Universidad Nacional Autónoma de México

Facultad de Ciencias

Departamento de Matemáticas

México, Septiembre 2020.

Introducción al Análisis de Algoritmos

Notas de Clase

(Primera Parte, Segunda Versión)

Dra. María De Luz Gasca Soto

Licenciatura en Ciencias de la Computación,
Departamento de Matemáticas,
Facultad de Ciencias, UNAM.

Prefacio

Estas notas forman parte del material que se revisa en el curso de Análisis de Algoritmos I que se imparte en la Facultad de Ciencias de la UNAM, para la Licenciatura en Ciencias de la Computación.

He tenido la oportunidad de impartir este curso los últimos años y he estado preparando y corrigiendo las presentes notas de clase para el curso, de las cuales esta resulta ser la segunda versión.

Considero que las áreas de Análisis, Diseño y Justificación de algoritmos debe ser tomado más en serio por las personas que de una u otra manera diseñan programas o bien están involucradas con la computación.

El presente trabajo, pretende dar una panorámica general de lo que es el análisis de algoritmos. Enfatizando que el análisis, diseño y justificación de algoritmos se puede realizar de manera formal usando como herramienta a la Inducción Matemática.

La bibliografía básica, para el material presentado en estas notas, está basada en los libros:

- **Udi Manber** [13]
  Introduction to Algorithms. A Creative Approach.
- **J. Kingston** [12]
  Algorithms and Data Structures: Design, Correctness and Analysis.
- **R. Neapolitan & K. Naimipour** [16].
  Foundations of Algorithms.

Dra. María De Luz Gasca Soto
Profesor Asociado, T.C.
Departamento de Matemáticas
Facultad de Ciencias, UNAM.

Índice general

Capítulo 2

Análisis de Algoritmos

En este capítulo se definen algunos de los conceptos básicos relacionados con el análisis de algoritmos, tales como modelos de cómputo, complejidad, desempeño computacional, entre otros.

2.1. Introducción.

Udi Manber[13] nos indica que *" ... el propósito del análisis de algoritmos es predecir el comportamiento de un algoritmo sin implantarlo en una máquina específica."*

Para lograr independencia sobre el tipo de computadora, concebimos a ésta como una simple máquina de operaciones elementales. Luego, si un algoritmo usa menos operaciones elementales que otro será más rápido. Al cálculo de operaciones elementales se le llama **Tiempo de Ejecución del Algoritmo**. Las operaciones elementales son: asignación, comparación y operaciones aritméticas básicas (+,-,*,/). Por el momento, diremos que cada una de estas operaciones cuesta una unidad. Al calcular el tiempo de ejecución de un algoritmo debemos preguntar primero si el algoritmo termina, después preguntar si existe alguna caracterización precisa sobre la cantidad de tiempo (número de pasos) que tomará ejecutar tal algoritmo.

El análisis y comparacion de algoritmos, tradicionalmente, se ha basado en el tiempo de ejecución y tamaño (cantidad) de memoria utilizado para almacenar los datos sobre todos los ejemplares del problema.

La cantidad de espacio requerida en la solución de un problema está determinada por el problema en sí mismo, durante la implementación del algoritmo habrá una pequeña variación. Sin embargo, ocasionalmente, diferencias en la utilización del espacio serán importantes. Por ejemplo, muchos algoritmos operan con estructuras auxiliares efectuando transformaciones y luego copian los resultados al contenedor original. Esto podría incluso duplicar la cantidad de memoria requerida para la tarea. En algunos casos es posible mejorar el tiempo de ejecución de los algoritmos incrementando la cantidad de memoria para estructuras auxiliares.

Para estudiar la eficiencia de los algoritmos requerimos un modelo computacional. Un **modelo computacional** es un conjunto de suposiciones que se asumen acerca de una

máquina (virtual o real) sobre la cual será ejecutado el algoritmo. Para este curso, estaremos trabajando con el modelo computacional de comparaciones[12, 13]. Una vez elegido el modelo computacional, se debe seleccionar una medida de desempeño. Trabajaremos con el Tiempo de ejecución.

2.2. Complejidad.

En esta sección, describiremos detalladamente el concepto tiempo de ejecución, daremos las nociones básicas sobre orden, revisaremos cómo el desempeño computacional de los algoritmos puede ser descrito como una función del peor de los casos, del caso promedio o del tiempo amortizado.

