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 5

Diseño de Algoritmos usando Inducción.

La Inducción Matemática está basada en el principio del dominó. Para estar seguros que todas las fichas caerán es necesario verificar que empujando la primera golpeará la segunda que caerá y golpeará la siguiente que caerá... No es necesario "derrumbar" todo el dominó cada vez que se agregue una nueva ficha sólo para verificar que el nuevo arreglo funcionará. El mismo principio es aplicado al diseño de algoritmos:

> No es necesario diseñar los pasos requeridos para resolver el problema desde cero, es suficiente garantizar que
>
> 1. Es posible resolver el problema para un ejemplar pequeño (Caso Base).
> 2. Una solución para cada problema puede ser construida por soluciones para problemas más pequeños (Paso Inductivo).

Con este principio en mente, ahora es posible concentrarse en reducir el problema en uno más pequeño, o en un conjunto de problemas más pequeños. Lo *malo* es que no es tan fácil encontrar una manera de reducir el problema. En esta sección se presentarán diferentes técnicas para realizar tal proceso. Se inicia con ejercicios que quiza no tengan mucha importancia sobre aplicaciones, pero si ilustran los principios que deseamos enfatizar.

5.1. Ejemplos

En esta sección presentaremos una serie de ejercicios, y su solución, que ejemplifican cómo utilizar la inducción matemática para diseñar algoritmos.

5.1.1. Evaluación de Polinomios.

Dada la secuencia de números reales $a_n, a_{n-1}, \cdots, a_1, a_0$ y un número real x, calcular el valor del polinomio:

$$P_n(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \cdots + a_1 \cdot x^1 + a_0.$$

Este problema no parece ser un natural candidato para un acercamiento a la inducción, pero se mostrará que la inducción puede llevar a una buena solución. Se inicia con el más simple y trivial acercamiento y entonces se encantraran variantes para llegar a las mejores soluciones. Nótese que el problema involucra $(n+2)$ números. La aproximación inductiva consite en resolver el problema en términos de una solución para un problema más pequeño. La estrategia ahora es reducir el problema a uno de menor tamaño, el cual, entonces, se resuelve recursivamente: por inducción.

El primer intento es reducir el problema quitando a_n, quedando el polinomio:

$$P_n(x) = a_{n-1} \cdot x^{n-1} + a_{n-2} \cdot x^{n-2} + \cdots + a_1 \cdot x^1 + a_0.$$

Es el mismo problema con menos parámetros. Entonces, se propone resolverlo usando inducción.

Hipótesis de Inducción.

Sabemos cómo evaluar un polinomio representado por la entrada $a_{n-1}, a_{n-2}, \cdots, a_1, a_0$ en el punto x. Es decir, sabemos calcular $P_{n-1}(x)$.

Caso Base.

Calcular a_0, lo cual es trivial.

Paso Inductivo.

Debemos mostrar cómo resolver el problema original, P_n, con la ayuda de la solución de un problema más pequeño, P_{n-1}. Este paso es una respuesta directa en este caso, se requiere simplemente calcular x^n multiplicada por a^n y sumarla al resultado de

$$P_{n-1}: \quad P_n(x) = P_{n-1}(x) + a^n \cdot x^n.$$

Este algoritmo es correcto pero requiere de un orden cuadrático de multiplicaciones: $[n + (n-1) + (n-2) + \cdots + 3 + 2 + 1] = n(n+1)/2$ y efectua n sumas. Así que el desempeño computacional de este algoritmo es $O(n^2)$. Hasta aquí pudiera parecer que el uso de la inducción para este problema es $frivola$ y sólo complica una solución simple.

Ahora, usaremos la inducción con una pequeña modificación para obtener una mejor solución. Nótese que las potencias de x se calculan desde cero. Es posible evitar muchas multiplicaciones usando el valor de x^{n-1} para calcualr x^n. Haciendo este cambio, podemos incluir el cálculo de x^k en la hipótesis de inducción.

Hipotesis de Inducción$^+$ Sabemos cómo calcular el valor del polinomio $P_{n-1}(x)$ y conocemos cómo calcular x^{n-1}.

Esta hipótesis es más fuerte, ya que requiere el cálculo de x^{n-1}. Pero es más fácil de extender porque ahora es fácil calcular x^n. Necesitamos efectuar únicamente una multiplicación para calcular x^n, una más para calcular $a_n \cdot x^n$ y una suma para completar el cálculo. Entonces se realizan sólo $2n$ multiplicaciones y n sumas.

Es interesante hacer notar que aunque la hipótesis de inducción requiere más cálculos, el trabajo global es menor. Este algoritmo es eficiente, simple y fácil de implantar, sin embargo aún puede optimizarse utilizando la inducción de otra forma.

