Instituto Tecnológico de Costa Rica
Unidad de Computación

Análisis de complejidad algorítmica

Kevin Walsh Muñoz
Jonathan Rojas Vargas

Sede San Carlos
1/10/2014

Tabla de contenidos

Introducción

Este proyecto se efectuará con el propósito de evaluar el rendimiento de tres algoritmos diferentes solucionando un mismo problema. En este caso el tienen que ser capaces de buscar la ruta más corta que hay entre el primer y el último vértice de un grafo.

Este tipo de conocimiento es muy valioso actualmente en el mundo de la programación debido a que existe una gran necesidad de contar con los algoritmos más eficientes y así gozar de todas sus ventajas como por ejemplo:

- Reducción de tiempos.
- Adecuado aprovechamiento de recursos.
- Disminución de costos.

Por estos motivos es necesario que siempre se tenga la capacidad de poseer los mejores códigos y mediante las mediciones analíticas y empericas son buenas formas de identificar buenos algoritmos.

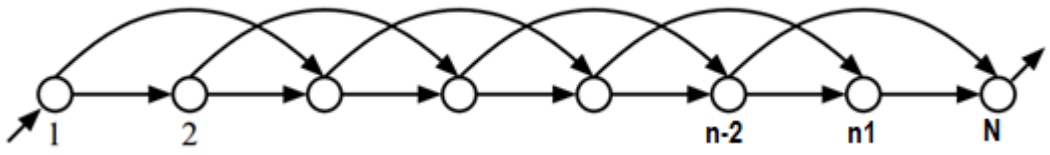
Análisis del problema

Para realizar de forma exitosa este proyecto será necesario la creación de tres algoritmos que sean capaces de encontrar la ruta más corta entre el vértice inicial y final de un grafo. De los cuales dos tienen que ser de forma iterativa y uno de forma recursiva, a estos códigos se les deberá calcular su complejidad temporal, además del número de comparaciones, asignaciones y líneas de código ejecutadas cuando el programa finaliza con una cantidad dada de vértices, también se tiene que llevar el control de la cantidad de líneas que poseen dichos algoritmos.

Pero todos estos algoritmos necesitaran de un grafo sobre el cual puedan realizar sus cálculos. Por este motivo se requiere la implementación de un grafo que cumpla con las siguientes restricciones:

- Cada vértice tendrá solo dos caminos uno que apunte al vértice siguiente y el otro al vértice siguiente de su siguiente vértice.
- Cada arco contará con su propia distancia y serán no dirigidos.

Ilustración 1



Ejemplo de la estructura del grafo

Todos los algoritmos anteriormente mencionados deberán ser implementados en el lenguaje de programación C# con sus respectivas mediciones en un documento de texto.

Solución del problema

Para solucionar de forma exitosa los problemas anteriormente citados se empleará C# como lenguaje de programación.

Como primer objetivo se creará el grafo el cual contará con un número de vértices indicado por el usuario con su respectivo nombre y arcos en los cuales se deberá de indicar su peso o distancia. Cuando el grafo esté finalizado será necesario interpretarlo y representarlo en una matriz para que pueda ser manipulado por los algoritmos que serán mencionados a continuación.

El primer algoritmo que busque la ruta más eficiente en implementarse será el recursivo. Debido a que el grafo cuenta con dos arcos por cada vértice se deberán de realizar dos llamadas recursivas por cada vértice, excepto en el penúltimo que solo requiere de una y en el último que no tiene ninguna. Estas dos excepciones serán dos de las tres formas de detener la recursividad y la tercera será cuando la distancia de una ruta sea mayor que una ruta calculada previamente utilizando así la poda.

Para realizar los códigos iterativos de optimización de ruta nos basaremos en dos algoritmos existentes como lo son:

- Algoritmo de Dijkstra.
- Algoritmo de Floyd.

Ambos códigos son muy reconocidos y se adaptan perfectamente a nuestras necesidades. Para que estos puedan funcionar de manera adecuada será necesario tener la matriz que se mencionó anteriormente con los datos de las distancias entre cada vértice del grafo.

Para el algoritmo de Dijkstra se deberán de crear una matriz en la que se guardarán las distancias más cortas entre cada vértice y dos arreglos en los que se almacenará el camino más corto y el vértice predecesor de un camino mínimo que se tiene calculado. Este código trabaja por etapas y toma en cada etapa la mejor sin considerar lo que pasará más adelante.

