**Introduction**

In an algorithm design there is no one 'silver bullet' that is a cure for all computation problems. Different problems require the use of different kinds of techniques. A good programmer uses all these techniques based on the type of problem. Some commonly-used techniques are:

1. Divide and conquer
2. Randomized algorithms
3. Greedy algorithms (This is not an algorithm, it is a **technique**.)
4. Dynamic programming

**What is a 'Greedy algorithm'?**

A greedy algorithm, as the name suggests, **always makes the choice that seems to be the best at that moment**. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

How do you decide which choice is optimal?

Assume that you have an **objective function** that needs to be optimized (either maximized or minimized) at a given point. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The Greedy algorithm has only one shot to compute the optimal solution so that **it never goes back and reverses the decision**.

Greedy algorithms have some advantages and disadvantages:

1. It is quite easy to **come up with a greedy algorithm** (or even multiple greedy algorithms) for a problem.
2. **Analyzing the run time for greedy algorithms will generally be much easier** than for other techniques (like Divide and conquer). For the Divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of gets smaller and the number of sub-problems increases.
3. The difficult part is that for greedy algorithms **you have to work much harder to understand correctness issues**. Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science. It involves a lot of creativity.

**Note:** Most greedy algorithms are **not** correct. An example is described later in this article.

**C. How to create a Greedy Algorithm?**

Being a very busy person, you have exactly T time to do some interesting things and you want to do maximum such things.

You are given an array **A** of integers, where each element indicates the time a thing takes for completion. You want to calculate the maximum number of things that you can do in the limited time that you have.

This is a simple Greedy-algorithm problem. In each iteration, you have to greedily select the things which will take the minimum amount of time to complete while maintaining two variables **currentTime** and **numberOfThings**. To complete the calculation, you must:

1. Sort the array **A** in a non-decreasing order.
2. Select each to-do item one-by-one.
3. Add the time that it will take to complete that to-do item into **currentTime**.
4. Add one to **numberOfThings**.

Repeat this as long as the **currentTime** is less than or equal to **T**.

Let **A = {5, 3, 4, 2, 1}** and **T = 6**

After sorting, **A = {1, 2, 3, 4, 5}**

After the 1st iteration:

* **currentTime** = 1
* **numberOfThings** = 1

After the 2nd iteration:

* **currentTime** is 1 + 2 = 3
* **numberOfThings** = 2

After the 3rd iteration:

* **currentTime** is 3 + 3 = 6
* **numberOfThings** = 3

After the 4th iteration, **currentTime** is 6 + 4 = 10, which is greater than T. Therefore, the answer is 3.

**Implementation**

#include <iostream>

#include <algorithm>

using namespace std;

const int MAX = 105;

int A[MAX];

int main()

{

int T, N, numberOfThings = 0, currentTime = 0;

cin >> N >> T;

for(int i = 0;i < N;++i)

cin >> A[i];

sort(A, A + N);

for(int i = 0;i < N;++i)

{

currentTime += A[i];

if(currentTime > T)

break;

numberOfThings++;

}

cout << numberOfThings << endl;

return 0;

}

**Where to use Greedy algorithms?**

A problem must comprise these two components for a greedy algorithm to work:

1. It has **optimal substructures**. The optimal solution for the problem contains optimal solutions to the sub-problems.
2. It has a **greedy property** (hard to prove its correctness!). If you make a choice that seems the best at the moment and solve the remaining sub-problems later, you still reach an optimal solution. You will never have to reconsider your earlier choices.