UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Departamento de Física

Tarea 3

Física Computacional

Estudiante: Constanza Nahuelhual

Profesor: Edson Carquín

abril, 2024

1. Problem 1

1.1. Please create an algorithm that solves the ancient Towers of Hanoi puzzle. To solve the puzzle, you must come up with a series of steps to move a stack of different sized rings from one pole to another. They must be moved one at a time, using only simple intermediate pole, so that not ring is ever placed on top of a smaller ring.

Consideremos la siguiente distribución de la Torre de Hanoi:

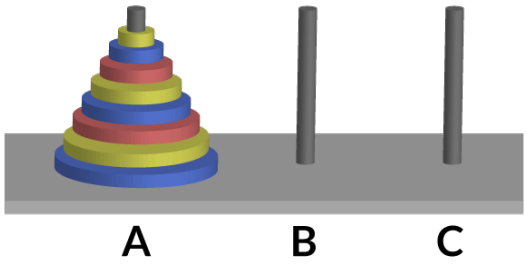

Figura 1: Torre de Hanoi

Para resolver el problema, es posible utilizar un algoritmo de recursión; moviendo los todos los anillos superiores de la torre A a la B, usando la C como auxiliar, para luego mover el anillo más grande a la torre C y por último moviendo el resto de los anillos de la torre B a la C utilizando la torre A.

Es decir, en el caso de 4 anillos:

```
Pasos a seguir para resolver la Torre de Hanoi con 4 anillos
Mover anillo 1 desde A hasta B
Mover anillo 2 desde A hasta C
Mover anillo 1 desde B hasta C
Mover anillo 3 desde A hasta B
Mover anillo 1 desde C hasta A
Mover anillo 2 desde C hasta B
Mover anillo 1 desde A hasta B
Mover anillo 4 desde A hasta C
Mover anillo 1 desde B hasta C
Mover anillo 2 desde B hasta A
Mover anillo 1 desde C hasta A
Mover anillo 3 desde B hasta C
Mover anillo 1 desde A hasta B
Mover anillo 2 desde A hasta C
Mover anillo 1 desde B hasta C
```

Figura 2: Procedimiento para resolver la Torre de Hanoi. (Salida de la terminal tras utilizar el código creado para resolver el problema 1.3)

1.2. Please provide an estimate of the asymptotic behavior of the execution time (for $n \to \infty$) of the algorithm developed in (a)

Si nos detenemos a pensar cuántos movimientos se necesitan para resolver el problema en función de la cantidad de anillos, nos damos cuenta de lo siguiente:

- Si existe 1 anillo, se necesita 1 movimiento.
- Si existen 2 anillos, se necesitan 3 movimientos.
- Si existen 3 anillos, se necesitan 7 movimientos.
- Si existen 2 anillos, se necesitan 15 movimientos.

Es decir, la función que modela el tiempo de ejecución es $2^a - 1$. Si la aproximamos a 2^a y hacemos tender esta función a infinito obtendremos lo siguiente:

$$\lim_{a \to \infty} 2^a = \infty \tag{1}$$

Por lo tanto, cuando el número de anillos tiende a infinito, el algoritmo también lo hará.

1.3. Please create a program in C++, that implements your algorithm in (a).

Utilizando el algoritmo descrito en la sección 1.1 se crea la siguiente función descrita en el código que permite obtener los pasos para resolver la Torre de Hanoi.

```
//Función que imprime las instrucciones para resolver la Torre de Hanoi
void Move(int Num, string A, string C, string B){
    if(Num == 1){
        cout << "Mover anillo " << Num << " desde " << A << " hasta " << C << endl;}
    else{
        //LLama a la función con entradas: A como la torre inicial, B como la torre de llegada y C como la auxiliar
        Move(Num - 1, A, B, C);
        cout << "Mover anillo " << Num << " desde " << A << " hasta " << C << endl;
        //LLama a la función con entradas: B como la torre inicial, C como la torre de llegada y A como la auxiliar
        Move(Num - 1, B, C, A);}}
```

Figura 3: Función que entrega los pasos para resolver la Torre de Hanoi.

1.4. Be free to represent the output of your program by using a plot, a table or whatever you think is the best way to do it so. You could use an alternative language like Mathematica or Python in order to display the results obtained with your program, but in that case, you should provide us with precise instructions to manage properly your codes.

Se realiza el siguiente gráfico del tiempo de ejecución del algoritmo en función de la cantidad de anillos. Notar que coincide con el análisis anterior cuando el número de anillos tiende a infinito. El código fue realizado en el lenguaje Python3.

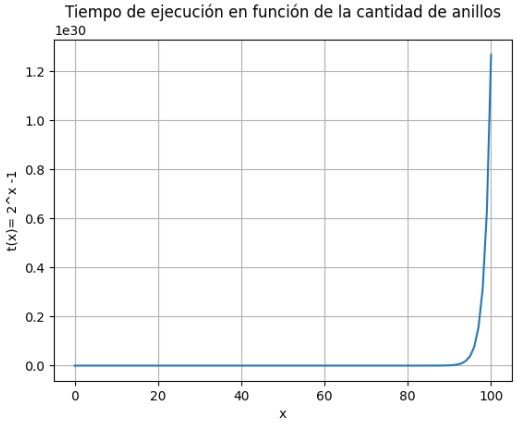

Figura 4: Tiempo de ejecución en función de la cantidad de anillos.