Listado 17 Algoritmo Evaluación Polinomial.

```
Evaluacion\_Polinomial(A,x)
//Entradas:A=a0,a1,a2,...,an —Coeficientes
                x un numero real, punto a evaluar
//Salida: P el valor del polinomio en x.
{  P<—a_n,
   For(i=1;i<=n;i++) Do
        P<—x*P+A[n-i]
}
```

Al quitar el primer coeficiente, a_0, del polinomio queda:

$$P'_{n-1} = a_n \cdot x^{n-1} + a_{n-1} \cdot x^{n-2} + \cdots + a_2 \cdot x + a_1.$$

Nótese que a_n es ahora el $(n-1)$-ésimo coeficiente, a_{n-1} es en $(n-2)$-ésimo coeficiente y así sucesivamente. La nueva hipótesis queda:

Hipótesis de Inducción (Orden Inverso). Sabemos evaluar el polinomio representado por los coeficientes $a_n, a_{n-1}, a_{n-2}, \cdots, a_2, a_1$ en el punto x. Es decir, sabemos calcular $P'_{n-1}(x)$.

Esta hipótesis es más conveniente para *nuestros* propósitos, ya que resulta fácil de entender. Claremente, se tiene que: $P_n(x) = x \cdot P'_{n-1}(x) + a_0$. Por lo tanto, una multiplicación y una suma son requeridads para calcular $P_n(x)$ a partir de $P'_{n-1}(x)$.

El algoritmo completo puede ser descrito como:

$$P_n(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \cdots + a_1 \cdot x^1 + a_0 \cdot x$$

$$= ((...((a_n \cdot x + a_{n-1}) \cdot x + a_{n-1})...) \cdot x + a_1) \cdot x + a_0.$$

Este algoritmo es conocido como la Regla de Horner. En el Listado 17 se describe una versión del mismo. El algoritmo Evaluación_Polinomial requiere únicamente de n multiplicaciones, n sumas y una localidad adicional de memoria. Aunque las demas soluciones son muy simples y muy eficientes, este algoritmo es mejor, ya que además de ser más rápido es más simple.

Ejemplo 5.1 *Aplicación del Método de Horner.*
Evaluar el polinomio $P(x) = 4x^5 + 3x^4 - 2x^3 + 5x^2 - 6x + 7$ en el punto $x_o = 2$.

Tenemos:
$$\begin{aligned}
P(x) &= 4x^5 + 3x^4 - 2x^3 + 5x^2 - 6x + 7 \\
P(x) &= (4x^4 + 3x^3 - 2x^2 + 5x - 6)x + 7 \\
P(x) &= ((4x^3 + 3x^2 - 2x + 5)x - 6)x + 7 \\
P(x) &= (((4x^2 + 3x - 2)x + 5)x - 6)x + 7 \\
P(x) &= ((((4x + 3)x - 2)x + 5)x - 6)x + 7 \\
P(x) &= (((((4)x + 3)x - 2)x + 5)x - 6)x + 7 \\
P(x) &= (((((4)2 + 3)2 - 2)2 + 5)2 - 6)2 + 7 \quad = \quad 175.
\end{aligned}$$

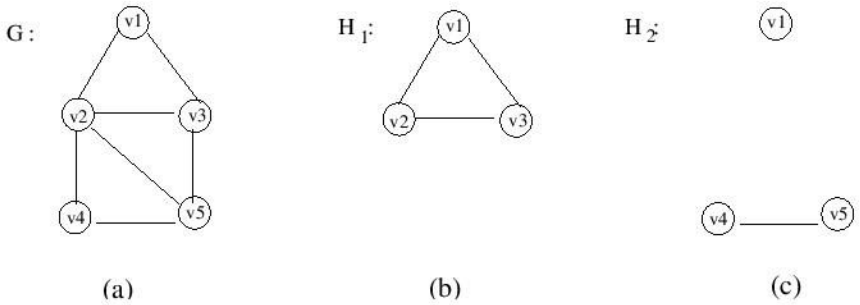


Figura 5.1: Ejemplo de SubGráfica Inducida.

5.1.2. Máxima SubGráfica Inducida.

Sea $G = (V, A)$ una gráfica no dirigida, donde V conjunto de vértices, $V \neq \emptyset$ y finito, A conjunto de aristas, $A \subseteq V \times V$. Una subgráfica inducida de G es una gráfica $H = (U, F)$ tal que $U \subseteq V$ y F es el conjunto de todas las aristas incidentes a los vértices en U.

Considere la gráfica G de la Figura 5.1(a). Las subgráfica inducidas por los conjuntos de vértices $U_1 = \{v_1, v_2, v_3\}$ y $U_2 = \{v_1, v_4, v_5\}$ se muestran en la Figura 5.1(b) y Figura 5.1(c), respectivamente.

Problema 5.1 Dada una gráfica no dirigida $G = (V, A)$ y un entero $k > 0$, encontrar una subgraáfica inducida $H = (U, E)$ de G de máximo tamaño tal que todos los vértices de H tengan grado mayor o igual que k o que concluya que tal subgráfica H no existe. El grado de un vértice v es el número de aristas adyacentes a él, se denota con $\delta(v)$.

