Lógica de programación I

Juan Pablo Restrepo Uribe

Ing. Biomedico

MSc. Automatización y Control Industrial

jprestrepo@correo.iue.edu.co

Institución Universitaria de Envigado

Análisis y costo de algoritmos

Generalmente, un problema dado puede resolverse mediante muy diversos algoritmos o programas. No todas las soluciones son igualmente buenas.

- ¿De qué depende la calidad de un programa que resuelve un problema?
- ¿Elegancia de la idea en la que se basa el algoritmo?
- ¿Claridad en la organización del código del programa?
- ¿Vistosidad de la presentación de los resultados?
- ¿Eficiencia en la forma de obtener la solución?

Desde un punto de vista **computacional**, el factor esencial es la **eficiencia**.

Análisis y costo de algoritmos

Eficiencia: Capacidad de resolver el problema propuesto empleando un bajo consumo de recursos computacionales.
Dos factores fundamentales de la eficiencia:

- Coste Espacial: cantidad de memoria requerida
- Coste Temporal: tiempo necesario para resolver el problema.

Para resolver un problema dado, un algoritmo o programa **A** será mejor que otro **B** si **A** resuelve el problema en menos tiempo y/o emplea menos cantidad de memoria que **B**. En ocasiones tiempo y memoria son recursos competitivos y un buen algoritmo es aquel que resuelve el problema con un buen compromiso **tiempo/memoria**.

Análisis y costo de algoritmos

Cuando contamos con un algoritmo que permite resolver algún problema de cálculo, es importante tener conocimiento de cuál es su costo de cálculo, es decir, cual es el número de operaciones que deberían realizarse para completarlo y obtener el resultado que estamos buscando. Esto permite calificar el algoritmo y comparar su eficiencia con respecto a otro que resuelva el mismo problema.

$$p(x) = a_0 + a_1 x + a_2 x^2$$

Análisis y costo de algoritmos

En una computadora, todos los cálculos matemáticos se reducen a un conjunto de operaciones aritméticas elementales denominadas operaciones de punto flotante, flops. Estas operaciones son las de suma, resta, multiplicación y división. Tomemos como ejemplo un polinomio genérico de grado

$$q(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots + a_{n-1} x^{n-1} + a_n x^n$$

Análisis y costo de algoritmos

Un algoritmo es considerado eficiente si su consumo de recursos está en la media o por debajo de los niveles aceptables. Hablando a grandes rasgos, 'aceptable' significa: que el algoritmo corre en un tiempo razonable en una computadora dada y optimiza la mayor cantidad de recursos físicos.

```
1. int algo(int n){
2.     if (n == 0)
3.         return 400;
4.     else
5.         return 1+algo(n/2)+algo(n/2)+algo(n/2)+algo(n/2);
6. }
```

$$O(n^{\log_2 4}) = O(n^2)$$

Análisis y costo de algoritmos

```
void main(){ /*A1*/
  int m;
  m = 10 * 10; /*producto*/
  printf("%d\n", m);
}
```

```
void main(){ /*A2*/
  int i,m; m=0;
  for(i=1; i<=10; i++)
    m = m + 10; /*suma*/
  printf("%d\n", m);
}
```

```
void main(){ /*A3*/
  int i,j,m; m=0;
  for(i=1; i<=10; i++)
    for(j=1; j<=10; j++)
      m++;    /*sucesor*/
  printf("%d\n", m);
}
```