2. Problem 2

2.1. Write a program using C++, which implements an algorithm to calculate recursively $\sqrt{2}$, with a precision of 6 decimal digits.

Se utiliza un algoritmo de recursión del tipo Newton-Raphson, la función empleada es la siguiente:

A continuación se encuentra el resultado descrito.

```cpp
//Función que aproxima la raíz cuadrada con el método de Newton-Raphson
float SquareRoot(float Num, float Aprox, float Tol){
    float Ap = Aprox*Aprox;
    if(abs(Ap - Num) < Tol){return Aprox;}
    else{
        float NewAprox = (Aprox + Num/Aprox)/2;
        return SquareRoot(Num, NewAprox, Tol);}}
```

Figura 5: Función que calcula la raíz cuadrada en base a aproximaciones con el método de Newton-Raphson.

```
elizabeth@Laptop-de-Elizabeth Computacional % g++ SquareRoot.cc
elizabeth@Laptop-de-Elizabeth Computacional % ./a.out
1.414216
elizabeth@Laptop-de-Elizabeth Computacional %
```

Figura 6: Salida de la terminal.

3. Problem 3

3.1. Write a C++ program that reads a word from the keyboard, stores it in a string and checks whether the word is a palindrome. A palindrome reads the same from left to right as from right to left (otto, level and deed are examples of palindromes). Use the subscript operator [ ]. Modify the code to continually read and check new words and store the output into a file with the test word and the result of the test in the same line.

Se crea un código que contiene la función Palíndrome que recibe la palabra escrita en la terminal y la compara con la misma palabra escrita al revés. Esto lo almacena en un archivo.dat.

```cpp
//Función que recibe la palabra y el archivo.dat a crear
//Esta define si la pabra es palíndrome o no
void Palindrome(string String, ofstream &archive){
    //Almacena la palabra en otra variable
    string StringR = String;
    string Si = " es un palindrome";
    string No = " no es un palindrome";
    //Da vuelta la palabra en la nueva variable
    reverse(StringR.begin(), StringR.end());
    //Compara ambas palabras
    if(StringR == String){
        //Guarda el resultado en un archivo
        archive << String << Si << endl;
        //Imprime el resultado en la terminal
        cout << String << Si << endl;}
    else{
        //Guarda el resultado en un archivo
        archive << String << No << endl;
        //Imprime el resultado en la terminal
        cout << String << No << endl;}
        return;}
```

Figura 7: Función Palíndrome

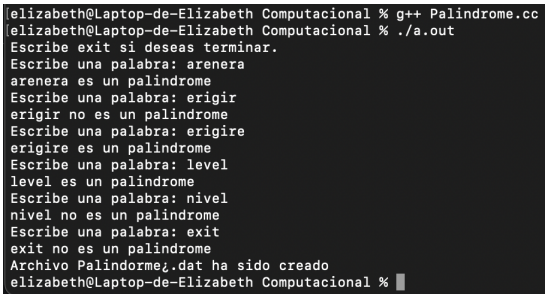

Figura 8: Entradas y salidas de la terminal utilizando el código Palindrome.cc

4. Problem 4

4.1. Explain each of the following definitions. Indicate whether any are illegal and if so why.

```
(a) int* ip;
(b) string s, *sp = 0;
(c) int i; double* dp = &i;
(d) int* ip, ip2;
(e) const int i = 0, *p = i;
(f) string *p = NULL;
```

Figura 9:

(a) **Legal** "int* ip;" declara un puntero a una variable ip de tipo entero.

(b) **Legal** "string s" declara una variable de carácteres y "*sp" declara un puntero nulo.

(c) **Ilegal** "int i" declara una variable "i" de tipo entero, "double* dp = &i" define el puntero "dp" para que apunte a la dirección de memoria de una variable "double", en este caso "i", sin embargo, "i" está definido como "int".

(d) **Legal** "int* ip" declara dos variables "ip" y "ip2 "de tipo entero.

(e) **Legal** "const int i = 0" declara una variable constante "i" de tipo entero, "p = &i;" determina un puntero "p" que apunta a la dirección de memoria de una constante de tipo entero.

(f) **Legal** "string *p = NULL;" define una variable de tipo string y la inicializa con el valor nulo.

4.2. Given a pointer, p, can you determine whether p points to a valid object? if so, how? It not, why not?

El puntero sólo almacena la dirección de memoria a la que apunta, pero no almacena ninguna información con respecto a la validez de ubicación a la que apunta. Así que no es posible determinar su validez sólo observando el puntero, sino que también se debe mirar la variable asociada.

Sin embargo, es posible probar la validez de un puntero usando un condicional, ya que éste arrojará "NULL" en caso de que esté apuntando a una dirección no válida.

4.3. Why is the first pointer initialization legal and the second illegal?

```
int i = 42;
void *p = &i;
long *lp = &i;
```

Figura 10:

El primer puntero inicializado "void *p = &;" es **legal** ya que se está apuntando a la dirección de memoria de la variable entera "i", esto es posible debido a que al definir el puntero "void" es posible apuntar a cualquier tipo de variable. En el otro caso, "long *lp = &i;" es **ilegal**, dado que está definiendo que el puntero "lp" apunta a una variable de tipo "long", entero de 4 bytes (muy largo), pero la variable "i" es de tipo "int".