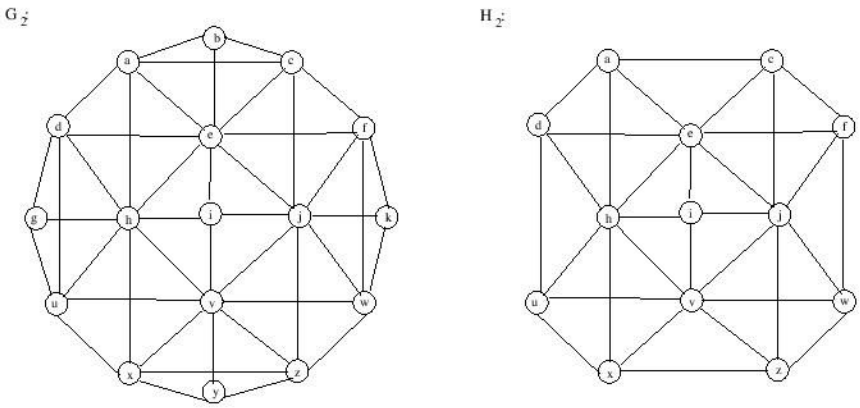


Figura 5.2: Ejemplo de Máxima SubGráfica Inducida con $\delta(v) \geq 4$

Considere la gráfica de la Figura 5.2. Se desea encontrar la máxima subgráfica H inducida de G_2, para la cual $\delta(v) = k \geq 4$, $\forall v \in V(H)$. Los vértices ω en G_2 para los cuales $\delta(\omega) < k = 45$ son: b, g, k, y. Quitando estos vértices, con sus respectivas aristas, la subgráfica inducida resulta ser maximal. La Figura 5.2 ilustra la subgráfica resultante. No es posible construir una subgráfica maximal con $k \geq 5$.

La primera idea que surgue para resolver el problema es: Quitar los vértices cuyo grado es menor que k y remover con ellos sus aristas adyacentes, *pero* los grados de los otros vértices pueden reducirse. El orden de borrado no es claro: ¿ Deberíamos remover todos los vértices v tales que $\delta(v) < k$ y entonces trabajar con los vértice que fueron reducidos ? ¿ Deberíamos quitar un vértice y continuar con los demás ?

En lugar de pensar en un algoritmo como una secuencia de pasos para obtener un resultado pensemos en probar un teorema que garantice que el algoritmo existe. La idea es *imitar* los pasos que tomamos para probar el teorema y aprovechar la percepción (intuición) del problema.

Hipótesis de Inducción. Dada una gráfica no dirigida $G = (V, A)$ y un entero $k > 0$, sabemos como encontrar la máxima subgráfica inducida por todos los vértices de v tal que $\delta(v) \leq k$ siempre que el número de vértices sea menor que n, $|V| = n$.
Demostración. Primero debemos probar que la hipótesis es válida para el caso base, después demostrar que si es verdad para $(n-1)$ resultara verdad para todo n.

Si $n \leq k$ entonces todos los grados son menores que k, lo cual resulta ser un caso trivial. El primer caso base no trivial sucede cuando $n = k + 1$, este caso ocurre cuando se tiene una gráfica completa. Asumamos entonces que $G = (V, A)$ es una gráfica con $n > k + 1$ vértices. Si todos los vértices tienen grado mayor o igual que k, entonces la gráfica completa satisface la condición.

Si exite un vértice v tal que $\delta(v) < k$, v no debe estar en la gráfica inducida que estamos buscando, entonces quitamos al vértice v con sus respectivas aristas *sin afectar* las condiciones del teorema. Después de quitar a v la gráfica queda con $(n-1)$ vértices y por Hipótesis de Induccion ya sabemos como resolver el problema.

5.1.3. Encontrando Proyecciones uno a uno.

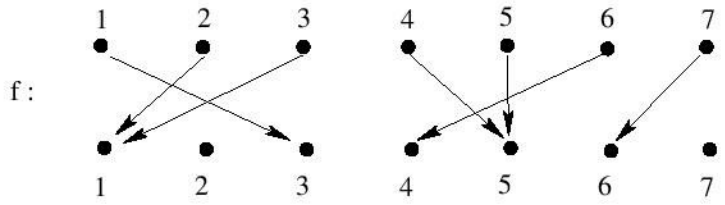


Figura 5.3: Ejemplo de una Función.

Sea $f : A \longrightarrow A$ una función que proyecta a un conjunto finito en sí mismo. Denotamos a los elementos de A por enteros de 1 a n. Asumimos que f se representa como un arreglo $f[1..n]$ donde $f[i] = f(n)$, $\forall i \in [1, n]$. La Figura 5.3 muestra un ejemplo de una función f de un conjunto de 7 elementos, $A = \{1, 2, 3, 4, 5, 6, 7\}$, en sí mismo.

Llamamos a f una función uno a uno si para cada elemento j hay al menos un elemento i que es proyectado a j. Si f es uno a uno entonces A es el conjunto Maximal, si no, existe $f(i) = f(j), p.a.\ i \neq j$. La Figura 5.4 muestra un ejemplo, donde f es uno a uno.

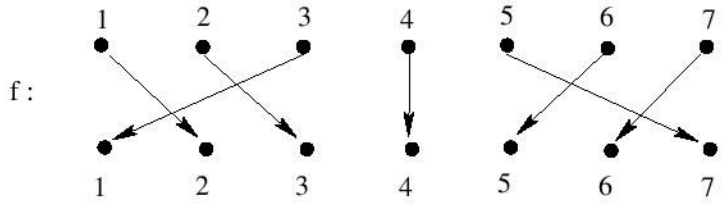


Figura 5.4: Función uno a uno.

