**Resumen de Algoritmo y Complejidada**

**Clase II Y III**

**Tiempo de Ejecución**

Generalmente la eficiencia de un código se contrapone a la sencillez y legibilidad de un programa. Es importante identificar cuándo y dónde preocuparse por la eficiencia, a veces es mejor preocuparse por la robustez y la sencillez de un programa.

Hablamos que el tiempo de ejecución de un programa y por ende la eficiencia depende de:

‣ El compilador

‣ El procesador

‣ Los datos

‣ El algoritmo o su complejidad algorítmica

Si tenemos que sumar un vector de n números, de que dependería el tiempo?

▸ Se puede decir que T(n) = cn ¿?

▸ int sumar (array vector){

int resultado=0;

for (i=0;i<vector.lenght();i++){ //N

resultado+=vector[i];// C

}

return resultado;

}Si tenemos una función que busca un numero en un arreglo y devuelve su posición:

int buscar (int a, int \*v, int l){

int j;

for(j=0;j<l;j++)

if(v[j]==a) break;

return j;

}

Cuanto Tardaria?

**Tiempo de Ejecución**

**Mejor Caso:** el objeto buscado esta en posición 0; T(n)=cj

**Peor Caso:** el objeto buscado esta en la posición n; T(n)=cn

**Caso Promedio:** el objeto esta a la mitad; T(n)=c.(n/2)

**Notación Asintótica**

**Asíntota:** línea recta que se aproxima continuamente a otra función o curva.

También se puede decir qué es la curva la que se aproxima continuamente a la recta; o que ambas presentan un comportamiento asintótico. Es utilizado para la comparación de diferentes algoritmos conociendo a esta como notación asintótica.

Decimos que el tiempo de ejecución de un algoritmo es cuando existen constantes tal que paraT(n) = O(n2) c, n0 > 0 T(n) ≤ cn2 n ≥ n0

**Orden de algoritmos**

Las funciones básicas utilizadas para denotar algoritmos son:

1 < log(n) < n < n < n2 < n3 < np < 2n < 3n < . . . < n! < nn

O(1)Tiempo constante= función que evalúa si un elemento x es nulo.

O(n)Recorre toda la lista = función que suma todos los números de un arreglo

O(n2)Recorre todo n veces = funciona que evalúa si cierto valor esta duplicado en una lista

O(2n ) funciones recursivas= en cada paso se duplica su tiempo. Por ejemplo el calculo recurso de numero de fibonacci.

int Fibonacci(int number){

if (number <= 1) return number;

return Fibonacci(number - 2) + Fibonacci(number - 1);

}

**Identificación de algoritmos según complejidad**

Si tenemos: 1 < log(n) < n < n < n2 < n3 < np < 2n < 3n < . . . < n! < nn

**BUENAS NOTICIAS**

O(1)

O(log(n))

O(n)

O(nlog(n))

**MALAS NOTICIAS**

O(nk ), k ≥ 2 O(kn ), k ≥ 2 O(n!) O(nn )

**Propiedades de O**

**Invariancia de constantes multiplicativas** es conocido como ya que implica la tasa de crecimiento. La tasa de crecimiento es invariable antes constantes multiplicaticas.

Es decir: Por ejemplo: n f(n) T(n) = O(c . f(n)) ⇒ T(n) = O(f(n)) T(n) = O(2n3 ) ⇒ T(n) = O(n3 )

**Transitividad** 15 es transitiva, es decir si y entonces O() T(n) = O(f(n)) f(n) = O(g(n)) T(n) = O(g(n))

**Regla de la suma** Si con y constantes positivas, entonces . Es decir, si en una expresión tenemos varios términos solo el “mayor” de ellos esta en sentido de Por ejemplo: f(n) = O(g(n)), a b a . f(n) + b . g(n) = O(g(n)) O( . . ) T(n) = 2n3 + 3n5

**Regla del producto**

Si T y entonces 1(n) = O(f 1(n)) T2(n) = O(f 2(n)) T1(n) . T2(n) = O(f 1(n) . f 2(n))

**Estimación de tiempo de operaciones**

Asumiendo que no existen recursiones, La suma de cada instancia básica nos retorna el tiempo de ejecución.

**Regla básica**

”Considerar los bucles o construcciones más internas e ir hacia las externas” Se asigna un costo computacional a la sentencia básica y a partir de esta se calcula el costo del bloque. Lo mismo se aplica a funciones y otras construcciones.

**Conteo de operaciones: bloques if**

f(cond){ hace algo; /\*cuerpo\*/

En el peor caso “hace algo;” se ejecuta siempre. Es decir:

**Tpeor = Tcond + Tcuerpo**

En el caso promedio, se calcula la probabilidad de que se

ejecute, es decir **Tprom = Tcond + P . Tcuerpo**

En caso de que tenga un else que pasaría?

if(cond){algo}else{otro} }

Tpeor = Tcond + max(Talgo, Totro)

Tpeor ≤ Tcond + Talgo + Totro

Tprom = Tcond + P . Talgo + (1 − P) . Totro

**Conteo de operaciones: bucles**

El caso más simple es cuando se ejecuta un número fijo de veces y el cuerpo del bucle es de tiempo contante.

for(i = 0; i < N; i + + ){ cuerpo; }

T = Tini + N. (Tcuerpo + Tinc + Tfin)

Tini es el tiempo de inicialización, x=0

Tinc es el tiempo del incremento, i++

Tfin es el tiempo de evaluación de la detención, i< N

El caso puede complicarse si el cuerpo no es constante. Para ese caso hay que calcular con el máximo de esa sentencia.

El caso aún más difícil es cuando no sabemos cuanto puede durar el bucle, como en el while: while(cond){algo; }

Para esta situación también se debe considerar cuando podría estadísticamente parar.

**Problemas NP**

Un algoritmo polinomial no determinista (NP), es aquel que no se ejecuta en un tiempo polinomial, por lo que dado un problema de clase NP no se conoce un algoritmo que sea capaz de dar solución en un tiempo polinomial. Estos problemas solo pueden ser resueltos encontrando las soluciones de forma aleatoria, para dar con la mejor solución posible. Estos solo pueden ser resueltos en un tiempo polinomial utilizando una máquina de Turing no determinista.

**Máquinas de Turing**

Es un dispositivo teórico que manipula símbolos sobre una cinta de acuerdo con una tabla de reglas definida. La máquina de Turing fue descrita por Alan Turing en 1936 como respuesta al ensayo del matemático alemán David Hilbert “*On Computable Numbers, with an Application to the Entscheidungsproblem*” donde demostró que algunas de estas máquinas de Turing serían capaces de realizar cualquier cálculo matemático concebible si fuera representable con un algoritmo .

Las máquinas de Turing Deterministas no permiten definir más de una instrucción para una situación en particular; por el contrario, las máquinas de Turing no Deterministas si lo permiten.