The problem **SCITIES** is a classic problem with no twist. Still, it is a relatively hard problem: weighted bipartite matching. I am learning now the Hungarian algorithm from Knuth's SGB so I'll solve this problem to make sure I understand and remember the algorithm.

The input format. The first line contains the number of tests which is at most 1000. Each test starts with two numbers M and N which are the dimensions of the matrix. We know that $1 \leq M, N \leq 100$. The following lines contain information about edges: a row index, a column index and the value of that matrix element. The indices are one based and the quantity of goods is non-negative and at most 100. The edge description ends with the dummy line "0 0 0".

The output format. For each test, one line should be written saying what is the *maximum* weight match.

The overall structure of the program.

⟨*⟩≡
```
#include <algorithm>
#include <iostream>
#include <cstdlib>
using namespace std;
⟨Macros⟩
⟨Constants⟩
⟨Globals⟩

int main() {
  int tests; cin >> tests;
  for (int i = 0; i < tests; ++i) {
    ⟨Read input⟩
    ⟨Solve the assignment problem⟩
    ⟨Write result⟩
    ⟨Report work⟩
  }
}
```

For this program I will use *mems* to estimate efficiency of the algorithm. The programmer annotates all references to memory. This is an imperfect measure but it is practical (as opposed to theoretical like the big-O analysis) and tends to be machine independent.

⟨Macros⟩≡
```
#define o   ++mems
#define oo  mems += 2
#define ooo mems += 3
```

⟨*Globals*⟩≡
```
int mems;  // Memory references
```

Since I do not feel confortable starting with the algorithm I will get rid first of some boring stuff: parsing the input. The Hungarian algorithm works on matrices not on edge lists. It also prefers to have $M \leq N$ so I will transpose the input if it says otherwise. And I will change from maximum to minimum.

⟨*Constants*⟩≡
```
const int MAX = 100;
```

⟨*Globals*⟩+≡
```
int M;            // Number of rows.
int N;            // Number of columns.
int w[MAX][MAX]; // The weights (MxN adjacency matrix)
int r;            // Row of interest.
int c;            // Column of interest.
```

⟨*Read input*⟩≡
```
cin >> M >> N;
bool transpose = false;
if (M > N) {
  swap(M, N);
  transpose = true;
}
for (r = 0; r < M; ++r) for (c = 0; c < N; ++c)
  o, w[r][c] = MAX;
while (true) {
  int weight;
  cin >> r >> c >> weight;
  if (!r) break;
  if (transpose) swap(r, c);
  o, w[r-1][c-1] = MAX - weight;
}
```

For implementing the algorithm I will use a set of arrays as Knuth does (saying he uses a suggestion of Papadimitriou). I will skip the overall description of the algorithm because it is well done in SGB. I do assume that the reader has also read that description or otherwise knows well the Hungarian algorithm. I will describe here the data structures. I feel that I need to have such a description near the code in order to implement it correctly. Also, understanding the code would be difficult without this.

The matrix of weigths can be seen as an adjacency matrix whose *zeros* define a graph structure. So it makes sense to talk about paths and other graph concepts. Notice that the rows and the columns of the matrix identify two *distinct* sets of nodes. In other words the matrix representation of a bipartite graph is different (more compact) than the generic adjacency matrix representation.

The matchings are held in two arrays. If column `c` is matched with row `r` then `col_mate[r] == c` and `row_mate[c] == r`. In this case we have `w[r][c]==0`. Otherwise the values in the arrays are `-1`.

We need to keep track of paths that start with an unchosen row and contain only matched columns. We need to travel these paths only backwards. The row `r` is preceded by column `col_mate[r]`. And column `c` is preceded by row `parent_row[c]`. To keep track of which rows are part of such an interesting path we use the first `t` locations of the array `unchosen_rows`.

⟨*Globals*⟩+≡
```
int t;                  // Number of unchosen rows.
int unchosen_rows[MAX];
int col_mate[MAX];
int row_mate[MAX];
int parent_row[MAX];
```

For the *unweighted* problem this is all we would need. However that is only a subproblem of what we are trying to solve. We also need to keep track of row decrements and column increments in arrays `row_dec` and `col_inc`. This is because the solution of the weighted problem is the same as the solution of the unweighted problem on a matrix with values `w[r][c] - row_dec[r] + col_inc[c]` if we choose the decrements and increments properly. For the choice of increments and decrements we need to identify the minimum value in areas of the matrix `w` that are not covered by *chosen* rows or columns. To do that efficiently we keep track of the minimum in each column, ignoring chosen rows, in the array `slack`.