Mientras tanto para el algoritmo de Floyd solo necesitará de una matriz en la que se realizaran los diferentes cálculos. A continuación se mostrará un pseudocódigo de este algoritmo.

Ilustración 2

```
Floyd-Warshall (G)
n=|V [G]|
for (int i=1; i<=numeroNodos; i++)
  for (int j=1; j<=numeroNodos; j++)
      si Hay conexión   MatrizdePeso[i][j]=peso;
          else  MatrizdePeso[i][j]=infinito;
        MatrizNodoIntermedio[i][j]=j;
  Si i=j
      MatrizNodoIntermedio[i][j]=0;
      MatrizdePeso[i][j]=0;

for(int k=1;k<=numeroNodos;k++)
  for (int i=1;i<=numeroNodos;i++)
    for (int j=1;j<=numeroNodos;j++)
          a=MatrizdePeso[i][k]+MatrizdePeso[k][j];
       if(a<MatrizdePeso[i][j])
     {
     MatrizdePeso[i][j]=a;
     MatrizNodoIntermedio[i][j]=k;
     }
return MatrizdePeso, MatrizNodoIntermedio;
```

Pseudocódigo Floyd

Cuando se tengan todos los algoritmos implementados se continuará realizando los correspondientes cálculos de comparaciones, asignaciones y tiempo de ejecución a cada código.

Análisis de resultados

Medición de algoritmos

Medición analítica Dijkstra

Código algoritmo método Dijkstra:

Código fuente	Medición de líneas ejecutadas en el peor de los casos (línea por línea)
`public void solucionDijkstra( )`	
`{`	
`int minValor = Int32.MaxValue;`	N
`cantLineasD += 2;`	N
`int minNodo = 0;` `asignacionesD += 2;`	
`for( int i = 0; i < rango; i++)`	$N^2 + 2$
`{`	$N^2 + 1$
`if (C[i] == 99999)`	
`{`	
`cantLineasD++;`	N
`continue;`	
`}`	
`if(D[i] > 0 && D[i] < minValor)`	$N^2 + 1$
`{`	
`minValor = D[i];` `cantLineasD += 2;`	N
`minNodo = i;` `asignacionesD += 2;`	N
`}` `comparacionesD += 4; asignacionesD++; cantLineasD += 3;`	
`}`	
`C[minNodo] = 99999;` `asignacionesD++; cantLineasD += 2;`	N
`for( int i = 0; i < rango; i++)`	$N^2 + 2$
`{`	
`if (L[minNodo, i] < 0) // si no existe arco`	$N^2 + 1$
`{`	
`asignacionesD++;`	
`continue;`	N
`}`	

if(D[i] < 0) // si no hay un peso asignado
{
D[i] = minValor + L[minNodo, i];
asignacionesD += 2;
continue;
}
 	
if ((D[minNodo] + L[minNodo, i]) < D[i])
{
D[i] = minValor + L[minNodo, i];
asignacionesD++;
} comparacionesD += 4;
asignacionesD++; cantLineasD += 5;
}
}
 	
// Función de implementación del algoritmo
 	
public void correrDijkstra( )
{
for(trango = 1; trango < rango; trango++)
{
solucionDijkstra();
comparacionesD++; asignacionesD++; cantLineasD += 3;
}
}
	N^2 + 1
	
	N
	
	N
	
	N^2 + 1
	
	N
	
	N + 2
	
	N

Total	$7n^2$+ 11n + 11
Clasificación en notación O	Log(n)

Medición empírica Dijkstra

Código algoritmo Dijkstra:

Operaciones	Tamaños datos de entrada en cantidad de vértices.			
	10	100	1000	10000
Asig	209	15618	1506336	150063653
Comp	549	59499	5994999	599949999
Cantidad de líneas ejecutadas	1074	100744	10007594	1000076224
Tiempo de ejecución	1 ms	2 ms	53 ms	4241 ms
cantidad de líneas del	67	67	67	67

Determinar el factor de crecimiento

	100/10	1000/100	10000/1000
Factor de talla	10	10	10
Asig	74,7272	96,4487	99,6216
Comp	108,3770	100,7579	100,0750
Cantidad de líneas ejecutadas	93,8026	99,3368	99,9317
Tiempo de ejecución	2 ms	26,5 ms	80,0188 ms

