MEMORIA
PROYECTO DE DIVIDE Y VENCERÁS - AED2
Problema 9

JOSE RAMON HOYOS BARCELO

Nombre	Grp	e-mail
Angel Ruiz Fernandez	G2.2	a.ruizfernandez@um.es
Alberto Velasco Cotes	G2.2	a.velascocotes@um.es

1. Diseño

El problema consiste en reconocer patrones en una cadena. El patrón consiste en cadenas compuestas por una combinación de dos subcadenas de 3 caracteres, formando 6 caracteres.

En este caso, la tecnica de divide y vencerás se aplica dividiendo la cadena por la mitad. Si la división es menor de 6 caracteres, se descarta pues el patrón es de 6 caracteres. Si la división es de 6 caracteres, se comprueba el patrón de forma directa, pues la cadena no se puede dividir. Y si la división es de 7 o más caracteres, se divide en 2 por la mitad.

Puede ser que la división parta una ocurrencia del patrón, y que debido a eso no se detecte, por tanto se ha de combinar la zona central al rededor de la división en busqueda de posibles occurencias de manera directa. Esta zona comprende 10 caracteres, 5 y 5, ya que la cadena de 6 puede tener un caracter en una división y 5 caracteres en la otra.

El codigo se compone de 3 funciones, un check de comprobación directa de 6 caracteres, la función de divide y vencerás, y el punto de entrada. A la función de divide y vencerás que es recursiva, se le pasa la cadena a comprobar las subcadenas que permutar para el patrón, y el rango de la cadena a comprobar y un conjunto de soluciones por referencia donde insertar estas. Las soluciones consisten en la posición de las ocurrencias de los patrones encontrados en la cadena.

```
funcion comprobar(cadena A, cadena sub[], entero i, entero d) {
    suba = A.subcadena(i, d - i + 1)
    devolver (suba está entre permutaciones de dos cadenas de sub)
}

funcion dyv(cadena A, cadena sub[], entero i, entero d, set<int> sol) {
    longitud = d - i + 1
    si longitud < 6:
        devolver
    si longitud == 6:
        si comprobar(cadena A, cadena sub[], entero i, entero d):
            insertar i a sol
    sino:
    // mitad izquierda
        dyv(A, sub, i, i + (longitud/2 - 1), sol)
    // mitad derecha
        dyv(A, sub, i + longitud/2, d, sol)
    // combinar resultados revisando la franja de unión
    para entero h desde i+(longitud/2-5) hasta (i+(len/2+4)
        si comprobar(A, sub, h, h+5)
            insertar h a sol
}

main() {
   cadena sub[3] = {"acb", "aac", "dca"}
   set<int> sol
   cadena A
   leer A
   dyv(A, sub, 0, A.length()-1, sol)
}
```

2. Analsis teorico

El algoritmo es recurrente, y su tiempo de ejecución viene dado en terminos de comprobaciones directas por la ecuación de recurrencia

$$t(n) = 2*t(n/2) + 5$$

ya que las divisiones toman la mitad de tiempo que la cadena actual, más las 5 comprobaciones resultantes del combinar, en el centro.

En el mejor caso con n < 6, ninguna comprobación se ejecuta.

$$t(n) = 0$$

En el mejor del resto de casos, se harán las mismas comprobaciones que la solución directa, es decir n - 5, el minimo posible. Este caso tiene una n que pertenece a n = 6 * 2^k. Su tiempo viene dado por

$$t\_m(n) = n/6 + 5*(2^{\log\_2(n/6)} - 1)$$

Operando con las anteriores formulas se prueba (no mostrada) que es cierto

$$t\_m(n) = n - 5$$

Por como está hecho el bucle que comprueba la zona combinada al rededor del centro, no se comprueban nunca posiciones redundantes ni con el caso base ni con otras zonas combinadas, por tanto el peor caso es el mismo que el mejor caso y en todo caso el tiempo es

$$t(n) = t\_M(n) = t\_m(n) = n - 5$$

Por tanto el orden del algoritmo es lineal

$$O(n)$$

En versiones anteriores del proyecto, los limites de combinar estaban distintos y se hacian comprobaciones redundantes provocando que t(n) fuera mayor que n - 5