Estamos trabajando con la medida de desempeño denominada tiempo de ejecución, la cual calcula el número de operaciones. Formalmente, el tiempo de ejecución de un algoritmo A, aplicado a un ejemplar específico E, es el número de pasos a efectuar en A para encontrar la solución de E, y lo denotaremos por $T_A(E)$. En particular, $T_A(E)$ representa, al menos, el número de comparaciones realizadas para encontrar la solución del ejemplar E.

El análisis de un algoritmo A en todos los casos, implicaría decir *para cada ejemplar* de E de un problema P (el que resuelve A), el tiempo de ejecución $T_A(E)$ que toma A para resolver E. Ya que, usualmente, el conjunto de *todos* los ejemplares E es muy grande, únicamente trabajaremos con familias de ejemplares de un mismo tamaño. Un ejemplar E es de tamaño n, si tiene n datos o una medida natural del tamaño, y se denota por $|E|$.

De esta forma, definimos $T_A(n)$ como el desempeño computacional para la clase de ejemplares E de tamaño n. En realidad sólo nos interesa el orden de la magnitud de $T_A(n)$, pues la implementación particular de un algoritmo se ve afectada únicamente por un factor constante.

Si $T(n)$ está asintoticamente acotada por $g(n)$, decimos que $T(n)$ es de orden $g(n)$ y lo denotamos como: $T(n)$ es $O(g(n))$.

De manera más formal, si f y g son funciones de variables no negativas $n_1, n_2, ..., n_k$ se dice que:

- f es $O(g)$ si $\exists c_1, c_2$ constantes positivas tales que: $f(n_1, ..., n_k) \leq c_1 \cdot g(n_1, ..., n_k) + c_2$. Representa una cota superior para f.

- f es $\Omega(g)$ si g es $O(f)$. Representa una cota inferior para f.

- f es $\Theta(g)$ si f es $O(g)$ y $\Omega(g)$.

La Figura 2.1 nos ilustra gráfica las cotas descritas. En la Figura 2.1(a) se ejemplifica la cota superior, en (b) la cota inferior y en (c) se tiene el acotamiento por ambos lados.

Ahora bien, dado un problema P, un algoritmo A que resuelva P y una definición de tamaño para los ejemplares de P, requerimos una función o expresión que nos indique del tiempo de ejecución del algoritmo A que dependa del tamaño del ejemplar E [13].

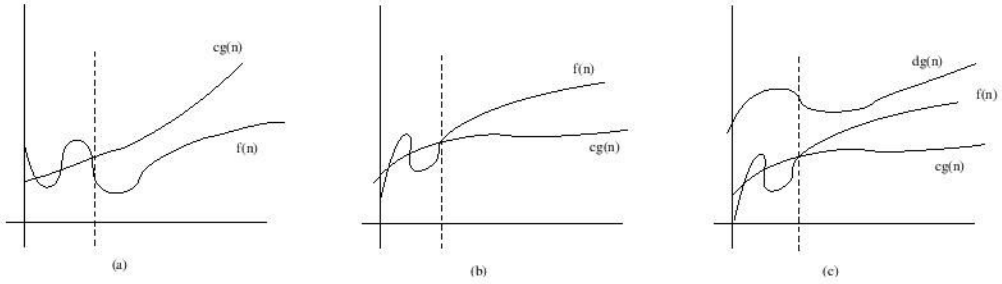

Figura 2.1: Ilustración de la notación O, Ω y Θ.

El desempeño computacional de un algoritmo puede ser analizado como una función del peor de los casos, del caso promedio o amortizado. A continuación se presenta una definición de las mismas.

Peor de los casos.

El desempeño computacional $T_A(n)$, de un algoritmo A sobre un ejemplar E, de tamaño n, se define como una función del peor de los casos, de la siguiente forma:

$$T_A(n) = T_A(E^*), \quad \text{donde} \quad T_A(E^*) = \max\{T_A(E) : |E| = n\}.$$

Es decir, el análisis del peor de los casos es la función definida por el máximo número de pasos tomados por el algoritmo para resolver cualquier ejemplar de tamaño n. El análisis del peor de los casos nos proporciona una ejecución garantizada del algoritmo. Esto es, un algoritmo nunca requerirá más tiempo o espacio del que ha sido especificado por la cota dada. Este análisis resulta ser muy pesimista.

