# Listado uno de problemas - Algoritmos y estructura de datos

Algoritmos Computacionales (Instituto Politécnico Nacional)

**1era Lista de Problemas de Algoritmos y Estructuras de Datos**

**(Primavera-Verano 2021)**

**Cristhian Alejandro Ávila-Sánchez**

**0.** En el siguiente código, determine los valores que van adquiriendo las variables y los apuntadores tras la ejecución de cada sentencia. Asuma que las variables a, b y c residen en las direcciones de memoria #A001, #B002 y #C003, respectivamente. Indique si alguna de las sentencias ocasiona un error de asignación a puntero nulo.

```
00 int main(int argc, char *argv[])
01 {
02     int *u= NULL, *v= NULL, *w= NULL, *q= NULL;
03     int a= 101, b= 201, c= 301;
04
05     u= &c; v= &b; w= &a;
06
07     (*w)+= a + b + c;
08     (*u)++;
09     (*v)*= 4;
10
11     q= w;  w= u;  u= v;  v= q;
12
13     (*u)-= (*q)%5 - a;
14     (*v)-= (*q)%3 - b;
15     (*w)-= (*q)%2 - c;
16
17     printf("a=%d b=%d c= %d\n", a, b, c);
18
19     return 0;
20 }
```

**1.** Codifique funciones que aparten y liberen memoria, de forma segura, para un arreglo dinámico `arr` de *N* enteros. Averigüe la dirección de memoria, en hexadecimal, de cada una de las celdas `arr[k]` e inicialice la *k*-ésima celda con el valor entero, en base *k*, de la dirección de memoria de la celda correspondiente.

**2.** Utilizando exclusivamente aritmética de apuntadores, codifique una función que reciba una cadena de entrada, invierta sus caracteres y que aloje la cadena invertida en una nueva cadena de salida. La cadena original debe permanecer inalterada.

**3.** Después de ejecutar cada sentencia, determine los valores que adquieren las variables, los apuntadores sencillos y los apuntadores dobles:

```
00 int main(int argc, char *argv[])
01 {
02     int **q= NULL, *p= NULL;
03     int *a= NULL, *b= NULL, *x= NULL, *y= NULL;
```

```
04      int N= 10, k=0;
05
06      a= (int *) malloc(N*sizeof(int));
07      b= (int *) malloc(N*sizeof(int));
08
09      for (x= a, y= b, k=0; k<N; k++, x++, y++)
10         {
11            (*x)= 2*k;
12            (*y)= 3*k;
13         }
14
15
16      for (k=0; k<N; k++)
17         {
18            if (k%2==0)
19              q= &a;
20            else
21              q= &b;
22
23            p= *q;
24            p= p+k;
25            (*p)*= -1;
26
27            (**q)+= *p;
28         }
29
30      for (k=0; k<N; k++)
31         printf("%X :: a[%d] = %d\n", &(a[k]), k, a[k]);
32
33      printf("\n");
34
35      for (k=0; k<N; k++)
36         printf("%X :: b[%d] = %d\n", &(b[k]), k, b[k]);
37
38      free(a);
39      free(b);
40
41      return 0;
42  }
```

**4.** Emplee un apuntador doble `int **dinosaurio` para crear un arreglo de apuntadores sencillos a datos enteros de longitud *N*. En las casillas impares `dinosaurio[m]` cree un arreglo de enteros comunes y silvestres de longitud *m*. En las casillas pares `dinosaurio[n]` guarde la dirección de la celda `dinosaurio[(n+1)%N][n].` ¿Cómo puede realizar estas operaciones sin incurrir en errores de manejo de memoria?

**5.** Aparte memoria para un hipercubo de dimensiones $A \times B \times C \times D \times E \times F$ empleando un apuntador séxtuple `int ******hipercubo`.

**6.** Considere un problema computacional $X$. En general, ¿siempre podrá encontrar un algoritmo para poder resolverlo? Argumente su respuesta en términos de computabilidad.

**7.** Describa las clases computacionales *P*, *NP*, *NP-Completo* y *NP-Difícil*. Por cada clase, de un ejemplo de un problema que pertenezca a la misma.

**8.** ¿Existe algún problema computacional que se encuentre tanto en la clase P como en la clase NP? Si es así, de un ejemplo y proponga un algoritmo para resolverlo. ¿Esto significa que ambas clases de complejidad son iguales o son diferentes? ¿Su respuesta puede considerarse como una solución general para responder el problema P vs NP?

**9.** Calcule la complejidad computacional de cada uno de los ciclos que forman parte del siguiente código. ¿Cuál es el valor de la variable x que se despliega en consola entre iteraciones?

```
00 int main(int argc, char *argv[])
01 {
02     int k=0, x=0, i=0, j=0;
03
04     for (k=0, x=0; k<N; k++)
05        x+= k;
06
07     printf("x= %d\n");
08
09     for (k=1, x=0; k<N; k*=6)
10        x+= k;
11
12     printf("x= %d\n");
13
14     for (i=0, x=0; i<N; i+=2)
15        for (j=0; j<N; j+=3)
16           x+= i*j;
17
18     printf("x= %d\n");
19
20     for (i=N, x=0; i>0; i/=2)
21        for (j=N; j>0; j--)
22           x+= i*j;
23
24     printf("x= %d\n");
25 }
```

**10.** Demuestre que la complejidad en el peor de los casos del Algoritmo de Kadane, para encontrar la suma más grande de un arreglo de $N$ enteros, es $O(N)$.