Cantidad de Vértices con Dijkstra: 10.

```
La solucion de la ruta mas corta en Dijkstra es : 23

Asignaciones Dijkstra: 209
Comparaciones Dijkstra: 549

Tiempo ejecucion: 0 Milisegundos

Cantidad de lineas ejecutadas: 1074
```

Cantidad de Vértices con Dijkstra: 100.

```
La solucion de la ruta mas corta en Dijkstra es : 502

Asignaciones Dijkstra: 15618
Comparaciones Dijkstra: 59499

Tiempo ejecucion: 2 Milisegundos

Cantidad de lineas ejecutadas: 100744
```

Cantidad de Vértices con Dijkstra: 1000.

```
La solucion de la ruta mas corta en Dijkstra es : 4759

Asignaciones Dijkstra: 1506336
Comparaciones Dijkstra: 5994999

Tiempo ejecucion: 53 Milisegundos

Cantidad de lineas ejecutadas: 10007594
```

Cantidad de Vértices con Dijkstra: 10000.

```
La solucion de la ruta mas corta en Dijkstra es : 47191

Asignaciones Dijkstra: 150063653
Comparaciones Dijkstra: 599949999

Tiempo ejecucion: 4241 Milisegundos

Cantidad de lineas ejecutadas: 1000076224
```

Medición analítica Floyd

Código algoritmo método Floyd:

Código fuente	Medición de líneas ejecutadas en el peor de los casos (línea por línea)
public int[,] correrfloyd(int[,] Matrix)	
{	1
Stopwatch tiempo;	1
tiempo = Stopwatch.StartNew();	1
int N = Matrix.GetLength(0);	
cantLineasF += 3;	1
dist = Matrix;	1
asignacionesF += 2;	
int i, j, k;	
for (k = 0; k < N; k++)	N + 2
{	
cantLineasF += 2;	N^2+2
for (i = 0; i < N; i++)	
{	
cantLineasF += 2;	N^3+2
for (j = 0; j < N; j++)	
{	
cantLineasF += 3;	N^3+ 1
if (dist[i, j] != 0 && dist[i, k] != 0 && dist[k, j] != 0)	
{	
cantLineasF++;	
dist[i, j] = min(dist[i, j], dist[i, k] + dist[k, j]);	N
asignacionesF++;	
}	
comparacionesF += 2; asignacionesF++;	
}	
comparacionesF++; asignacionesF++;	
}	
comparacionesF++; asignacionesF++;	
}	
cantLineasF++;	1
return dist;	
}	
public int min(int a, int b)	
{	
comparacionesF++; cantLineasF++;	N + 1
if (a < b)	
{	1
cantLineasF++;	

return a;
}
else
{
cantLineasF
+= 2;
return b;
}
}
	1

Total	$2N^3 + N^2$ + 3N + 16
Clasificación en notación O	Log(n)

Medición empírica Floyd

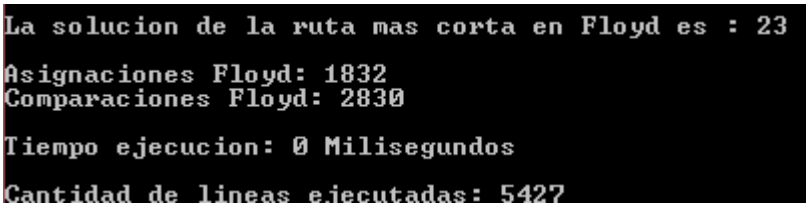
Código algoritmo Floyd:

	Tamaños datos de entrada en cantidad		
Operaciones	10	100	1000
Asig	1832	1980302	1998003002
Comp	2830	2980300	2998003000
Cantidad de líneas ejecutadas	5427	5937450	1698714697
Tiempo de ejecución	1 ms	82 ms	67042 ms
cantidad de líneas del	35	35	35

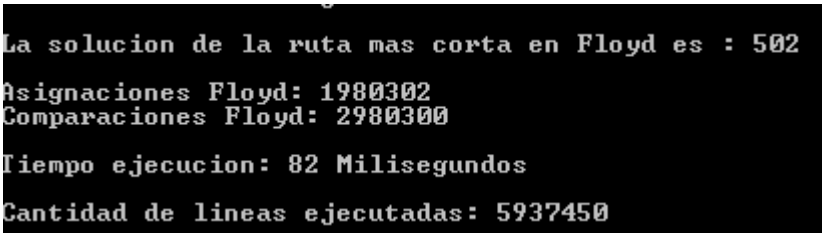
Determinar el factor de crecimiento

	100/10	1000/100
Factor de talla	10	10
Asig	1080,9508	1008,9385
Comp	1053,1095	1005,9400
Cantidad de líneas ejecutadas	1094,0574	286,1017
Tiempo de ejecución	107 ms	773,8317 ms

Cantidad de Vértices con Floyd: 10.