Hasta ahora no se ha tenido en cuenta el coste de guardar las soluciones, que se insertan en un conjunto. que vamos a asumir que es O(log(size())), que por k soluciones en la cadena sería O(klogk) si size() empieza en 0. Sumandolo queda como

$$t(n) = t\_dyv(n) + t\_setinsert(n + klogk)$$

En promedio, el numero de soluciones k es proporcional a n, ya que la aparición de soluciones es aleatoria, con lo cual, tras la suma, el orden sigue siendo de

$$O(n)$$

Teniendo esto en cuenta, el peor caso es cuando haya el mayor numero de soluciones posible, n / 3, y el mejor, cuando no haya soluciones, k = 0.

3. Implementación

La implementación se realiza en C++ usando std::string, y std::set para guardar las soluciones. Se evitan las globales optando por pasar parametros.

Esta implementación no está especialmente optimizada. Se copian muchas cosas, se hacen computos redundantes que se pueden precalcular, se pasan muchas cosas en cada llamada que sería mas rapido obtener del espacio de nombres global, y se construyen objetos innecesariamente. Así que aparte se ha hecho una reimplementación en C (no incluida), mitigando todas estas ineficiencias, ya que las limitaciones del lenguaje te obligan a pensar de forma optima.

3.1. Listado de código simplificado

```main.cpp (simplificado)
#include <iostream>
#include <string>
#include <set>

bool comprobar(const std::string& a, const std::string sub[3], int i, int d) {
    // construir subcadena para comprobar
    std::string suba = a.substr(i, d - i + 1);
    // comprobar subcadena con permutaciones de las dos subcadenas del patron
    return
        (suba == sub[0] + sub[1]) ||
        (suba == sub[0] + sub[2]) ||
        (suba == sub[1] + sub[0]) ||
        (suba == sub[1] + sub[2]) ||
        (suba == sub[2] + sub[0]) ||
        (suba == sub[2] + sub[1]);
}

void dyv(const std::string& A, const std::string sub[3], int i, int d,
    std::set<int>& sol
) {
    int len = (d - i + 1); // longitud de subcadena
    if (len < 6) {
        // caso sub-base
    } else if (len == 6) {
        // caso base
        if (comprobar(A, sub, i, d))
           sol.insert(i);
    } else {
        // dividir
        dyv(A, sub, i, i + ((len/2) - 1), sol); // mitad izq
        dyv(A, sub, i + (len/2), d, sol); // mitad dcha
        // combinar
        for (int h = std::max(i + ((len/2) - 5), 0);
                h + 5 <= std::min(i + ((len/2) + 4), d); h++)
            if (comprobar(A, sub, h, h + 5))
                sol.insert(h);
    }
}
```

```
int main() {
    // leer cadena
    std::string A;
    std::cin >> A;

    // subcadenas a 2-permutar para patron
    std::string sub[3] = {"acb", "aac", "dca"};

    // conjunto de soluciones
    std::set<int> sol;

    // iniciar divide y venceras
    dyv(A, sub, 0, A.length() - 1, sol);

    // imprimir soluciones
    for (int s : sol)
        printf("%d, ", s + 1);
    // imprimir numero de soluciones
    printf("\nsol: %ld\n", sol.size());
}
```

4. Validación

Como validación, se implementa el algoritmo de forma directa (no incluido), y se someten las dos implementaciones contra el mismo problema, y se comprueba que dan las mismas soluciones.

La implementación de el test consiste en un script de bash que genera una cadena a probar, de longitud 1 a infinito, y la procesa llamando a las dos implementaciones, y entonces comparando las dos salidas.

```validate.sh (simplificado)
#!/bin/bash

i=1

while :
do
    test=$(./generador $i)
    printf "$i "
    outdyv=$(echo $test | ./proyecto-cpp -dyv)
    outdir=$(echo $test | ./proyecto-cpp -dir)

    if [ "$outdyv" != "$outdir" ]; then
        printf "bad\n"
    else
        printf "ok\n"
    fi
    ((i++))
done
```

5. Estudio experimental

5.1. Tiempo

Se modifica el generador para poder generar casos peor, promedio y mejor que en este caso es variando la k, el numero de patrones que hay en la cadena generada. En el peor de los casos, el mayor numero de patrones, cada 3 caracteres. El promedio es el normal, aleatorio, y el mejor es cuando no hay instancias del patrón, generando una string conformada por 'x' de la longitud especificada.