**11.** Ordene el siguiente arreglo de números $arr = \{8,4,1,6,0,3,25,7,9\}$ mediante los algoritmos de *inserción*, *selección*, *burbuja* y *mezcla* (muestre las configuraciones relevantes del arreglo a través de los pasos de ordenamiento). Calcule la complejidad temporal de cada algoritmo e indique, para cada uno, cuántos pasos le toma ordenar este arreglo en específico.

**12.** Utilizando el algoritmo de búsqueda binaria iterativa, indique los pasos para buscar los números $x = 2,7,29,59,89$, en el arreglo:

$arr = \{1,2,4,16,28,29,33,40,52,54,55,58,59,64,65,75,83,89,90,94,95\}$.

Calcule la complejidad temporal del algoritmo utilizado.

**13.** Codifique un algoritmo recursivo para calcular el factorial $N!$, para $0 \leq N \leq 13$

**14.** Considere el siguiente programa recursivo. Considere, por el momento, que la impresión en consola le toma un tiempo constante. Dibuje el árbol de invocaciones recursivas y con base a ello, determine su complejidad computacional.

```
00 #include <stdio.h>
01 #include <stdlib.h>
02
03 void arbolDoble(int N, int nivel);
04 void arbolTriple(int N, int nivel);
05
06 int main(int argc, char *argv[])
07 {
08     int N= 64, nivel=0;
09
10     arbolDoble(N, nivel);
11     arbolTriple(N, nivel);
12
13     return 0;
14 }
15
16 void arbolDoble(int N, int nivel)
17 {
18     int k=0;
19
20     if (N<=0)
21       return;
22
23     printf("nivel= %d, N= %d\n", nivel, N);
24
25     arbolDoble(N/2, nivel+1);
26     arbolDoble(N/2, nivel+1);
27 }
28
29 void arbolTriple(int N, int nivel)
30 {
31     int k=0;
32
33     if (N<=0)
34       return;
35
36     printf("nivel= %d, N= %d\n", nivel, N);
37
38     arbolTriple(N/3, nivel+1);
39     arbolTriple(N/3, nivel+1);
40     arbolTriple(N/3, nivel+1);
41 }
```

**15.** Analice cuidadosamente el siguiente algoritmo. ¿Qué valores va adquiriendo la variable contador al irse invocando las funciones recursivas `arbolBinario()` y `arbolBaseB()`?

```
00 #include <stdio.h>
01 #include <stdlib.h>
02
03 void arbolBinario(int nivel, int nodo, int limite, int *cont);
04 void arbolBaseB(int nivel, int nodo, int limite, int base, int*cont);
05
06 void main()
07 {
08     int N=0, n=0, lim= 4, base=3;
09     int conteo=0;
10
11     arbolBinario(N, n, lim, &conteo);
12     printf("conteo arbol binario= %d\n", conteo);
13
14     conteo= 0;
15
16     arbolBaseB(N, n, lim, base, &conteo);
17     printf("conteo arbol base %d= %d\n", base, conteo);
18 }
19
20 void arbolBinario(int nivel, int nodo, int limite, int *cont)
21 {
22     int k=0;
23
24     if (nivel>=limite)
25       return;
26
27     for (k=0; k<nivel; k++)
28        printf("\t");
29
30     (*cont)++;
31
32     printf("nivel= %d, nodo= %d\n", nivel, nodo);
33
34     arbolBinario(nivel+1, 2*nodo+0, limite, cont);
35     arbolBinario(nivel+1, 2*nodo+1, limite, cont);
36 }
37
38 void arbolBaseB(int nivel, int nodo, int limite, int base, int *cont)
39 {
40     int k=0;
41
42     if (nivel>=limite)
43       return;
44
45     (*cont)++;
46
47     for (k=0; k<nivel; k++)
48        printf("\t");
49
50     printf("nivel= %d, nodo= %d\n", nivel, nodo);
```

```
51
52     for (k=0; k<base; k++)
53         arbolBaseB(nivel+1, base*nodo+k, limite, base, cont);
54 }
```

**16.** Codifique un algoritmo para resolver recursivamente el problema de las *Torres de Hanoi* para $N$ discos. Pruebe su algoritmo para $N = 3, 4, 5\ y\ 6$ discos. Calcule la complejidad de su algoritmo. ¿Cuántos pasos le tomaría para resolver el problema con $N = 8, 16, 32\ y\ 64$ discos?

**17.** Codifique el algoritmo recursivo de *Quicksort* y calcule su complejidad temporal. Indique los pasos para ordenar el arreglo números $arr = \{8,4,1,6,0,3,25,7,9\}$ mediante el empleo de este algoritmo.

**18.** Utilizando el algoritmo de búsqueda binaria recursiva, indique los pasos para buscar los números $x = 2,7,29,59,89$, en el arreglo:

$$arr = \{1,2,4,16,28,29,33,40,52,54,55,58,59,64,65,75,83,89,90,94,95\}.$$

Calcule la complejidad temporal del algoritmo utilizado.

**19.** Proponga un algoritmo recursivo, utilizando la técnica de *backtracking*, para entrar, recorrer y encontrar la salida de un laberinto.

**20.** Empleando la técnica de *backtracking*, diseñe un algoritmo para resolver el problema de las *"N Reinas"*. Calcule la complejidad de su algoritmo.