Problema 5.2 Dado un conjunto finito a y una función $f : A \longrightarrow A$, encontrar un subconjunto $S, S \subseteq A$, con el máximo número de elementos tal que:

1. La función f proyecta cada elemento de S en otro elemento de S.
2. No existen dos elementos de S que sean proyectados a un mismo elemento, esto significa que f es uno a uno cuando se restringe a S.

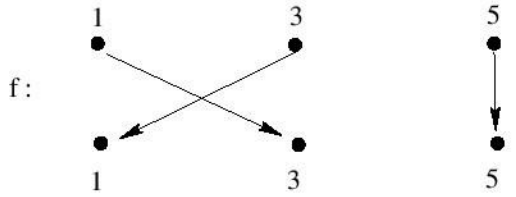


Figura 5.5: Solución $S = \{1, 3, 5\}$

Ejemplo 5.1 Considere la función descrita en la Figura 5.3.
Tenemos que $f(2) = f(3) = 1$ y $f(4) = f(5) = 5$. Si eliminamos al 3 tendríamos que quitar al 1 lo que implicaría eliminar al 2. Estas operaciones no nos llevan a contruir un conjunto maximal. En cambio, si eliminamos al 2 sólo tenemos que quitar al 2 y no a 1, 2, y 3. Después podemos eliminar al 4, lo que implica quitar al 6 y al 7. De esta forma, el conjunto maximal es: $S = \{1, 3, 5\}$. La solución se ilustra en la Figura 5.5.

A continuación se da una técnica para diseñar un algoritmo, usando inducción matemática, que solucione el problema de encontrar la función uno a uno.

Hipótesis de Inducción.

Sabemos resolver el problema para conjuntos de $(n-1)$ elementos.

Caso Base. Si $n = 1 \Rightarrow A = \{a\} \Rightarrow f(a) = a \Rightarrow S = \{a\}$.

Paso Inductivo.

Supongamos ahora que tenemos un conjunto e n elementos y buscaremos S que cumpla con las condiciones del problema.

Cualquier elemento i de S tiene su imagen en otro elemento de S. Sea $|S| = k$. Si $\exists\ i \in S$ tal que $f(j) \neq i,\ \forall j \in S$, entonces, los k elementos de S están proyectados en a lo más $(k-1)$ elementos. Lo que implica que f no es uno a uno. Por lo tanto, debemos quitar a i de S.

Listado 18 Algoritmo de Proyección.

```
Proyecta121(f,n){
//Entrada. f:Arreglo con n enteros entre 1 y n.
   n:Tamanio del arreglo.
//Salida. S: Subconjunto de enteros de 1 a n tal que
  f es una funcion uno a uno en S.

  S<—A;  // A numeros de 1 a n

  For(j=1;j<=n;j++) do     // c[j] es el numero de
    c[j]<--0;              //elementos proyectados a j

  For(j=1;j<=n;j++) do c[f(j)]++;

  For(j=1;j<=n;j++) do
     If(c[f(j)]=0) Then Q.Mete(j);

     While not Q.Empty Do{
        i <— Q.Saca;
        S <— S\{i};
        c[f(i)]--
        Si c[f(i)]=0 Then
           Q.Mete(f[i])
     }//while

}
```

Ahora tenemos el conjunto $A' = A \setminus \{i\}$ con $n-1$ elementos y f es la función, $f : A' \longrightarrow A'$. Por hipótesis de inducción, sabemos como resolver el problema para un conjunto A' con $(n-1)$ elementos. Si no existe tal elemento i, entonces la función es uno a uno y no hay nada por demostrar. Nótese que el truco fue eliminar al elemento i.

En el Listado 18 se presenta una versión de un algoritmo que soluciona el problema de encontrar una función uno a uno.

5.1.4. El Problema de la Celebridad.

El siguiente ejemplo es un clásico en el diseño de algoritmos. Es un bonito ejemplo que tiene una solución que no requiere revisar *todos* los datos, o una significante parte de ellos.

De entre n personas, se define una **celebridad** como alguien quien es conocido por todos pero que no conoce a nadie. El problema consiste en identificar a la celebridad, si es que existe, haciendo preguntas de la forma *¿Disculpe, usted conoce a aquella persona?* Se asume que todas las respuestas son verdaderas y que también la celebridad debe contestar. El objetivo es minimizar el número de preguntas. Ya que hay $n(n-1)/2$ parejas de personas, hay potencialmente la necesidad de hacer $n(n-1)$ preguntas, en el peor caso, si las preguntas son contestadas de manera arbitraria.

Podemos formular del problema usando Teoría de Gráficas y construir una gráfica dirigida donde los vértices correspondan a las personas y una arista ira de la persona A a la persona B si A conoce a B. La Figura 5.6 muestra tal representación. Bajo este modelo,

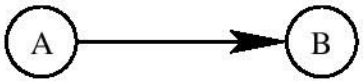
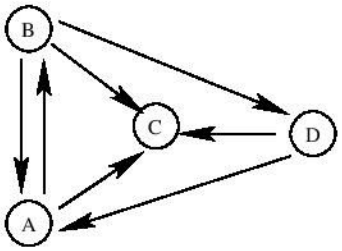


Figura 5.6: A conoce a B

una celebridad corresponde a un *sumidero o pozo* de la gráfica. Un sumidero es un vértice con ingrado $n-1$ y exgrado[1] 0. Note que una gráfica puede tener a lo más un sumidero. La entrada del problema corresponde a una matriz de adyacencia de orden $n \times n$, donde la entrada ij es 1 si la $i-$ésima persona conoce a la $j-$ésima persona y 0 en otro caso.