Para estudiar empíricamente el tiempo del programa, se implementa otro script 'measure.sh' que va llamando al ejecutable en modo divide y venceras y en modo directo, y mide cuanto tardan en ejecutarse, incrementando n linearmente cada vez. El script genera una salida en CSV con n, el tiempo que se tardó para DyV y directo, y su ratio. El estudio se hace 3 veces, para el peor, mejor y caso promedios.

```measure.sh (simplificado)
i=1
while [ $i -le $points ]
do
    ./generador $n $case > /tmp/test
    timedyv=$(bash -c "time cat /tmp/test | ./proyecto-cpp -dyv")
    timedir=$(bash -c "time cat /tmp/test | ./proyecto-cpp -dir")

    ratio=$(echo "scale=6; $timedyv/$timedir" | bc -l)

    printf "$n,$timedyv,$timedir,$ratio\n"

    if [ "$mode" == "lin1" ]; then
        ((n++))
    elif [ "$mode" == "linn" ]; then
        ((n+=step))
    fi

    ((i++))
done
```

Estudiando los resultados, se puede observar facilmente que el problema no es adecuado para resolverse con divide y vencerás, pues este metodo tarda mas tiempo que el metodo directo. Además, se observa que el crecimiento es lineal, y que el peor tarda un poco mas que el mejor, que es muy similar al promedio, al haber pocas soluciones.

La regresión lineal (del promedio) soporta esta conclusión, ya que el error es bajo (es lineal).

	Promedio			Peor		
	Ordena	Pendient	Std Err	Ordena	Pendient	Std Err
DyV	0.0022	2.708e-7	1.31e-8	-0.017	3.923e-7	1.78e-8
dir	0.0007	2.338e-8	2.7e-9	0.0097	3.45e-7	6.37e-9

Tabla 1. Regresión lineal del promedio

A continuación se presentan los datos resultados usando GNU Plot (plot.sh)