Mejor de los casos.

El desempeño computacional $T_A(n)$, de un algoritmo A sobre un ejemplar E, de tamaño n, se define como una función del mejor de los casos, de la siguiente forma:

$$T_A(n) = T_A(E^*), \quad \text{donde} \quad T_A(E^*) = \min\{T_A(E) : |E| = n\}.$$

Es decir, el análisis del mejor de los casos es la función definida por el mínimo número de pasos tomados por el algoritmo para resolver cualquier ejemplar de tamaño n. En general este tipo de análisis no es de mucha utilidad, salvo para hacer comparaciones específicas.

Caso Promedio.

El análisis del caso promedio, nos proporciona una función que depende de la esperanza de $T_A(n)$, y donde Prob(E) es la probabilidad de que ocurra E. esto es:

$$\mathbf{E}[T_A(n)] = \sum_{\forall |E|=n} \text{Prob}(E) \cdot T_A(E)$$

Es decir, el análisis del caso promedio es la función definida por el número de pasos que se espera tome el algoritmo para resolver cualquier ejemplar de tamaño n. Este análisis

depende de la función de probabilidad con la que se distribuyan los datos. Resulta ser un análisis más preciso, pero es más complicado y específico. Al cambiar la forma como se distribuyen los datos, habrá que re-hacer el análisis.

La Figura 2.2 nos ilustra una gráfica en la que se describe el comportamiento del desempeño computacional de un algoritmo como una función del peor caso, del caso promedio y del mejor de los casos.

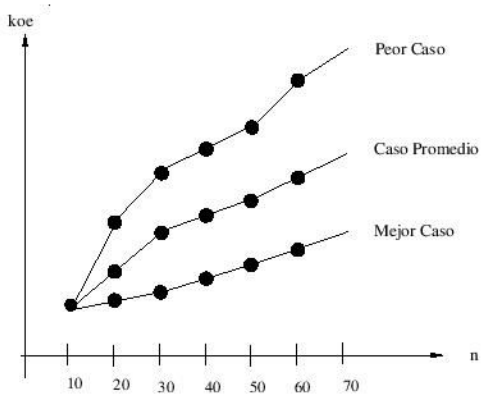

Figura 2.2: Complejidad en el Peor caso, caso promedio y mejor caso.

Tiempo Amortizado.

El análisis del tiempo amortizado es, intuitivamente, un promedio sobre el tiempo. Esto es, se toma $T_A(n)$ para diferentes instantes sobre la ejecución del algoritmo y después se calcula el promedio sobre tales valores.

Sea $T_A(E)_{t_i} = T_A(E)$ en el tiempo (instante) i. Si tomamos $T_A(E)$ para η instantes diferentes, el tiempo amortizado queda definido como:

$$T_A(n) = \frac{1}{\eta} \sum_{t_i=1}^{\eta} T_A(E)_{t_i}.$$

2.3. Cálculo del Tiempo de Ejecución.

En esta sección indicamos cómo calcular el tiempo de ejecución o desempeño computacional para los constructores de programación más comunes. Iniciamos con el tiempo constante, continuamos con los ciclos simples, después con los ciclos anidados y concluimos con llamadas a funciones.

A partir de este momento, suponemos que las operaciones elementales requieren tiempo constante, denotado por $O(1)$.

2.3.1. Tiempo Constante.

Una secuencia fija de operaciones de tiempo constante requiere también tiempo constante. Un ejemplo de esto es el proceso de intercambio de valores en un arreglo, sin importar el tamaño del arreglo. El Listado 3 mustra este proceso.

Listado 3 Algoritmo Intercambio

```
Intercambio(i,j){
   var t                    // variable temporal

       t   <— A[i]
    A[i]   <— A[j]
    A[j]   <— t
   }
```

Condicional If

El máximo tiempo que toma ejecutar un condicional **if** es la suma de los tiempo que toma ejecutar la prueba (condicion del if) y el máximo tiempo requerido por las alternativas. Si estas tres evaluaciones se ejecutan en tiempo constante, entonces el condicional if estará acotado por un valor constante. El Listado 4 muestra un ejemplo de este tipo.