Cantidad de Vértices con Floyd: 100.



Cantidad de Vértices con Floyd: 1000.

```
La solucion de la ruta mas corta en Floyd es : 4759

Asignaciones Floyd: 1998003002
Comparaciones Floyd: 2998003000

Tiempo ejecucion: 67042 Milisegundos

Cantidad de lineas ejecutadas: 1698714697
```

Medición analítica método Recursivo

Código algoritmo método recursivo:

Código fuente	Medición de líneas ejecutadas en el peor de los casos (línea por línea)
public void rutaCorta(vertice tempVertice, int distancia, string ruta1)	
{	
string ruta2 = ruta1 + tempVertice.numero + ", ";	1
cantLineasR++; comparacionesR++;	N + 1
if (distancia > distanciaCorta)	
{	
cantLineasR++;	
return;	
}	
cantLineasR++; comparacionesR++;	
if (tempVertice.sigV == null)	N + 1
{	
ruta = ruta2;	1
distanciaCorta = distancia;	1
asignacionesR++;	
return;	
}	
cantLineasR++; comparacionesR++;	
if (tempVertice.sigV.sigV == null)	N + 1
{	
cantLineasR += 2;	
	2^n

rutaCorta(tempVertice.sigV, (distancia + tempVertice.sigA.distancia), ruta2);
asignacionesR++;
return;
}
cantLineasR += 2; asignacionesR += 2;
rutaCorta(tempVertice.sigV, distancia + tempVertice.sigA.distancia, ruta2);
rutaCorta(tempVertice.sigV.sigV, distancia + tempVertice.sigA.sigA.distancia, ruta2);
	2^n
	2^n

Total	6^n + 3N + 6
Clasificación en notación O	O(n)

Medición empírica método Recursivo

Código algoritmo método recursivo:

Operaciones	***Tamaños datos de entrada en cantidad de vértices.***				
	20	40	60	80	100
Asig	1095	96766	144186134	6747194581	----------
Comp	2190	193579	288373091	1349438836	----------
Cantidad de líneas ejecutadas	3759	338500	504652422	23615178204	----------
Tiempo de ejecución	1 ms	4 ms	4,687 s	3,6453 min	----------
cantidad de líneas del	12	12	12	12	----------

Determinar el factor de crecimiento

	40/20	60/40	80/60
Factor de talla	2	1,5	1,33
Asig	88,3707	1490,0495	46,7950
Comp	88,3922	1489,6920	46,7948
Cantidad de líneas ejecutadas	90,0505	1490,8491	46,7949
Tiempo de ejecución	4 ms	1171,75 ms	46,6656 ms

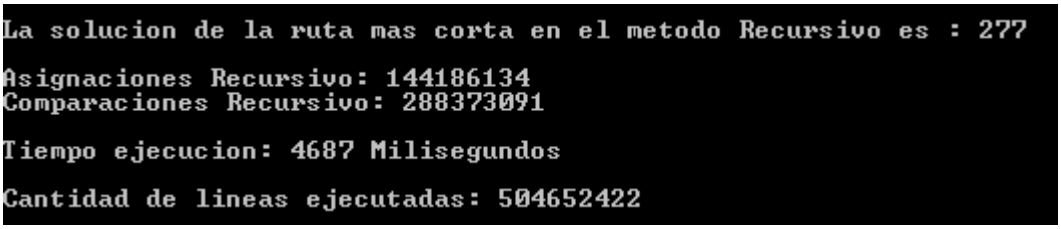
Cantidad de Vértices con Recursivo: 20.

```
La solucion de la ruta mas corta en el metodo Recursivo es : 73

Asignaciones Recursivo: 1095
Comparaciones Recursivo: 2190

Tiempo ejecucion: 0 Milisegundos

Cantidad de lineas ejecutadas: 3759
```