Análisis y costo de algoritmos

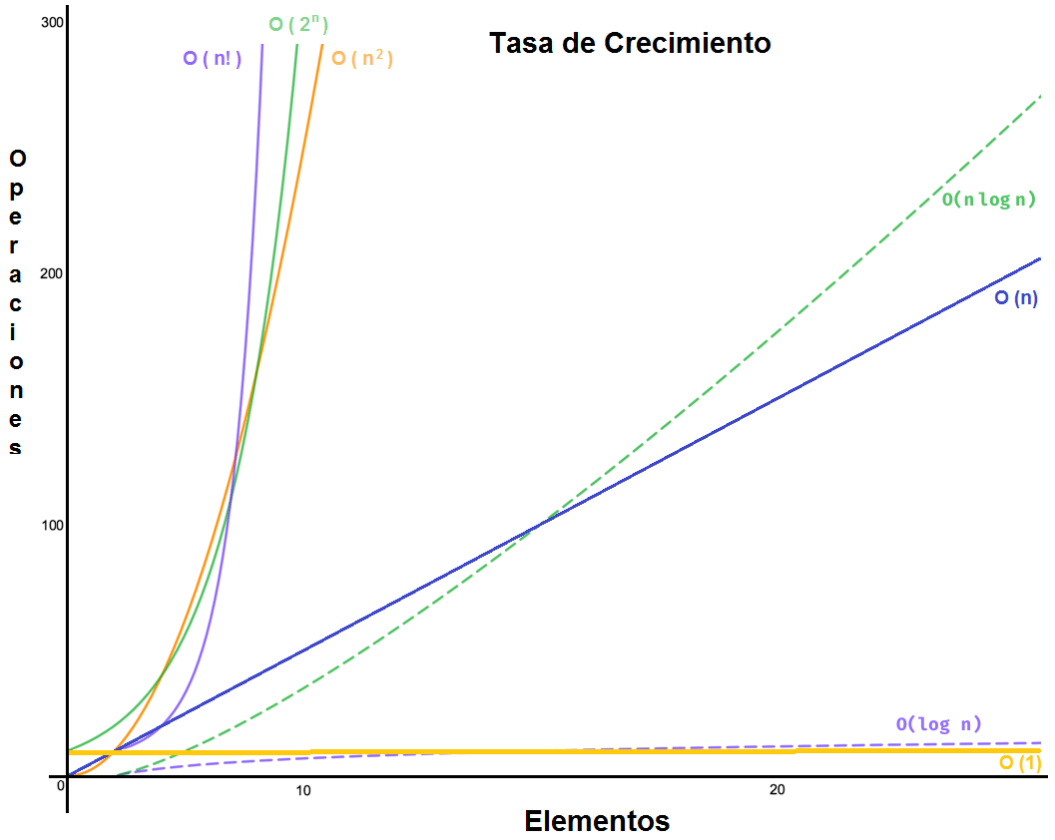
$$T_{A1} = t_* \qquad T_{A2} = 10\,t_+ \qquad T_{A3} = 100\,t_s$$

t_*	$100\ \mu s$	$50\ \mu s$	$100\ \mu s$	$200\ \mu s$
t_+	$10\ \mu s$	$10\ \mu s$	$5\ \mu s$	$10\ \mu s$
t_s	$1\ \mu s$	$2\ \mu s$	$1\ \mu s$	$0.5\ \mu s$
$A1$	$100\ \mu s$	$50\ \mu s$	$100\ \mu s$	$200\ \mu s$
$A2$	$100\ \mu s$	$100\ \mu s$	$50\ \mu s$	$100\ \mu s$
$A3$	$100\ \mu s$	$200\ \mu s$	$100\ \mu s$	$50\ \mu s$

Análisis y costo de algoritmos

Notación	Nombre	Ejemplos
$O(1)$	constante	Determinar si un número es par o impar. Usar una tabla de consulta que asocia constante/tamaño. Usar una función hash para obtener un elemento.
$O(\log n)$	logarítmico	Buscar un elemento específico en un array utilizando un árbol binario de búsqueda o un árbol de búsqueda balanceado, así como todas las operaciones en un Heap binomial.
$O(n)$	lineal	Buscar un elemento específico en una lista desordenada o en un árbol degenerado (peor caso).
$O(n \log n)$	loglinear o quasilinear	Ejecutar una transformada rápida de Fourier; heapsort, quicksort (caso mejor y promedio), o merge sort
$O(n^2)$	cuadrático	Multiplicar dos números de *n* dígitos por un algoritmo simple. bubble sort (caso peor o implementación sencilla), Shell sort, quicksort (caso peor).
$O(c^n),\ c > 1$	exponencial	Encontrar la solución exacta al problema del viajante utilizando programación dinámica. Determinar si dos sentencias lógicas son equivalentes utilizando una búsqueda por fuerza bruta

Eficiencia algorítmica

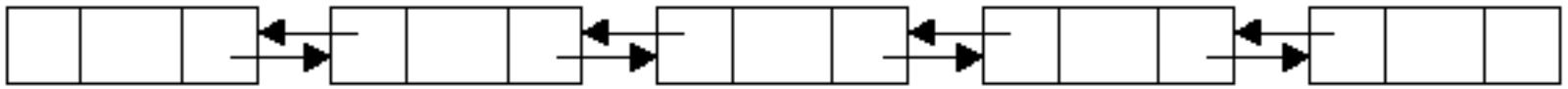


¿Coste de los algoritmos recursivos?