⟨*Globals*⟩+≡
```
int row_dec[MAX];
int col_inc[MAX];
int slack[MAX];
```

⟨*Macros*⟩+≡
```
  #define W(r, c) (ooo, w[r][c] - row_dec[r] + col_inc[c])
```

Algorithm overview. We stop when all rows are matched. We begin by exploring (BFS) interesting paths and trying to find a new existing independent zero to add to the match. If we find such a zero we add it; otherwise we introduce a new zero by adjusting the increments. And we repeat.

⟨*Solve the assignment problem*⟩≡
```
  ⟨Initialize matchings⟩
  while (unmatched_rows) {
    int q;        // Number of unchosen rows that were explored.
    ⟨Prepare for this step⟩
    ⟨Search for an improving path⟩
    if (q == t) { // No improving path found.
      ⟨Adjust increments and decrements⟩
    } else {
      ⟨Improve matching⟩
    }
    ⟨Report progress⟩
  }
```

⟨*Globals*⟩+≡
```
  int unmatched_rows; // The number of rows yet to be matched.
```

It is probably good to begin with the easiest part of the algorithm: improve the matching. This will consolidate our understanding of what data we need to collect.

⟨*Improve matching*⟩≡
```
  o, r = parent_row[c];
  while (o, col_mate[r] != -1) {
    int nc;
    o, nc = col_mate[r];
    ⟨Match r with c⟩
    c = nc;
    o, r = parent_row[c];
  }
  ⟨Match r with c⟩
  --unmatched_rows;
```

⟨*Match r with c*⟩≡
```
  o, row_mate[c] = r;
  o, col_mate[r] = c;
```

Now we will look at how decrements and increments are adjusted. For improving the matching we need to set up the content of `parent_row` so that the code above makes sense. Similarly, for adjusting the increments we need to have correct information in `slack`.

⟨*Adjust increments and decrements*⟩≡
```
int del = INF;
for (c = 0; c < N; ++c) if (o, parent_row[c] == -1)
  if (o, slack[c] < del) o, del = slack[c];
for (c = 0; c < N; ++c) if (o, parent_row[c] != -1)
  o, col_inc[c] += del;
for (q = 0; q < t; ++q) oo, row_dec[unchosen_rows[q]] += del;
```

The above assumes that `slack[c]` contains the minimum (adjusted) value in column `c` that is part of an unchosen row. So while we are searching for an improving path we should maintain the meaning of `slack` and of `parent_row`. Establishing the invariants is easy.

⟨*Prepare for this step*⟩≡
```
MEMSET(parent_row, -1);
for (c = 0; c < N; ++c) o, slack[c] = INF;
t = 0;
for (r = 0; r < M; ++r) if (o, col_mate[r] == -1)
  o, unchosen_rows[t++] = r;
```

And now the most interesting part: solving (unweighted) bipartite matching. Note that `slack[c] == 0` only for chosen columns.

⟨*Search for an improving path*⟩≡
```
for (q = 0; q < t; ++q) {
  o, r = unchosen_rows[q];
  for (c = 0; c < N; ++c) if (o, W(r, c) < slack[c]) {
    o, slack[c] = W(r, c);
    if (o, slack[c] == 0) {
      o, parent_row[c] = r;
      if (o, row_mate[c] == -1) goto done_bfs;
      oo, unchosen_rows[t++] = row_mate[c];
    }
  }
}
 done_bfs:
```

When we begin nothing is matched.

⟨*Initialize matchings*⟩≡
```
  MEMSET(col_inc, 0);
  MEMSET(row_dec, 0);
  MEMSET(col_mate, -1);
  MEMSET(row_mate, -1);
  unmatched_rows = M;
```

Now we have the answer. All we need to do is write it in the required format. For that remember that we have switched from maximization to minimization when we read the input.

⟨*Write result*⟩≡
```
  int result  = 0;
  for (r = 0; r < M; ++r)
    oo, result += MAX - w[r][col_mate[r]];
  cout << result << endl;
```

And finaly we can dump, for diagnostic purposes, the number of mems.

⟨*Report work*⟩≡
```
  if (verbose)
    fprintf(stderr, "I used %d mems to solve test %d.\n", mems, i+1);
  mems = 0;
```

Mostly for debug.

⟨*Report progress*⟩≡
```
  if (verbose)
    fprintf(stderr,
            "After %d mems there are %d matched rows\n",
            mems, M - unmatched_rows
            );
```

Some loose ends that I might move later.

⟨*Constants*⟩+≡
```
  const int INF = 0x7fffffff; // Almost infinity
```

⟨*Macros*⟩+≡
```
  #define MEMSET(var, val) \
    mems += sizeof(var) / sizeof(int), memset(var, val, sizeof(var))
```

⟨*Globals*⟩+≡
```
  bool verbose = false;
```