Las dimensiones del puzzle

El problema consiste en determinar la longitud del lado más largo de un “supuesto” puzzle. Como entrada, se proporciona el número de piezas totales del puzzle.

Solución Time Limit

Para el caso del Time Limit se ha elegido un algoritmo iterativo.

Es un algoritmo que tiene como parámetros el largo del puzzle, su ancho, la mínima diferencia y n que es el tamaño total del puzzle. El algoritmo consiste en un bucle que va desde 0 hasta n (en el caso de que el número sea primo) o hasta encontrar la mínima diferencia durante el recorrido.

Para saber su mínima diferencia, miramos si mi largo, que se va actualizando en cada vuelta del bucle, junto a n, su módulo es 0, en ese caso usamos una variable auxiliar donde guardamos la diferencia de la división y si la resta entre la diferencia y la división es menor o igual que la diferencia mínima(al principio vale n) entra y modifica su valor con la resta y el lado más grande lo guardamos en ancho que es lo que mostraremos por pantalla.

La idea es como una curva cóncava, llegar hasta el punto más alto, que sería la diferencia mínima y a<partir del siguiente ya todos serían mayores, por ello cuando vemos que la diferencia entre largo y resto es mayor que mi diferencia actual, eso quiere decir que hemos encontrado la solución a nuestro problema y nos salimos del bucle poniendo el booleano éxito a false. El coste del algoritmo es O(n).

Código:

```
#include <iostream>
#include <fstream>
#include <climits>
#include <vector>
#include <time.h>


using namespace std;


void haypuzzle(int &largo,  int &ancho,  int &dif,  int &n)
{
	
	bool exito = true;
	int i = 0;
	 int resto;
	while (i < n && exito)
	{

		resto = n / largo;
		if (n%largo == 0)
		{
			if (abs(largo - resto) <= dif)
			{
				if (largo > resto) { ancho = largo; }
				else {
					ancho = resto;
				}
				dif = abs(largo - resto);
			}
```

```
                    else exito = false;
                }
                largo++;
                i++;
        }

}

int resuelve(int n, int &tiempo)
{
        int largo = 1;
        int ancho = 1;
        int diferencia = n;
        int dif2 = n;
        int i = 1;
        int k = 1;
        clock_t t0, t1;

        t0 = clock();
        haypuzzle(largo, ancho, diferencia, n);
        t1 = clock();

        tiempo= (double)(t1 - t0) / (CLOCKS_PER_SEC)*1000.0;

        return ancho;
}


int main() {
        // Para la entrada por fichero.

#ifndef DOMJUDGE
        ifstream in("casosTLE.txt");
        auto cinbuf = std::cin.rdbuf(in.rdbuf());
#endif
        ofstream salida("salida.xls");
        int n, ancho, tiempo;
        cin >> n;
        salida << "Entrada\tTiempo" << endl;
        // Resolvemos
        while (n != 0) {
                ancho=resuelve(n, tiempo);

                salida << n <<"\t"<<tiempo << endl;
                cout << "Execution Time: " << tiempo << endl;
                cout << ancho << endl;

                cin >> n;
        }
        salida.close();
#ifndef DOMJUDGE // para dejar todo como estaba al principio
        cin.rdbuf(cinbuf);
        system("PAUSE");
#endif

        return 0;
}
```

Solución eficiente

Dado un número n = número total de piezas del puzle. Se haya su raíz cuadrada “x” y mientras que el módulo de “n” entre “x” sea distinto de 0 se va decrementando el valor de “x”, de esta manera se consigue que la distancia entre ancho y alto del puzle sea mínima. Finalmente se imprime la división de “n” entre “x” la cual es la longitud del lado más largo del puzzle.

Código:

```
#include <iostream>
#include <fstream>
#include <math.h>
#include <ctime>

using namespace std;


void resuelveCaso(int& n) {
	//resuelve aqui tu caso
	   //Lee los datos
	   //Calcula el resultado
	   //Escribe el resultado

	int x = sqrt(n);

	while (n % x != 0) {
		x--;
	}

	//cout << n / x << endl;

}

int main() {
	// Para la entrada por fichero.
#ifndef DOMJUDGE
	std::ifstream in("casosTLE.txt");
	auto cinbuf = std::cin.rdbuf(in.rdbuf());
#endif
	ofstream salida;
	salida.open("salida2.xls");
	int n;
	std::cin >> n;
	salida << "Entrada\tTiempo" << endl;
	// Resolvemos
	while(n != 0){
		clock_t start = clock();
		resuelveCaso(n);
		float time = ((double)(clock() - start) / CLOCKS_PER_SEC);
		salida << n << "\t" << time << endl;
		cout << "Execution Time: " << time << endl;
		std::cin >> n;
	}
	salida.close();
#ifndef DOMJUDGE // para dejar todo como estaba al principio
	std::cin.rdbuf(cinbuf);
	system("PAUSE");
#endif

	return 0;
}
```

Comparación de gráficas

Comparando el código eficiente con el TLE, podemos comprobar que los tiempo de ejecución son dispares, siendo constante el eficiente mientras que el TLE no lo es.