https://www.wextensible.com/temas/recursivos/coste.html

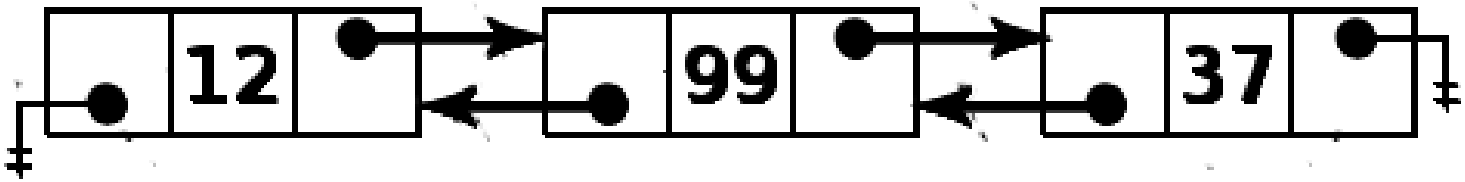
Lista enlazada doble

En algunas aplicaciones podemos desear recorrer la lista hacia adelante y hacia atrás, o dado un elemento, podemos desear conocer rápidamente los elementos anterior y siguiente. En tales situaciones podríamos desear darle a cada celda sobre una lista un puntero a las celdas siguiente y anterior en la lista tal y como se muestra en la figura.



Lista enlazada doble

Es una estructura de datos que consiste en un conjunto de **nodos enlazados secuencialmente**. Cada nodo contiene tres campos, **dos para los enlaces**, que son referencias al **nodo siguiente y al anterior** en la secuencia de nodos, y otro más para el **almacenamiento de la información.** El enlace al nodo anterior del primer nodo y el enlace al nodo siguiente del último nodo, apuntan a un tipo de nodo que marca el final de la lista, normalmente un nodo centinela o puntero null, para facilitar el recorrido de la lista. Si existe un único nodo centinela, entonces la lista es circular a través del nodo centinela.

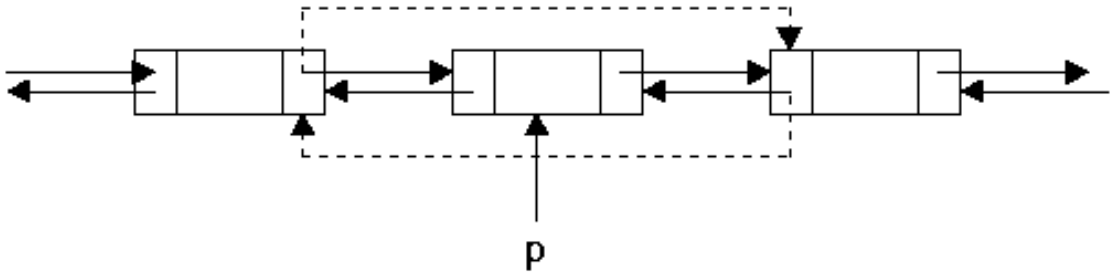


Lista enlazada doble

El enlace al nodo anterior del primer nodo y el enlace al nodo siguiente del último nodo, apuntan a un tipo de nodo que marca el final de la lista, normalmente un nodo centinela o puntero **NULL**, para facilitar el recorrido de la lista.

Lista enlazada doble

Podemos usar un puntero a la celda que contiene el i-ésimo elemento de una lista para representar la posición i, mejor que usar el puntero a la celda anterior, aunque lógicamente, también es posible la implementación similar a la expuesta en las listas simples haciendo uso de la cabecera. El único precio que pagamos por estas características es la presencia de un puntero adicional en cada celda y consecuentemente procedimientos algo más largos para algunas de las operaciones básicas de listas.