Listado 4 Algoritmo ColorValido

```
ColorValido(a){          // recibe una arista a=(u,v)
   var u,v               // variable temporal

    u  <— a.inicial      // vertice inicial del arco a
    v  <—  a.final       // vertice final

    if (u.color=v.color)
      then return FALSE
    else   return TRUE
   }
```

2.3.2. Ciclos Simples.

Un ciclo que se ejecute un número constante de iteraciones tendrá un tiempo de ejecución constante. El Listado 5 muestra un ejemplo de este tipo. Para este ejemplo, tenemos que el ciclo siempre itera cinco veces, sin importar el valor del parámetro j y además el cuerpo del ciclo requiere tiempo constante, pues solo ejecuta dos asignaciones y dos operaciones aritméticas simples. Así que este algoritmo requiere en total tiempo constante.

Listado 5 Algoritmo UnCalculo

```
int UnCalculo(j){              // recibe un entero j
   var i,z,r                   // variables auxiliares

   z=r=1                       // valores iniciales
   for (i=1; i<=5; i++){
      z <-- j*i                // realiza calculos
      r <-- z-i
   }
   return r
}
```

En general los ciclos dependen de algún valor determinado por la entrada. Para describir el tiempo de ejecución de un ciclo, necesitamos caracterizar *cómo* se ejecutarán las iteraciones del ciclo y entonces calcular el tiempo de ejecución del cuerpo del ciclo. Un caso sencillo sucede cuando el ciclo está limitado por algún valor de la entrada y el cuerpo del ciclo requiere tiempo constante. Ejemplifiquemos esto, consideremos el problema de obtener el valor mínimo de un conjunto de datos. El Listado 6 presenta un algoritmo para este problema.

Listado 6 Algoritmo Obten Minimo

```
int ObtenMin(V){           // recibe un arreglo de enteros
  // PreC : V es no vacio y posee un valor minimo
  // PostC: regresa al minimo elemento de V.

  int n=V.length;          // tamanio de V.
  int m=V[0];              // valor minimo actual

   for (i=1; i<=n; i++){
     if (V[i] < m) then    // si el valor actual es menor
       m = V[i];           // modifica al minimo.
   }
   return m                // regresa al minimo
}
```

En el proceso ObtenMin el ciclo se ejecuta $(n-1)$ veces. Ya que cada iteración ejecutará a lo más una comparación y una asignación, el tiempo de ejecución para el cuerpo es constante. Por lo tanto, el tiempo total de ejecución del proceso es $O(n)$.

Un ciclo no necesariamente tiene una condición de terminación simple y por lo tanto se requiere un análisis cuidadoso para caracterizar el número de iteraciones que ejecutará el ciclo. Considere el problema de determinar si un entero es primo o no. El método clásico consiste en revisar sólo los enteros menores que la raíz cuadrada del valor dado. El Listado 7 muestra un algoritmo para este problema.

Listado 7 Algoritmo EsPrimo

```
boolean EsPrimo(n){          // recibe un entero n

   for (i=2; i*i<=n; i++){
     if (0==n mod i)         // Si es factor entonces
        return FALSE         // no es primo
   }
   return TRUE               // si salio, debe ser primo
}
```

En este caso, el ciclo termina cuando el valor de la variable del ciclo, i, excede el valor de la raíz cuadrada del dato original. Así que el número de iteraciones es a lo más $\sqrt{n}$. Ya que cada iteración del ciclo requiere tiempo constante, el algoritmo EsPrimo requiere en total tiempo $O(\sqrt{n})$.

2.3.3. Ciclos Anidados.

El caso más simple de ciclos anidados sucede cuando los límites de los ciclos son independientes. En este caso, el tiempo de ejecución es el producto de los valores que cada ciclo itera multiplicado por el tiempo de ejecución requerido por el cuerpo del ciclo. Un ejemplo clásico de este caso es la multiplicación de dos matrices de $n \times n$ para producir una nueva matriz de $n \times n$. El Listado 8 muestra una implementación del algoritmo.

El número de iteraciones en cada ciclo es de n. El cuerpo del ciclo más profundo requiere tiempo constante, ya que sólo realiza una suma, una multiplicación y una asignación. Por lo tanto, el tiempo total de ejecución es $O(n^3)$.