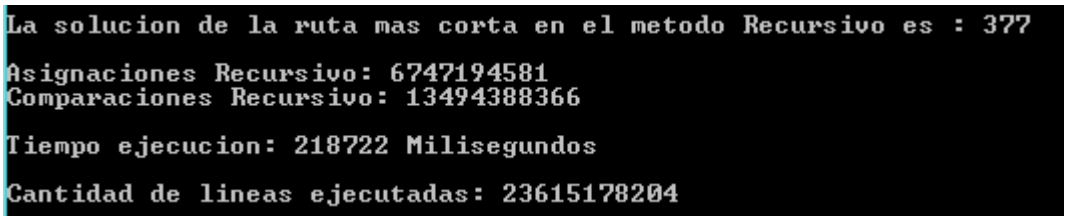
Cantidad de Vértices con Recursivo: 40.

```
La solucion de la ruta mas corta en el metodo Recursivo es : 146

Asignaciones Recursivo: 96766
Comparaciones Recursivo: 193579

Tiempo ejecucion: 4 Milisegundos

Cantidad de lineas ejecutadas: 338500
```

Cantidad de Vértices con Recursivo: 60.



Cantidad de Vértices con Recursivo: 80.



Medición analítica Iterativo

Código algoritmo método iterativo:

Código fuente	Medición de líneas ejecutadas en el peor de los casos (línea por línea)

`public void rutaCor(vertice tempVertice)`
`{`
`vertice tempV = tempVertice;`
 	
`while (true)`
`{`
`ruta = ruta + tempV.numero + ", ";`
`comparacionesC++; asignacionesC++;`
	1
	
	N + 2
	1
	
	N + 1

```
                if (tempV.sigV == null)
                {
                    comparacionesC++;
                    return;
                }
                else if (tempV.sigV.sigV == null)
                {
                    distancia += tempV.sigA.distancia;
                    tempV = tempV.sigV; comparacionesC
+= 2; asignacionesC += 2; cantLineasC += 5;
                }

                else if (tempV.sigV.sigA.sigA == null)
                {
                    if (tempV.sigA.sigA.distancia <
(tempV.sigA.distancia + tempV.sigV.sigA.distancia))
                    {
                        distancia +=
tempV.sigA.sigA.distancia;
                        tempV = tempV.sigV.sigV;
comparacionesC += 4; asignacionesC += 2; cantLineasC +=
7;
                    }
                    else
                    {
                        distancia +=
tempV.sigA.distancia;
                        tempV = tempV.sigV;
comparacionesC += 3; asignacionesC += 2; cantLineasC +=
7;
                    }
                }

                else if (tempV.sigA.sigA.distancia <
(tempV.sigA.distancia + tempV.sigV.sigA.distancia))
                {

                    if ((tempV.sigA.distancia +
tempV.sigV.sigA.sigA.distancia) <=
(tempV.sigA.sigA.distancia +
tempV.sigV.sigV.sigA.distancia))
                    {
                        distancia +=
tempV.sigA.distancia;
                        tempV = tempV.sigV;
comparacionesC += 5; asignacionesC += 2; cantLineasC +=
8;
                    }
                    else
                    {
                        distancia +=
tempV.sigA.sigA.distancia;
                        tempV = tempV.sigV.sigV;
comparacionesC += 4; asignacionesC += 2; cantLineasC +=
8;
                    }
                }
                else
```

N + 1

1
1

N + 1
N + 1

1
1

1

1

N + 1

N + 1

1
1

1
1

{
distancia += tempV.sigA.distancia;
tempV = tempV.sigV; comparacionesC += 4; asignacionesC += 2; cantLineasC += 8;
}
}
	1
	1

Total	7n + 22
Clasificación en notación O	Log(n)

Medición empírica Iterativo

Código algoritmo iterativo:

	Tamaños datos de entrada en cantidad de vértices.			
Operaciones	100	1000	10000	100000
Asig	202	2014	20293	202819
Comp	362	3591	36329	363151
Cantidad de líneas ejecutadas	535	5367	54109	540845
Tiempo de ejecución	1 ms	2 ms	57 ms	23720 ms
cantidad de líneas del código	26	26	26	26

Determinar el factor de crecimiento

	1000/100	10000/1000	100000/10000
Factor de talla	10	10	10
Asig	9, 9702	10, 0759	9, 9945
Comp	9, 9198	10, 1166	9, 9961
Cantidad de líneas ejecutadas	10, 0317	10, 0817	9, 9954
Tiempo de ejecución	2 ms	28, 5 ms	416, 1403

