Asignatura: Fundamentos de algoritmia
Evaluación continua algoritmos recursivos. Cuestionario.

Profesor: Isabel Pita.

Nombre del alumno:

Progresión aritmética: sucesión de números llamados términos, en la que cada término se obtiene sumando al término anterior una constante llamada *diferencia*.

Suma de una progresión aritmética: $\sum_{i=1}^{i=n} a_i = \frac{(a_1+a_n)\times n}{2}$

Progresión geométrica: sucesión de números llamados términos, en la que cada término se obtiene multiplicando el término anterior por una constante denominada *razon*: r

Suma de una progresión geométrica: $\sum_{i=1}^{i=n} a_i = \frac{(a_n\times r)-a_1}{r-1}$

1. (1 punto) El siguiente algoritmo comprueba si una cadena de caracteres es capicúa. Completa la implementación con uno o varios casos base. LLamada inicial: `capicua(s,0,s.length());`

```
bool capicua (std::string const& str, int ini, int fin) {
    if (...) ....
    else return str[ini] == str[fin-1] && capicua(str, ini+1, fin-1);
}
```

Respuesta:

La función sucesor empleada en la llamada recursiva disminuye el tamaño del vector en dos elementos. Por lo tanto hacen falta dos casos base, uno para tratar los vectores de tamaño par y otro para tratar los vectores de tamaño impar. Si se consideran los casos base de cero y un elemento, como el resultado en ambos casos es `true` se pueden agrupar en un sólo condicional.

```
if (ini + 1 >= fin) return true; // caso base de 0 y 1 elementos
```

Se podrían considerar los casos base de uno y dos elementos. En este caso sería una precondición del algoritmo que el vector no fuese vacío.

2. (3 puntos) El siguiente algoritmo calcula el máximo de los valores de un vector de forma recursiva.

```
int maximo (std::vector<int> const& v, int ini, int fin) {
    if (...) ...;
    else {
        int m = (ini+fin-1) / 2;
        int iz = maximo(v,ini,m+1);
        int dr = maximo(v,m+1,fin);
        if (iz >= dr) return iz;
        else return dr;
    }
}
```

Completa el algoritmo escribiendo uno o varios casos base. Plantea la recurrencia que permite obtener el coste del algoritmo. Despliega la recurrencia. Indica el orden de complejidad del algoritmo.

Respuesta:

a) Caso base: Dado que el algoritmo calcula el máximo de un vector y no se puede calcular el máximo de un vector vacío, es precondición del algoritmo que el vector no puede ser vacío. Por lo tanto hace falta un caso base de un elemento. Con este caso base es suficiente para resolver el problema.

```
if (ini+1 == fin) return v[ini];
```

b) Se realizan dos llamadas recursivas cada una con tamaño la mitad del tamaño inicial. El coste de cada llamada recursiva es constante, porque solo se realizan operaciones aritméticas y lógicas, todas ellas de coste constante.

La recurrencia que permite calcular el coste del algoritmo es:

$$T(n) = \begin{cases} c_0 & if \quad ini + 1 == fin \\ 2T(n/2) + c_1 & if \quad ini + 1 < fin \end{cases}$$

siendo n el tamaño del vector, dado por $fin - ini$.

c) El desplegado se puede consultar en el cuaderno de problemas de complejidad de algoritmos recursivos en el campus.

d) El coste final del algoritmo es $\mathcal{O}(n)$.

Se observa que no hay ninguna ventaja respecto al coste si se utiliza este algoritmo o si se utiliza un algoritmo iterativo que recorra el vector.

3. (2 puntos) El siguiente algoritmo resuelve el problema de las torres de Hanoi. Plantea la recurrencia que permite calcular su coste, haz el desplegado e indica el orden de complejidad del algoritmo.

```
void Hanoi (int n, int ini, int fin, int aux) {
    if (n == 1) std::cout << ini << ' ' << fin << '\n';
    else {
        Hanoi(n-1, ini, aux, fin);
        std::cout << ini << ' ' << fin << '\n';
        Hanoi(n-1, aux, fin, ini);
    }
}
```

Respuesta:

Se realizan dos llamadas recursivas cada una con tamaño una unidad menos del tamaño inicial. El coste de cada llamada recursiva es constante, porque solo se realizan operaciones aritméticas y de escribir por consola, todas ellas de coste constante.

