

CHARM

HP muốn tạo ra một chiếc vòng đeo tay từ N hạt trang sức. Mỗi hạt trang sức được cho bởi 2 yếu tố: trọng lượng W_i và độ quyến rũ D_i . HP muốn tạo ra một chiếc vòng có độ quyến rũ lớn nhất và trọng lượng không vượt quá M .

INPUT: CHARM.INP

- Dòng 1: N, M ($1 \leq N \leq 3.402; 1 \leq M \leq 12.880$)
- N dòng tiếp theo, dòng thứ i ghi 2 số là W_i và D_i ($1 \leq W_i \leq 400; 1 \leq D_i \leq 100$)

OUTPUT: CHARM.OUT

- Một số duy nhất là độ quyến rũ tối đa thu được của chiếc vòng

Ví dụ:

CHARM.INP	CHARM.OUT
4 6 1 4 2 6 3 12 2 7	23

USACO DEC07 Problem 'charm' Analysis

by Rob Kolstad

This is one of the classic "knapsack" problems: 'Items' with a 'value' and 'size' can be stuffed an integer number of times into a container of some limited size with the goal of maximizing the value of the items inserted.

The classic solution pivots on a list of the best possible value that can be obtained for the set of items so far seen. The array's subscript is 'size'; it's value is "the best value" that can be obtained for the items so far processed.

Initially, of course, the array's values are all zero. Adding the first element is easy: `bestvalue[itemweight] = itemvalue`.

Subsequent items are processed one-by-one as follows. The item's weight and value are (experimentally) added to each of the *existing* weight/value pairs in the `bestvalue` array (including the 0/0 element). If the new weight/value pair is better, it replaces the old weight value.

By way of example, consider the four items from the problem description:

```
1 4
2 6
3 12
2 7
```

The maximum allowed weight is 6, so 6+1 elements are needed in the 'bestvalue' array:

```
0 0 0 0 0 0 0 <-- best value
0 1 2 3 4 5 6 <-- for this weight (subscript)
```

The first element (weight 1, value 4) is easy to insert:

```
0 4 0 0 0 0 0 <-- best value
0 1 2 3 4 5 6 <-- for this weight (subscript)
```

The second element (weight 2, value 6) requires two additions. For purposes of explanation, we will first add it to the 0/0 element:

```
V-----V
0 4 *6* 0 0 0 0 <-- best value
0 1 2 3 4 5 6 <-- for this weight (subscript)
```

and then to the weight=1 element, which currently has value 4:

```
V-----V
0 4 6 *10* 0 0 0 <-- best value
0 1 2 3 4 5 6 <-- for this weight (subscript)
```

Important point: It is now clear that we can run into trouble if, for instance, our weight was 1. In that case, we'd be modifying sequential elements -- elements whose new value should be based on the original 'best' matrix value and not the newly increased values. Thus, **we must either make a new (properly copied)**

best matrix so modifications are made properly OR we must modify the 'best' matrix from the larger elements to smaller

The third element (weight 3, value 12) requires **four** additions (to weights 0, 1, 2, and 3). We will start at weight 3 and proceed 'backwards':

```

      V-----V
0   4   6   10   0   0 *22* <-- best value
0   1   2   3   4   5   6 <-- for this weight (subscript)

```

```

      V-----V
0   4   6   10   0 *12* 22 <-- best value
0   1   2   3   4   5   6 <-- for this weight (subscript)

```

```

      V-----V
0   4   6   10 *16* 12 22 <-- best value
0   1   2   3   4   5   6 <-- for this weight (subscript)

```

In the final case, we will **overwrite** a now inferior 'best value' element:

```

      V-----V
0   4   6 *12* 16 12 22 <-- best value
0   1   2   3   4   5   6 <-- for this weight (subscript)

```

The fourth and final element (weight 2, value 7) will require potential augmentation of all 7 of the best values since all slots are now occupied. Working backwards:

```

      V-----V
0   4   6  12  16  12 22 <-- best value
0   1   2   3   4   5   6 <-- for this weight (subscript)