La Figura 5.7 muestra un ejemplo de este problema para $n = 4$. En (a) se muestra la gráfica correspondiente a la relación entre las personas y en (b) se presenta la matriz de adyacencia correspondiente. En este ejemplo, C representa a la celebridad del grupo dado.



(a)

	A	B	C	D
A	1	1	1	0
B	1	1	1	1
C	0	0	1	0
D	1	0	1	1

(b)

Figura 5.7: Un Ejemplar del Problema de la Celebridad.

Problema 5.3 Dada una matriz de adyacencia de orden $n \times n$, determinar si existe una i tal que todas las entradas en la $i-$ésima columna, excepto por la entrada ii, son 1 y todas las entradas en el $i-$ésimo renglón, excepto por la entrada ii, son 0.

Caso Base: $n = 2$, resulta trivial.
Consideremos, como es usual, la diferencia entre los problemas con $n-1$ personas y con n personas.
Hipótesis de Inducción.

Podemos encontrar la celebridad entre las primeras $n-1$ personas.

Paso Inductivo.

Asumimos que podemos encontrar la celebridad entre las primeras $n-1$ personas, por inducción. En virtud de que que existe a lo más una celebridad, existen tres posibilidades a considerar: Caso 1. La celebridad está entre las primeras $n-1$ personas; Caso 2. La celebridad es la $n-$ésima persona; y Caso 3. No hay celebridad.

El primer caso es fácil de manejar. Sólo necesitamos revisar que la $n-$ésima persona conozca a la celebridad y que la celebridad no conozca a esta persona.

[1] informalmente, el ingrado es el número de flechas que llegan a un vértice y el exgrado es el número de flechas que salen del vértice

Los otros dos casos son más difíciles, pues para determinar si la $n-$ésima persona es la celebridad, necesitamos realizar $2(n-1)$ preguntas. Si realizamos $2(n-1)$ preguntas en el $n-$ésimo paso, el número total de preguntas será $n(n-1)$, lo cual queremos evitar. Así que requerimos de otra estrategia.

El truco ahora es considerar el problema *al revés*. Puede ser díficil identificar una celebridad, pero es probablemente más fácil identificar a alguien que no es celebridad. Después de todo, hay, definitivamente más no-celebridades que celebridades. Si eliminamos a alguien como candidato a celebridad, entonces estaremos reduciendo el tamaño del problema de n a $n-1$. Además, no es necesario eliminar a alguien en especifico, cualquiera puede ser eliminado.

Supongamos que le preguntamos a A si conoce a B, entonces resolvemos el problema para las otras $n-1$ personas. Estamos garantizando que el Caso 2 no ocurrirá, ya que la persona eliminada no puede ser la celebridad. Además si el Caso 3 sucede, entonces no existe una celebridad entre las n personas. Únicamente queda el Caso 1, pero éste es fácil: Si la celebridad está entre las $n-1$ personas, nos tomará dos preguntas más para verificar que ésta es una celebridad del conjunto completo de n personas; de otra forma no hay celebridad.

El algoritmo trabaja de la siguiente manera: Preguntamos a A si conoce a B y eliminamos, ya sea a A o a B, de acuerdo con la respuesta. Si A contesta sí, entonces A no es celebridad, podemos eliminarle; en cambio si la respuesta de A es no entonces B no es la celebridad y podemos eliminar a B.

Supongamos que eliminamos a A, entonces encontramos por inducción una celebridad entre las $n-1$ personas restantes. Si no hay celebridad el algoritmo termina; en otro caso, verifica que A conoce a la celebridad y que la celebridad no conoce a A.

Implementación. Es más eficiente implementar el algoritmo en forma iterativa, aunque haya sido ideado recursivamente. El algoritmo se divide en dos fases. En la primera, eliminamos a todos menos a un candidato y en la segunda verificamos si el candidato es en efecto la celebridad.

Empezamos con n candidatos, supongamos que estos son almacenados en una pila, *stack*, sólo es para facilitar la explicación. Para cada par de candidatos eliminamos uno realizando una pregunta -si uno de ellos conoce al otro. Iniciamos tomando a los dos primeros candidatos de la pila y eliminamos uno de ellos. Entonces en cada paso, tenemos un candidato *sobreviviente* y continuamos de esta manera hasta terminar con los elementos de la pila. Al final, cuando la pila queda vacía, un candidato perdura. Entonces verificamos si tal candidato es en efecto la celebridad. En el Listado 19 se presenta una versión de un algoritmo que soluciona el problema. Nótese que para la implementación no es necesario usar una pila, nos bastan los índices i, j y s.