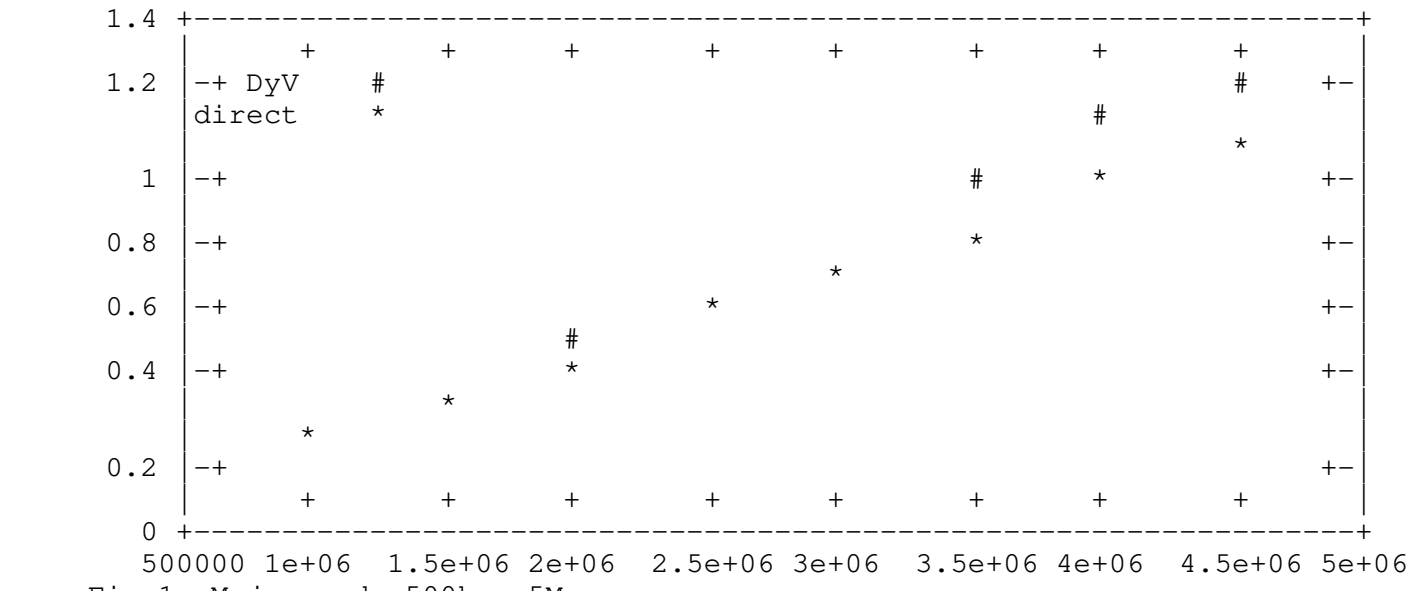

Fig 1. Mejor - de 500k a 5M

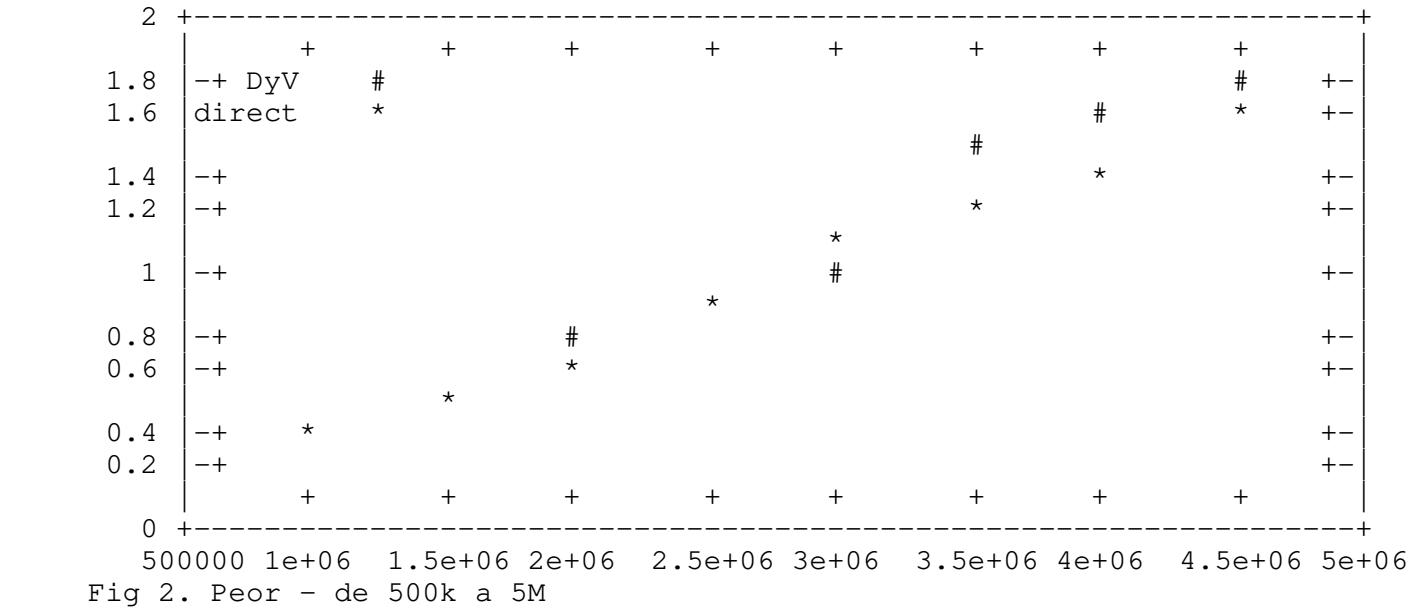

Fig 2. Peor - de 500k a 5M

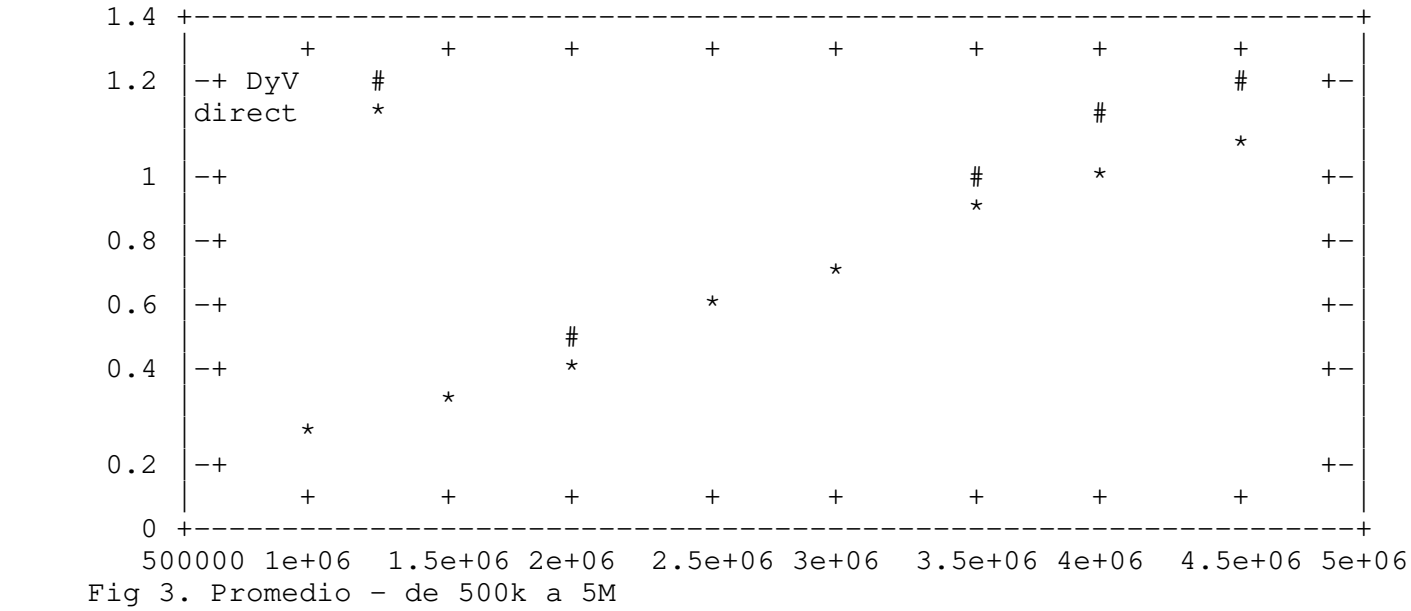

Fig 3. Promedio - de 500k a 5M

5.2. Memoria

Para investigar la memoria, habiendo generado los test apropiados, se ha usado la herramienta massif de valgrind, llamandola una sola vez, ya que el uso de memoria es mucho mas determinista, y interpretandose la salida con ms_print, que produce un grafico del perfil de uso a lo largo del tiempo, y las muestras indicando el pico de uso.

```
$ valgrind --tool=massif --stacks=yes ./proyecto-cpp < test
$ ms_print massif.out.*
```

Caso	Heap (MB)	Stack (B)
Mejor	3.027	928
Peor	14.556	2976
Promedio	3.026	928

Tabla 2. Pico de uso de memoria exacto en MB, con una n de 1M

Efectivamente, el mejor y el caso promedio al tener un numero de soluciones similar (0 y cercano a 0), consumen una cantidad de memoria similar, al guardar las soluciones en un contenedor std::set, y en el grafico tienen una pendiente bastante plana. Sin embargo el peor caso tiene un uso de memoria mucho mayor, y empinado, al hacer n / 3 inserciones.