Listado 8 Algoritmo Producto de Matrices

```
ProdMat(A,B,C){          // recibe tres matrices
   var i,j,k             // contadores
   var n                 // dimension de A

   n=A.size;

   for (i=0; i < n; i++){
      for (j=0; i < n; j++){
         C[i][j] = 0.0;
         for (k=0; k < n; k++){
            C[i][j] += A[i][k]*B[k][j];
         }
      }
   }
}
```

Cuando los límites de la iteración para los ciclos internos están ligados con los de los límites externos, el análisis resulta un poco más complejo. Consideremos el problema de llenar diagonalmente una matriz. El Listado 9 ilustra un algoritmo para este problema.

Listado 9 Algoritmo Medio Llena

```
MedioLlena(A){          // recibe una matriz
   int  i,j             // contadores
   int  n               // dimension de A

   n=A.size;

   for (i=0; i<n; i++){
      for (j=i; j<n; j++){
         A[i,j] = j;
      }
   }
}
```

Para determinar el tiempo de ejecución de este proceso, simulamos la ejecución de pocos valores. Para la primera iteración del ciclo exterior el ciclo interior se ejecuta n veces. Para la segunda iteración del ciclo exterior el ciclo interior se ejecuta $n-1$ veces. Para la tercera iteración del ciclo exterior el ciclo interior se ejecuta $n-2$ veces; y así, hasta la última iteración que se ejecuta una vez. Por tanto, el número total de iteraciones resulta ser:

$$n + (n-1) + (n-1) + \cdots + 1 = \frac{n(n+1)}{2} \quad \text{es} \quad O(n^2)$$

2.3.4. Otros Ciclos.

El análisis de ciclos while o repeat ... until es similar al de los ciclos for, la clave está en determinar el número de iteraciones que se ejecutará el ciclo y finalmente calcular el tiempo requerido por el cuerpo del ciclo.

Un caso complicado del uso del while sucede cuando las variables involucradas no son simplemente el resultado de una progresión aritmética sencilla. Por ejemplo, consideremos el problema de la Búsqueda Binaria: se desea buscar un elemento en una lista ordenada de datos, si el elemento dado está en la lista regresa su posición en el arreglo y si no está indica en que posición debería estar. El Listado 10 muestra una implementación de la Búsqueda Binaria usando un ciclo while.

El programa trabaja dividiendo, repetidamente, en dos el área de búsqueda hasta acorralar al elemento buscado, x, o la posición donde éste estaría. Ya que originalmente teníamos n valores, el número máximo de subdivisiones posibles es $\log n$, por lo cual el tiempo total de ejecución del Algoritmo BBinaria es $O(\log n)$.

Listado 10 Algoritmo de Búsqueda Binaria

```
BBinaria(A, x){    // A arreglo de datos; x dato a buscar.
 //  PreC: A esta ordenado ascendentemente.
 // PostC: Regresa el indice donde x fue encontrado o, si no esta,
 //        regresa el indice del siguiente elemento mayor a x.

   int  low =0              // indice inferior
   int  high=A.length       // tamanio de A

   // Va reduciendo el area de busqueda
   // hasta solo tener un elemento
    while  (low<high){
      m = (low + high) / 2;
      if (A[m]< x)  then
        low = m +1;
      else high = m;
   }
   return low;
}
```

2.3.5. Llamadas a Procesos.

Cuando ocurre una llamada el tiempo de ejecución de la llamada está asociado al tiempo de ejecución del proceso. Consideremos el problema de listar todos los números menores que un valor dado indicando si son números primos o no, nos auxiliaremos del Proceso EsPrimo del Listado 7. El Listado 11 ilustra un algoritmo para este problema.

Listado 11 Algoritmo Imprime Primos

```
ImprimePrimo(n){          // recibe un entero

   for (i=2; i<= n; i++){
   if (EsPrimo(i)) then
      println ( i, ``es primo'');
   else
      println ( i, ``no es primo'');
   }
}
```

Sabemos que la función EsPrimo requiere tiempo $O(\sqrt{n})$, el ciclo for sobre i se realiza n veces, así que el proceso ImprimePrimos requiere en total tiempo $O(n \cdot \sqrt{(n)})$.