La recurrencia que permite calcular el coste del algoritmo es:

$$T(n) = \begin{cases} c_0 & if \quad n == 1 \\ 2T(n-1) + c_1 & if \quad n > 1 \end{cases}$$

El coste final del algoritmo es $\mathcal{O}(2^n)$.

El desplegado se puede consultar en el cuaderno de problemas de complejidad de algoritmos recursivos en el campus.

4. (2 puntos) Dado el problema de contar el número de veces que aparece el dígito 1 en un número:

a) Escribe una definición recursiva del problema. Observa que se pide una *definición recursiva*, no la implementación en un lenguaje de programación.

b) Indica si la definición recursiva anterior daría lugar a una implementación final o no final del problema. Justifica tu respuesta.

c) Indica la recurrencia que permite calcular el coste en el caso peor del algoritmo y el coste final del algoritmo.

Respuesta:

a) Definición recursiva cuya implementación daría lugar a una recursión no final.

$$digito1(n) = \begin{cases} 0 & if \quad n < 10 \wedge n \neq 1 (caso\ base) \\ 1 & if \quad n < 10 \wedge n == 1 (caso\ base) \\ digito1(n/10) & if \quad n \geq 10 \wedge n\,\%10 \neq 1 \\ digito1(n/10) + 1 & if \quad n \geq 10 \wedge n\,\%10 == 1 \end{cases}$$

Definición recursiva cuya implementación daría lugar a una recursión final.

$$digito1(n, cont) = \begin{cases} cont & if \quad n < 10 \wedge n \neq 1 (caso\ base) \\ cont + 1 & if \quad n < 10 \wedge n == 1 (caso\ base) \\ digito1(n/10, cont) & if \quad n \geq 10 \wedge n\,\%10 \neq 1 \\ digito1(n/10, cont + 1) & if \quad n \geq 10 \wedge n\,\%10 == 1 \end{cases}$$

b) La recurrencia que permite calcular el coste del algoritmo, si consideramos n el número de entrada al algoritmo, es:

$$T(n) = \begin{cases} c_0 & if \quad n < 10 \\ T(n/10) + c_1 & if \quad n >= 10 \end{cases}$$

El coste final del algoritmo es $\mathcal{O}(\log n)$ siendo n el número de entrada.
Si consideramos n el número de cifras del número de entrada, entonces la recurrencia es:

$$T(n) = \begin{cases} c_0 & if \quad n < 10 \\ T(n-1) + c_1 & if \quad n >= 10 \end{cases}$$

y el coste final del algoritmo es $\mathcal{O}(n)$ siendo n el número de cifras del número de entrada.

5. (1 punto) Escribe una definición recursiva de una función que dado un número devuelva su inverso. Por ejemplo, dado el 23456 debe devolver el 65432. Indica si la recursión aplicada es final o no final.

Respuesta:

Definición no final. La función devuelve dos valores, el número invertido y 10^k, siendo k el número de cifras del número

$$inv(n) = \begin{cases} < n, 1 > & n < 10 \\ < n\,\%10 * inv(n/10).second + inv(n/10).first, 10 * inv(n/10).second > & n >= 10 \end{cases}$$

Definición final. La función tiene dos parámetros, el primero es el número que debemos invertir y el segundo es un número que corresponde al inverso del valor con el que empezamos la recursión.

$$inv(n, k) = \begin{cases} k & n < 10 \\ inv(n/10, k * 10 + n\,\%10) & n >= 10 \end{cases}$$

6. (1 punto) Dada la siguiente definición recursiva del cálculo de una potencia, implementa el algoritmo de forma eficiente.

$$pot(n, e) = \begin{cases} 1 & e == 0 \\ n & e == 1 \\ pot(n, e/2) * pot(n, e/2) & e > 1 \wedge e\,\%2 == 0 \\ pot(n, e/2) * pot(n, e/2) * n & e > 1 \wedge e\,\%2 == 1 \end{cases}$$

Respuesta:

La implementación debe ser eficiente, por ello no se pueden repetir llamadas recursivas iguales. se guarda su valor en una variable para utilizarlo después.

```
int pot ( int n, int e){
  if (e == 0) return 0;
  else if (e == 1) return n;
  else  {
    int p = pot(n/2);
    if (e % 2 == 0) return p * p;
    else return p * p * n;
  }
}
```