(196) << "\n  STEP 10\n" << **char**(196) << **char** (216) << **char**(196)
366. << "\n Repeat the algorithm starting from Step 4\n\n";
367. pauseheader();
368. system("pause");
369. // Reset Zero Matrix
370. createzeromatrix(zeromatrix,replicate,matrixsize);
371. }**while**(**true**);
372. // Alternate Part of Step 4c
373. cout << "\n The number of assigned zeros (n=" << matrixzerocount(zeromatrix,matrixsize)
374. << ") is enough to complete an optimal assignment.\n\n";
375. matrixdisplay(replicate,zeromatrix,matrixsize,tick,mark,**false**,**false**,**true**,**false**);
376. pauseheader();
377. system("pause");
378. // Hungarian Method: Step 11
379. cout << endl << endl << endl
380. << **char**(196) << **char** (216) << **char**(196) << "\n  STEP 11\n" << **char**(196) << **char** (216) << **char**(196)
381. << "\n Report all the marked zeroes. Assigning these cells will give us the least cost.\n\n"
382. << " Salary Matrix:\n";
383. matrixdisplay(salary,zeromatrix,matrixsize,tick,mark,**false**,**false**,**false**,**true**);
384. cout << "\n Job Assignments with Least Total Salary Cost:\n";
385. numvalue = 0;
386. **for** (**int** x=0; x<matrixsize; x++){
387. **for** (**int** y=0; y<matrixsize; y++){
388. **if** (zeromatrix[x][y] == **true**){
389. cout << "    Employee #0" << x+1 << " will be assigned Job " << y+1
390. << " with an expected salary of Php " << salary[x][y] << endl;
391. numvalue += salary[x][y];
392. **break**;
393. }
394. }
395. }
396. cout << "\n Least Total Cost: " << numvalue << endl << endl;
397. pauseheader();
398. system("pause");
399. }
400. // Confirm Program Loop Reiteration
401. **do**{
402. **if** (cin.fail()){
403. cin.clear();
404. cin.ignore(100,'\n');
405. }
406. system("cls");
407. header();
408. cout << "\t\t\t[1]:Yes [2]:Terminate\n\n Try another matrix?: ";
409. cin >> numvalue;
410. }**while**(cin.fail() || (numvalue != 1 && numvalue != 2));
411. // Terminates Loop by User Input
412. **if** (numvalue == 2){
413. **break**;
414. }
415. system("cls");
416. }
417. **return** 0;
418. }
419. // User-defined Functions Definitions
420. **void** header(){
421. cout << "\n      " << **char**(201) << "    ";
422. **for** (**int** x=0; x<8; x++){
423. cout << **char**(196);
424. }
425. cout << "    " << **char**(187)
426. << "\n       Hungarian Method"
427. << "\n      " << **char**(200) << "    ";
428. **for** (**int** x=0; x<8; x++){
429. cout << **char**(196);
430. }
431. cout << "    " << **char**(188) << endl << endl;
432. }
433. **void** pauseheader(){
434. cout << **char**(199);
435. **for** (**int** x=0; x<31 ;x++){
436. cout << **char**(196);
437. }
438. }
439. // Matrices Display Format
440. **void** matrixdisplay(**int** (&matrix)[9][9], **bool** (&zeromatrix)[9][9], **int** matrixsize, **bool** (&tick)[2][9], **int** (&mark)[9][9], **bool** ticked, **bool** lined, **bool** marked, **bool** final){
441. cout << "      Job\n  #  ";
442. **for** (**int** x=0; x<matrixsize; x++){
443. cout << "   " << x+1 << "    ";
444. }
445. cout << "\n    " << **char**(201);
446. **for** (**int** x=0; x<matrixsize; x++){
447. cout << **char**(205) << **char**(205) << **char**(205);
448. **if** (lined == **true** && tick[1][x] == **true**){cout << **char**(216);}
449. **else**{cout << **char**(205);}
450. cout << **char**(205) << **char**(205) << **char**(205);
451. **if** (x+1 == matrixsize){cout << **char**(184);}
452. **else** {cout << **char**(209);}
453. }
454. cout << endl;
455. **for** (**int** x=0; x<matrixsize; x++){
456. cout << " 0" << x+1 << " ";
457. **if** (lined == **true** && tick[0][x] == **true**){cout << **char**(215);}
458. **else**{cout << **char**(186);}
459. **for** (**int** y=0; y<matrixsize; y++){
460. **if** (lined == **true** && tick[0][x] == **true**){
461. cout << right << setfill(**char**(196));
462. **if** (marked == **true** && matrix[x][y] == 0 && mark[x][y] != 1){
463. **if** (mark[x][y] == 2){
464. cout << setw(5) << **char**(251) << " 0";
465. }
466. **else** **if** (mark[x][y] == 3){
467. cout << setw(7) << "X";
468. }
469. }
470. **else**{
471. cout << setw(7) << matrix[x][y];
472. }
473. **if** (lined == **true** && tick[0][x] == **true**){cout << **char**(197);}
474. **else**{cout << **char**(179);}
475. }
476. **else**{
477. cout << right;
478. **if** (marked == **true** && matrix[x][y] == 0 && mark[x][y] != 1){
479. **if** (mark[x][y] == 2){
480. cout << setw(5) << **char**(251) << " 0";
481. }
482. **else** **if** (mark[x][y] == 3){
483. cout << setw(7) << "X";