```

No modifications since the aggregate weight is too large. Likewise:

```

      V-----V
0   4   6  12  16  12 22 <-- best value
0   1   2   3   4   5   6 <-- for this weight (subscript)

```

Regular calculations and potential modifications then commence. The 6th item is overwritten since $16+7 > 22$ (and so on):

```

      V-----V
0   4   6  12  16  12 *23* <-- best value
0   1   2   3   4   5   6 <-- for this weight (subscript)

```

```

      V-----V
0   4   6  12  16 *19* 23 <-- best value
0   1   2   3   4   5   6 <-- for this weight (subscript)

```

For weight $2 + 2$, the new value $6+7=13$ is less than the current best value of 16, so no modification occurs:

```

      V-----V
0   4   6  12  16  19 23 <-- best value
0   1   2   3   4   5   6 <-- for this weight (subscript)

```

Likewise for weight $1 + 2$ no modification occurs since $4+7=11 < 12$:

```

      V-----V
0   4   6  12  16  19 23 <-- best value
0   1   2   3   4   5   6 <-- for this weight (subscript)

```

The final weight $0+2$ has a new best value:

```

      V-----V
0   4   *7* 12  16  19 23 <-- best value

```

```
0  1  2  3  4  5  6  <-- for this weight (subscript)
```

Having now processed all the input, the 'best value' array is complete. In this case element `bestvalue[6]=23` is the largest of all the values. One can well imagine this would not be the case if, e.g., an item with weight 5 and value 1,000 existed. That might make the largest value appear in some slot other than the final one (i.e., if the other weights were all > 1). Thus we scan the best value array to find the largest element, which is the highest possible value that can be created within the size/weight constraints.

I coded this task using two arrays and inefficient copy operators:

```
#include <stdio.h>

#define MAX 12880
int best[MAX+1], newbest[MAX+1], nb;

main () {
    FILE *fin = fopen ("charm.in", "r");
    FILE *fout = fopen ("charm.out", "w");
    int n, m, w, d, i, j, biggest;

    fscanf (fin, "%d %d", &n, &m);
    bzero (best, (m+1)*sizeof(int));
    bzero (newbest, (m+1)*sizeof(int));
    for (i = 0; i < n; i++) {
        fscanf (fin, "%d %d", &w, &d);
        for (j = 0; j <= m; j++) {
            if (j != 0 && best[j] == 0 || j+w > m) continue;
            nb = best[j] + d;
            if (nb > newbest[j+w]) newbest[j+w] = nb;
        }
        bcopy (newbest, best, (m+1)*sizeof(int));
        /*for (j = 0; j <= m; j++) printf("%2d ", best[j]);
        printf (" -- %d %d\n", w, d);  /* debug */
    }
    biggest = best[0];
    for (j = 1; j <= m; j++)
        if (best[j] > biggest) biggest = best[j];
    fprintf (fout, "%d\n", biggest);
    exit (0);
}
```

Richard Peng, who has coded tasks like this many times, has evolved a more sophisticated solution that requires but one array (processed in reverse order) and uses the cool `'>?='` operator for replacing 'improved' elements. Note that his loops are generally 'correct by construction' -- no need to check for out-of-bounds indices since the loops are properly constrained. Of course his solution is also just about as fast as we know how to solve a task like this:

```
#include <cstdio>
#include <cstring>

int bes[12880], n, m, a, b, i, j;

int main () {
    freopen ("charm.in", "r", stdin);
```

```
freopen ("charm.out", "w", stdout);
scanf ("%d%d", &n, &m);
memset (bes, 0, sizeof (bes));
for (i=0; i<n; i++){
    scanf ("%d%d", &a, &b);
    for (j=m; j>=a; j--){
        bes[j]>?=bes[j-a]+b;
    }
}
printf ("%d\n", bes[m]);
return 0;
}
```