Listado 19 Algoritmo de la Celebridad.

```
Celebridad(P) {
//  P: Matriz de nxn, con valores logicos 0 y 1.
     i <-- 1;                    // primer candidato
     j <-- 2;                    // segundo candidato
     s <-- 3;                    // siguiente candidato

    // Primero eliminamos a todos, dejando un candidato
     While  ( s <= n+1 )   Do  {
       If   (P[i,j])   Then  i  <-- s;
         Else j <-- s;
       s <-- s + 1;
     }                           // i o j fueron eliminados

     If   (i=n+1)  Then   candidato  <-- j;
      Else                candidato  <-- i;

    // Ahora Verificamos que sea la celebridad

    malo <-- False;          // candidato malo
      k  <-- 1;              // contador de personas
                             // condicion para pasar la prueba:
    P[candidato, candidato] <-- false;

    While  (not malo && k<=n) Do {
      If  (P[candidato, k])   Then   malo <-- True;
      If  (not P[k,candidato])  Then
        If  (candidato != k)  Then   malo <-- True;
      k <-- k + 1;
    }

    If  (not malo)
      Then  celebridad <-- candidato;
      Else  celebridad <-- 0;          // No hay celebridad.

    Return celebridad
}
```

Complejidad. Se realizan a lo más $3(n-1)$ preguntas: $n-1$ preguntas en la primera fase para eliminar a $n-1$ personas y se realizan, a lo más, $2(n-1)$ preguntas para verificar si el candidato es o no la celebridad.

Nótese que el tamaño del ejemplar no es n, el número de personas, sino el número de entradas en la matriz. La solución muestra que es posible identificar a la celebridad revisando únicamente del $O(n)$ entradas en la matriz de adyacencia, aun cuando la solución pueda ser tentativa a cada una de las $O(n^2)$ entradas.

5.1.5. Factor de Balance en un Árbol Binario.

Sea T un árbol binario con raíz r. La altura de un nodo v es la distancia entre v y la hoja más profunda del árbol. El *Factor de Balance* de un nodo v está definido como la diferencia entre la altura del nodo del subárbol izquierdo, HijoIzq, y la altura del nodo del subárbol derecho, HijoDer. Se asume que las hojas de los nodos están etiquetadas de izquierda a derecha.

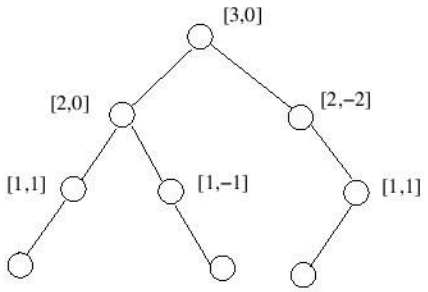


Figura 5.8: Ejemplo de Factor de Balance en un Árbol Binario...

Problema 5.4 Dado un árbol binario T con n nodos y un nodo distinguido r, llamado raíz r, calcular el factor de balance de cada nodo.

Hipótesis de Inducción. Sabemos cómo calcular el factor de balance de todos los nodos en un árbol con menos de n nodos.
Caso Base.
$n = 1$, árboles con un nodo: caso trivial.
Paso Inductivo.
Dado un árbol con $n > 1$ nodos quitamos la raíz, entonces resolvemos el problema por inducción para los dos subárboles restantes. Elegimos quitar la raíz porque el factor de balance de un nodo depende únicamente de los nodos *inferiores*.

Conocemos el factor de balance de todos los nodos, excepto de la raíz. El factor de balance de la raíz *no* depende del factor de balance de las raices de sus hijos, si no de la altura de los hijos. Por lo tanto, esta inducción no funciona del todo bien, necesitamos saber la altura de los hijos.

Hipótesis de Inducción$^+$.
Sabemos calcular el factor de balance y la altura de todos los nodos en los árboles que tienen menos de n nodos.
Caso Base. $n = 1$, árboles con un nodo: caso trivial.
Paso Inductivo. Cuando consideramos la raíz podemos determinar su factor de balance fácilmente calculando la diferencia entre las alturas de sus hijos, las cuales ya conocemos. Podemos determinar la altura, h, de la raíz r:

$$h(r) = 1 + \text{máx}\{h(HijoDer), h(HijoIzq)\}$$

5.1.6. Máxima Subsecuencia Consecutiva.

Problema 5.5 Dada una secuencia $x_1, x_2, \ldots, x_n$ de números reales, no necesariamente positivos encontrar la subsecuencia de elementos consecutivos $x_i, x_{i+1}, \ldots, x_j$ tal que la suma de los números de la subsecuencia sea la máxima entre todas las subsecuencias de elementos consecutivos.

Ejemplo. Sea $S =< 2, -3, 1{,}5, -1, 3, -2, -3, 3 >$ la máxima subsecuencias de elementos consecutivos es $S_M =< 1{,}5, -1, 3 >$ su suma es 3.5.

Hipótesis de Inducción. Sabemos como encontrar la máxima subsecuencia en secuencias de tamaño menor que n.

Caso Base.

$n = 1$, sea $S = \{e\} \Rightarrow S_M = \{e\}$ si $e > 0$ y $S_M = \emptyset$ si $e \leq 0 \Rightarrow$ suma es cero.

Paso Inductivo.