484. }
485. }
486. **else**{
487. **if** (final == **true** && mark[x][y] == 2){
488. cout << **char**(251 )<< setw(6) << matrix[x][y];
489. }
490. **else**{
491. cout << setw(7) << matrix[x][y];
492. }
493. }
494. cout << **char**(179);
495. }
496. }
497. **if** (ticked == **true**){
498. **if** (tick[0][x] == **true**){
499. cout << "<";
500. }
501. }
502. cout << "\n    ";
503. **if** (x+1 == matrixsize){cout << **char**(211);}
504. **else** {cout << **char**(199);}
505. **for** (**int** y=0; y<matrixsize; y++){
506. cout << **char**(196) << **char**(196) << **char**(196);
507. **if** (lined == **true** && tick[1][y] == **true**){cout << **char**(197);}
508. **else**{cout << **char**(196);}
509. cout << **char**(196) << **char**(196) << **char**(196);
510. **if** (y+1 != matrixsize){
511. **if** (x+1 == matrixsize){cout << **char**(193);}
512. **else** {cout << **char**(197);}
513. }
514. }
515. **if** (x+1 == matrixsize){cout << **char**(217);}
516. **else** {cout << **char**(180);}
517. cout << setfill(' ');
518. cout << endl;
519. }
520. **if** (ticked == **true**){
521. cout << "    ";
522. **for** (**int** y=0; y<matrixsize; y++){
523. **if** (tick[1][y] == **true**){
524. cout << "    ^   ";
525. }
526. **else**{
527. cout << "       ";
528. }
529. }
530. cout << endl;
531. }
532. }
533. // Populates Zero Matrix
534. **void** createzeromatrix(**bool** (&zeromatrix)[9][9], **int** (&matrix)[9][9], **int** matrixsize){
535. **for** (**int** x=0; x<matrixsize; x++){
536. **for** (**int** y=0; y<matrixsize; y++){
537. **if** (matrix[x][y] == 0){
538. zeromatrix[x][y] = **true**;
539. }
540. **else**{
541. zeromatrix[x][y] = **false**;
542. }
543. }
544. }
545. }
546. // Calculates the Amount of Zeros Per Row
547. **void** rowzerocount(**bool** (&zeromatrix)[9][9], **int** (&zerocount)[2][9], **int** matrixsize){
548. **int** rowcount;
549. **for** (**int** x=0; x<matrixsize; x++){
550. rowcount = 0;
551. **for** (**int** y=0; y<matrixsize; y++){
552. **if** (zeromatrix[x][y] == **true**){
553. rowcount++;
554. }
555. }
556. zerocount[0][x] = rowcount;
557. }
558. }
559. // Calculates the Amount of Zeros Per Column
560. **void** colzerocount(**bool** (&zeromatrix)[9][9], **int** (&zerocount)[2][9], **int** matrixsize){
561. **int** colcount;
562. **for** (**int** x=0; x<matrixsize; x++){
563. colcount = 0;
564. **for** (**int** y=0; y<matrixsize; y++){
565. **if** (zeromatrix[y][x] == **true**){
566. colcount++;
567. }
568. }
569. zerocount[1][x] = colcount;
570. }
571. }
572. **int** matrixzerocount(**bool** (&zeromatrix)[9][9], **int** matrixsize){
573. **int** totalzeros = 0;
574. **for** (**int** x=0; x<matrixsize; x++){
575. **for** (**int** y=0; y<matrixsize; y++){
576. **if** (zeromatrix[x][y] == **true**){
577. totalzeros++;
578. }
579. }
580. }
581. **return** totalzeros;
582. }
583. /\* Reference Algorithm Flow Retrieved from "http://home.iitk.ac.in/~mpatel/cs300/5B/cs300\_a4.pdf"
584. 1.  Arrange the cost as described in the representation section.
585. 2.  Reduce all rows by the least value of that row to get atleast one 0 in every row.
586. 3.  Reduce all columns by the least value of that column to get atleast one 0 in every column.
587. 4.  If a row has exactly one 0 then mark it and cross out all 0 in the corresponding column.
588. Do this for all rows. Then do similar procedure for all column till all zeroes have either been marked or crossed.
589. If the number of marked zeroes equal n then skip to step 11.
590. 5.  Tick all the unassigned rows.  For every ticked row if there is a zero then cross the corresponding column of that zero.
591. 6.  For every crossed column if there is a zero such that it’s row is unticked then tick it.
592. Repeat steps 5 and 6 till no more ticking is possible.
593. 7.  Cross out unticked rows.
594. 8.  Find the smalled value of the matrix such that neither it’s row nor column are marked.  Call it theta.
595. 9.  For all entries of the matrix , if there is no line the subtract theta .  If there is one line passing through then do nothing.
596. If there are 2 lines passing through then add theta.
597. 10. Mark zeroes similar to how they were done in step 4 i.e if there is a single zero then mark it and cross all other zero in
598. the corresponding column. Do this for all rows and columns. If number of marked zeroes is n the continue else go back to step 5.
599. 11. Report all the marked zeroes. Assigning these cells will give us the least cost.
600. \*/