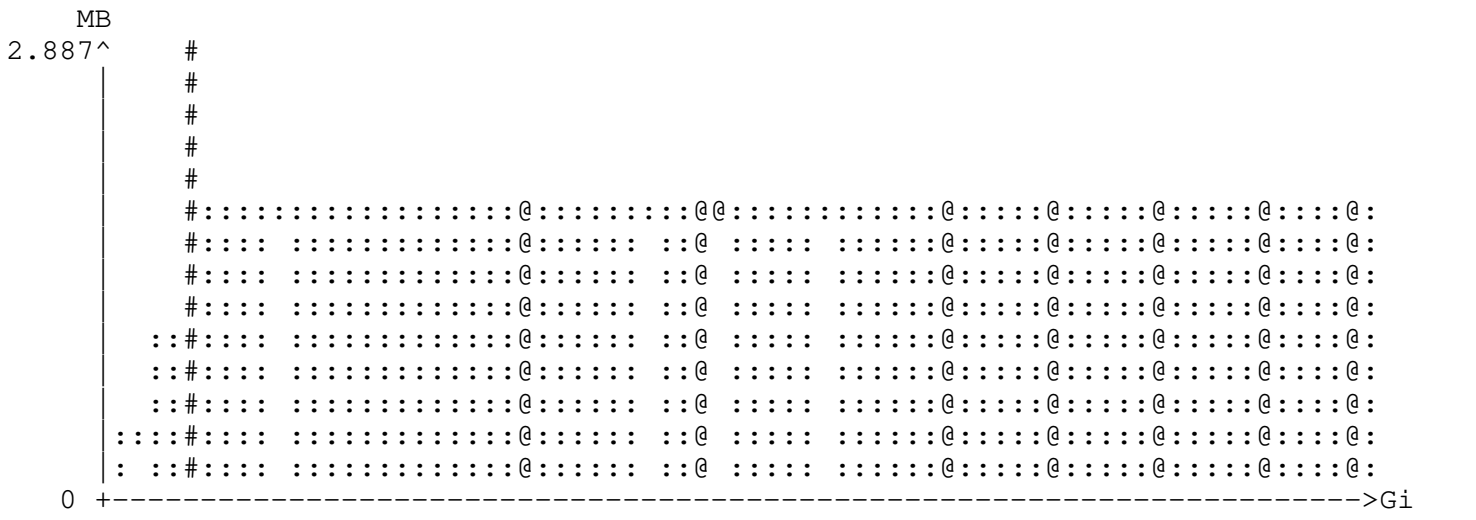

Fig 5. Perfil de uso de memoria – Mejor – n=1M

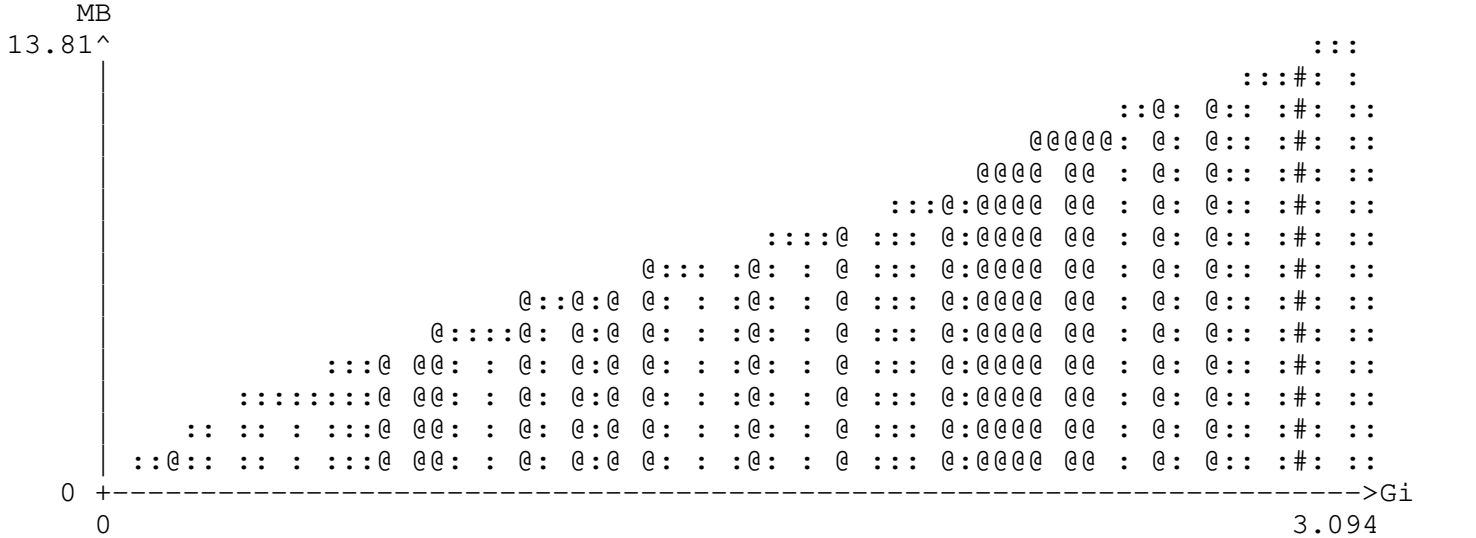

Fig 5. Perfil de uso de memoria – Peor – n=1M

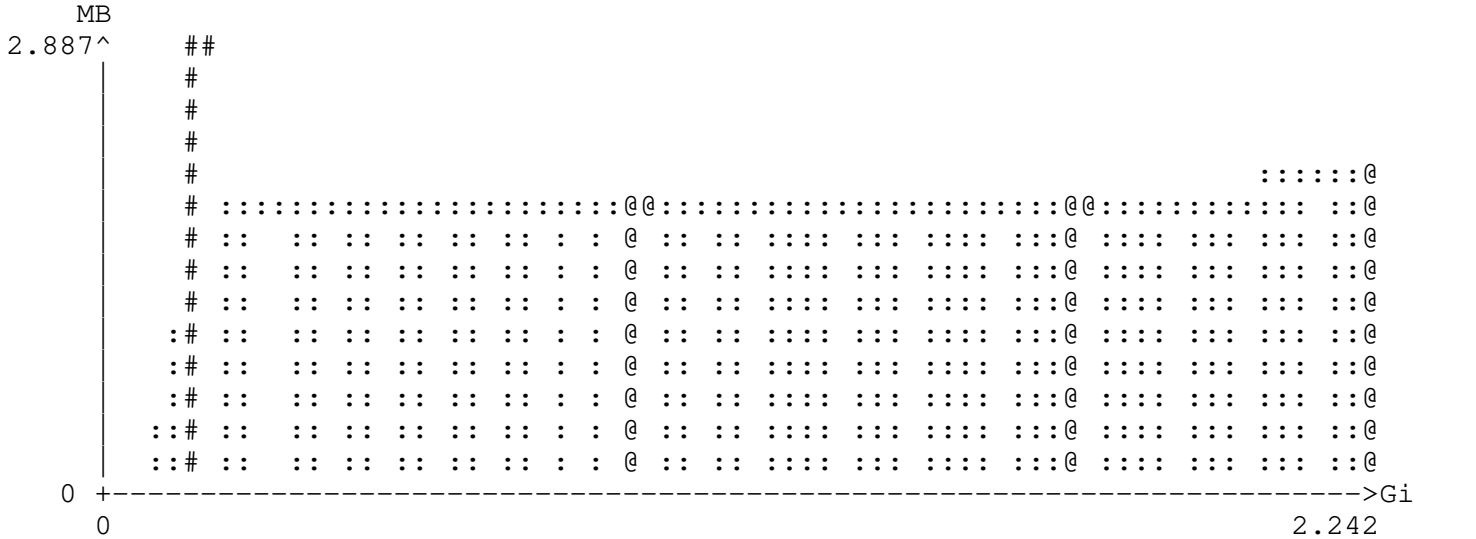

Fig 6. Perfil de uso de memoria – Promedio – n=1M

6. Contraste del estudio teorico y experimental

El estudio experimental de tiempo soporta la teoría de que el algoritmo es efectivamente de orden O(n), ya que la regresión linear sale con bajo error estandar.

Además tambien se observa que cuantas mas soluciones se encuentran, mas tiempo toma el algoritmo, confirmado que las inserciones suman al tiempo de ejecución significativamente. Además, el estudio de memoria propone que el uso de memoria es directamente proporcional de n, ya que el numero de soluciones en promedio es proporcional a n, al ser aleatorio.

7. Conclusiones y valoraciones

El metodo de divide y vencerás toma demonstrablemente mas tiempo que el directo, por tanto no es un problema adecuado para aplicar divide y vencerás.

Podemos especular que es mas lento por el tremendo arbol de llamadas que se forma, el overhead de el pase de parametros, la copia de valores, ... etc.

Muchas de estas ineficiencias se pueden eliminar, reimplementando el proyecto en C, obligandose a usar practicas eficientes y optimas. Tras hacerlo, de manera preeliminar se encuentra que esta versión (main.c) es más de un orden de magnitud mas rapida, y que la diferencia entre con o sin divide y vencerás se reduce.