Considere $S =< x_1, x_2, \ldots, x_n >$, de tamaño $n > 1$. Por hipótesis de inducción, sabemos encontrar S'_M en la secuencia $S' =< x_1, x_2, \ldots, x_{n-1} >$. Si la subsecuencia $S'_M = \emptyset \Rightarrow x_i < 0, \ \forall i$. Consideramos, entonces, solo al elemento x_n, lo que implica resolver el caso base.

Sea $S'_M =< x_i, x_{i+1}, \ldots, x_j >, 1 \leq i \leq j \leq (n-1)$ la subsecuencia encontrada por inducción en S'. Si $j = n-1$, es fácil extender la solución a S. Pues si el elemento $x_n \geq 0$, entonces $S'_M = S'_M \cup \{x_n\}$. En otro caso, S'_M no cambia y continua siendo máxima.

Sin embargo, si $j < n-1$ suceden dos casos: (1) S'_M sigue siendo máxima, o (2) existe otra subsecuencia que no es máxima en S' pero es máxima en S cuando se agrega el elemento x_n.

La idea clave aquí es usar **Inducción Reforzada**. Primero ilustraremos con este ejemplo la técnica y después la formalizaremos en la siguiente subsección.

El problema que tuvimos con la inducción directa es que x_n puede extender una subsecuencia que no es máxima en S', pero con la cual podemos crear una nueva subsecuencia máxima. Conocer sólo la subsecuencia S' no resulta suficiente. Sin embargo, x_n puede extender únicamente una subsecuencia que termine en $n-1$, es decir, un sufijo de S'. Suponga que fortalecemos la hipótesis de inducción incluyendo el conocimiento del máximo sufijo, denotado por $S'_E = (x_k, x_{k+1}, ..., x_{n-1})$.

Hipótesis de Inducción$^+$.

Sabemos cómo encontrar, en subsecuencias de tamaño menor que n, una subsecuencia máxima y la subsecuencia máxima que es un sufijo.

Si conocemos ambas subsecuencias, el algoritmo se vuelve claro. Agregamos x_n al máximo sufijo. Si la suma es mayor a la de la subsecuencia máxima, entonces tenemos una nueva subsecuencia máxima y también un nuevo sufijo. En otro caso, conservamos la subsecuencia máxima anterior, pero necesitamos encontrar el nuevo sufijo máximo. No podemos sólo agregar x_n al máximo sufijo previo, podría ser que el máximo sufijo que termine en x_n sea negativo. En este caso, es mejor tomar el conjunto vacío como el máximo sufijo. El Listado 20 presenta un algoritmo para solucionar este problema.

5.1.7. Inducción Reforzada

En esta sub-sección presentaremos de manera breve la Inducción Reforzada, la cual es una de las más importantes y poderosas técnicas para demostrar teoremas en matemáticas usando inducción.

Listado 20 Algoritmo Máxima Subsecuencia Consecutiva.

```
MaxSubSC(X,n) {
//  X: Arreglo de tama~no n
     GMax <— 0;                    // suma global
     SMax <— 0;                    // suma del sufijo

     For  (i=1; i<=n; i++)  Do {
       If    (X[i]+ SMax > GMax)    Then {
         SMax <— SMax + X[i];
         GMax <— SMax;
       }
        Else
          If    (X[i]+ SMax > 0)    Then
                SMax <— SMax + X[i];
          Else  SMax <— 0;
     }
    Return SMax
}
```

Denotemos el teorema o propiedad a demostrar por P. La hipótesis de inducción puede ser denotada por $P(< n)$ y la prueba debe concluir que $P(< n) \Rightarrow P(n)$.

En muchos casos, podemos añadir otra suposición, digamos Q, con la cual la prueba se vuelve más fácil. Es decir, es más fácil probar

$$[P \text{ y } Q](< n) \Rightarrow P(n)$$

a probar únicamente que

$$P(< n) \Rightarrow P(n).$$

La expresión $[P \text{ y } Q]$ es un teorema, o propiedad, más fuerte que sólo P y frecuentemente los teoremas más fuertes resultan más faciles de probar. Este proceso puede ser repetido y con la adición de suposiciones la prueba se transforma en una demostración tratable.

5.2. Ejercicios

Ejercicio 5.1 Diseñe un algoritmo para convertir un número binario a un numero decimal. La entrada es un arreglo de bits b de tamaño k, y la salida es un número n. *Justifique el algoritmo usando un invariante del ciclo y determine la complejidad del mismo.*

Ejercicio 5.2 Diseñe un algoritmo que convierta un número en base 6, a binario. La entrada es un arreglo de dígitos en base 6, y la salida es un arreglo de bits. *Probar que el algoritmo es correcto usando invariantes del ciclo.*

Ejercicio 5.3 Sea $x_1, x_2, ..., x_n$ una subsecuencia de números reales, no necesariamente positivos. Diseñe un algoritmo de orden $O(n)$ que encuentre la subsecuencia de elementos consecutivos $x_i, x_{i+1}, ..., x_j$ tal que el producto de los números en ella sea el máximo sobre todas las subsecuencias consecutivas. El producto de una subsecuencia vacía se define como 1. *Justifique su algoritmo y su complejidad.*