Cantidad de Vértices con Iterativo: 100.

```
La solucion de la ruta mas corta con el Iterativo es : 640

Asignaciones Iterativo: 202
Comparaciones Iterativo: 362

Tiempo ejecucion: 1 Milisegundos

Cantidad de lineas ejecutadas: 535
```

Cantidad de Vértices con Iterativo: 1000.

```
La solucion de la ruta mas corta con el Iterativo es : 5433

Asignaciones Iterativo: 2014
Comparaciones Iterativo: 3591

Tiempo ejecucion: 2 Milisegundos

Cantidad de lineas ejecutadas: 5367
```

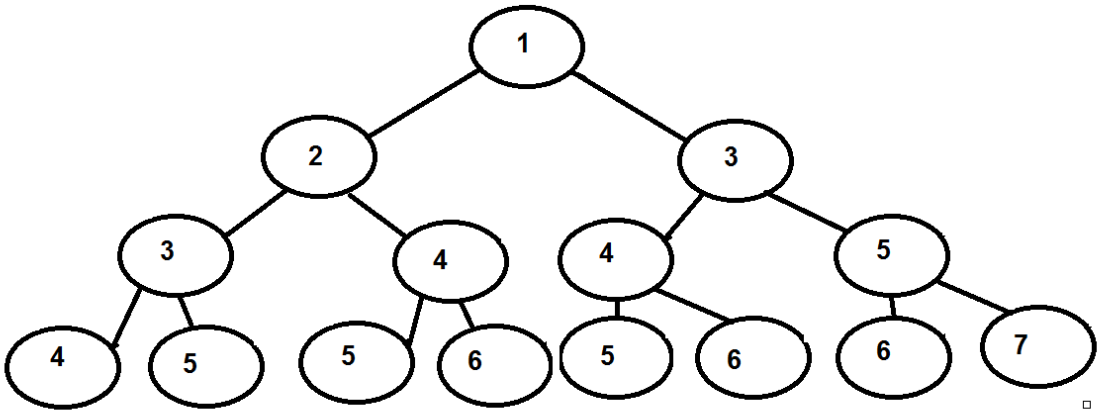
Cantidad de Vértices con Iterativo: 10000.

```
La solucion de la ruta mas corta con el Iterativo es : 55939

Asignaciones Iterativo: 20293
Comparaciones Iterativo: 36329

Tiempo ejecucion: 57 Milisegundos

Cantidad de lineas ejecutadas: 54109
```

Cantidad de Vértices con Iterativo: 100000.

```
La solucion de la ruta mas corta con el Iterativo es : 564479

Asignaciones Iterativo: 202819
Comparaciones Iterativo: 363151

Tiempo ejecucion: 23720 Milisegundos

Cantidad de lineas ejecutadas: 540845
```

Árbol de llamadas recursivas



Conclusiones

Comparando los diferentes resultados obtenidos por las pruebas a las que fue sometido el algoritmo de Dijkstra, el iterativo y el recursivo se pueden llegar a diferentes conclusiones como lo son:

- El algoritmo recursivo tuvo el peor desempeño con respecto a los otros dos. Pero la forma de programarlo es mucho más sencilla que sus rivales.
- El algoritmo de Dijkstra obtuvo un desempeño muy superior con respecto al recursivo y un poco inferior al iterativo. Pero su implementación es la más complicada con respecto a los demás.
- El algoritmo iterativo obtuvo un excelente rendimiento y además su implementación es muy sencilla

Estos análisis efectuados demostraron la importancia de medir la eficiencia de los algoritmos ya que en algunos casos existen diferentes formas de resolver un problema pero algunas soluciones son más eficientes que otros.

Recomendaciones

Actualmente en el mercado de aplicaciones de software es de gran importancia que dichas aplicaciones se ejecuten de la manera más rápida posible por este motivo es recomendable evitar algoritmos que generen una gran carga de ejecución. Como los son los exponenciales o $\mathbf{2^N}$. Y en cambio utilizar otros mas eficientes como los son los de tiempo logarítmico o **logN.**

Referencias Bibliográficas

Alvarez, C. (21 de 09 de 2014). *Calvarezg*. Obtenido de Algoritmo de Floyd: http://calvarezg.blogspot.com/2012/04/algoritmo-de-floyd.html

Universidad don Bosco. (21 de 09 de 2014). Obtenido de http://www.udb.edu.sv/udb/archivo/guia/informatica-ingenieria/programacion-iv/2013/ii/guia-10.pdf