Ejercicio 5.4 Sea $x_1, x_2, ..., x_n$ una subsecuencia de números reales, no necesariamente positivos. Diseñe un algoritmo que encuentre la subsecuencia de elementos consecutivos $x_i, x_{i+1}, ..., x_j$ tal que la suma de los números en ella sea la máxima sobre todas las subsecuencias consecutivas. *Justifique su algoritmo y su complejidad.*

Ejercicio 5.5 Sea $x_1, x_2, ..., x_n$ una subsecuencia de números reales, no necesariamente positivos. Diseñe un algoritmo que encuentre la subsecuencia de elementos consecutivos $x_i, x_{i+1}, ..., x_j$ tal que la suma de los números en ella sea la mínima sobre todas las subsecuencias consecutivas. *Justifique su algoritmo.*

Ejercicio 5.6 Diseñe un algoritmo para el siguiente problema, justifíquelo y determine su complejidad. Dada una lista de n enteros positivos y distintos, particionar la lista en dos sublistas, cada una de tamaño $n/2$, tal que la diferencia entre las sumas de los enteros entre las dos sublistas sea máxima. Puede suponer que n es múltiplo de 2.

Ejercicio 5.7 Diseñe un algoritmo para el siguiente problema, justifíquelo y determine su complejidad. Dada una lista de n enteros positivos y distintos, particionar la lista en dos sublistas, cada una de tamaño $n/2$, tal que la diferencia entre las sumas de los enteros entre las dos sublistas sea mínima. Puede suponer que n es múltiplo de 2.

Ejercicio 5.8 Sea $T = (V, A)$ un árbol binario con n vértices. Queremos construir una matriz M de $n \times n$ tal que el elemento $M[i, j]$ sea igual a la distancia entre los vértices v_i y v_j. Diseñe un algoritmo de orden $O(n^2)$ que construya tal matriz. Suponga que el árbol T está representado por su lista de adyacencias. *Justifique su algoritmo*

Bibliografía

[1] **K.A. Berman & J.L. Paul**, *Sequential and Parallel Algorithms*, PWS Publish Co, USA, 1986.

[2] **G. Brassard & P. Bratley**, *Fundamentals of Algorithmics.*, Prentice Hall International, USA, 1996.

[3] **T. Budd**, *Classic Data Structures in Java*, Addison Wesley Co, 2001.

[4] **G. Chartran y O. R. Oellermannd,** *Applied and Algorithmic Graph Theory*, McGraw-Hill, Inc., Singapore, 1993.

[5] **T.H. Cormen, E. Leiserson & R.L. Rivest,** *Introduction to Algorithms.*, McGraw-Hill, Inc., Singapore, 1990.

[6] **W.J. Collins**, *Data Structures. An Object Oriented Approach*, Addison Wesley Co, 1992.

[7] **B. Flamig,** *Practical Algorithms in C++* , Coriolis Group Book, USA 1995

[8] **R.W. Floyd**, *Assigning meaning to programs*, Symposium on Applied Mathematics, AMS, 1967, pp 19-32.

[9] **D. Harel**, *Algorithmics. The Spirit of Computing*, Addison Wesley Publishing Co. USA, 1992.

[10] **C. A. R. Hoare**, *An Axiomatic Basis for Computer Programming*, Comm. of the ACM, No. 12 (1969) pp. 576-583.

[11] **D. E. Knuth**,
The Art of Computer Programing, Vol. I: Fundamental Algorithms,
Addison Wesley , 1973.

[12] **J. Kingston**,
Algorithms and Data Structures: Design, Correctness and Analysis,
Addison Wesley Co., Australia, 1990.

[13] **U. Manber**,
Introduction to Algorithms. A Creative Approach,
Addison-Wesley Publishing Co., USA, 1989.

[14] **J. J. Martin**,
Data Types and Data Structures,
Prentice Hall International, USA, 1986.

[15] **K. Mehlhorn**,
Data Structures and Algorithms. Vol I Sorting and Searching,
Springer-Verlag, Monographs on Theorical Computer Science, Berlin, 1984.

[16] **R. Neapolitan & K. Naimipour**,
Foundations of Algorithms,
D.C. Heath and Company. USA, 1996.

[17] **G.J.E. Rawlins**,
Compared to What? An Introduction to the Analysis of Algorithms,
Computer Science Press, USA 1991.

[18] **S.S. Skeina**, *The Algorithms Design Manual*,
Springer Verlag, USA 1998.

[19] **J.A. Storer**, *An Introduction to Data Structures and Algorithms* ,
Springer, USA 2002.

[20] **R. E. Tarjan**,
Complexity of Combinatorial Algorithms,
SIAM Review, No. 3 (1978) pp. 457-491.

[21] **R. E. Tarjan**,
Worst-Case Analysis of Set Union Algorithms,
Journal of ACM, Vol. 31, No. 2, 1984, p.p. 245-281.

[22] **M.S. Weiss**, *Data Structures and Algorithm Analysis in C++* ,
The Benjamin/Cummings Pu. Co. Inc., USA 1994.

[23] **M.S. Weiss**, *Data Structures and Problem Solving in Java* ,
Addison Wesley, USA 1998.