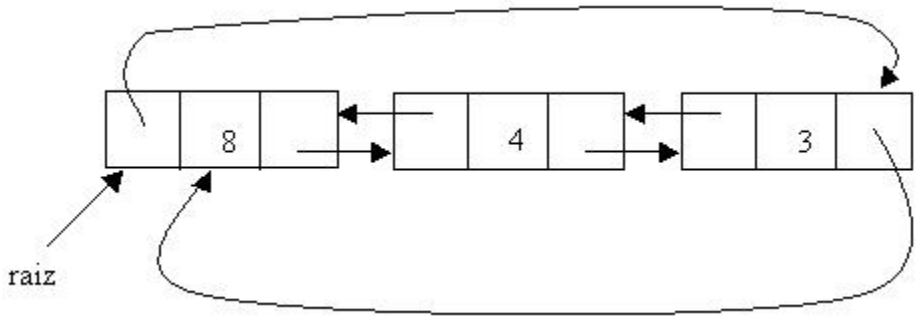


Lista enlazada doble

Operaciones	Matricial	Enlace Simple	Enlace Doble
Crear	O(1)	O(1)	O(1)
Destruir	O(1)	O(n)	O(n)
Primero	O(1)	O(1)	O(1)
Fin	O(1)	O(n)/O(1)	O(1)
Insertar	O(n)	O(1)	O(1)
Borrar	O(1)	O(n)	O(n)
Elemento	O(1)	O(1)	O(1)
Siguiente	O(1)	O(1)	O(1)
Anterior	O(1)	O(n)	O(1)
Posicion	O(n)	O(n)	O(n)
• Memoria	• Tamaño fijo	• Sobrecarga de un puntero por nodo. • Un nodo sin información.	• Sobrecarga de dos punteros por nodo. • Un nodo sin información.

Lista enlazada doble

- Nos permite almacenar datos de una forma organizada.
- Es una estructura TDA dinámica.
- Cada nodo de la lista doblemente enlazada contiene dos punteros, de forma que uno apunta al siguiente nodo y el otro predecesor

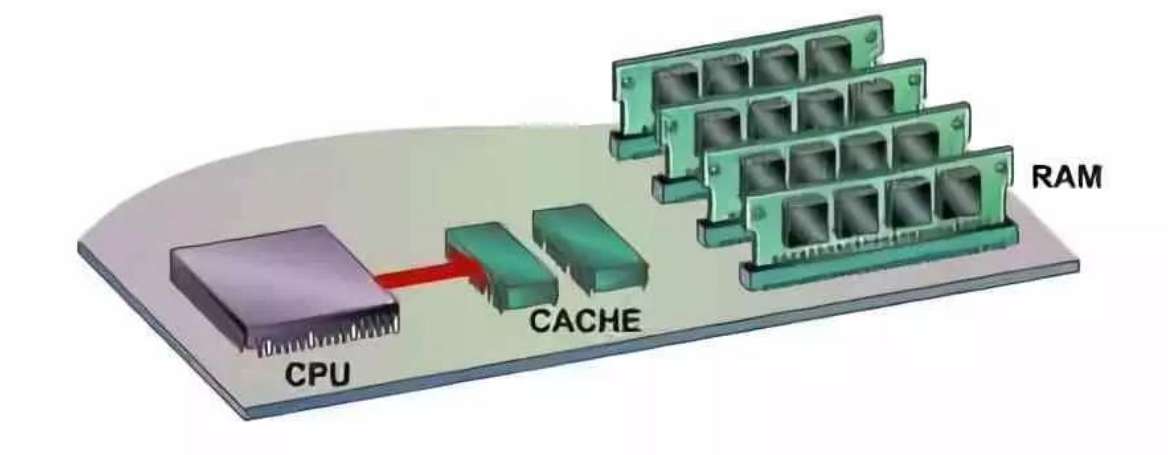


Lista enlazada doble

- **Uso de memoria:** Las listas doblemente enlazadas requieren más memoria que las listas enlazadas simples porque cada nodo debe contener dos punteros
- **Complejidad de implementación**
- **Mayor tiempo de acceso:** Aunque las listas doblemente enlazadas ofrecen la capacidad de acceso directo tanto al siguiente como al nodo anterior, esto también significa que los accesos requieren más operaciones de puntero en comparación con las listas enlazadas simples.
- **Uso ineficiente del caché:** Debido a que los nodos no están necesariamente contiguos en memoria, puede resultar en un uso menos eficiente del caché del sistema.

Memoria cache

Una memoria caché es una capa de almacenamiento de datos de alta velocidad que almacena un subconjunto de datos, normalmente transitorios, de modo que las solicitudes futuras de dichos datos se atienden con mayor rapidez que si se debe acceder a los datos desde la ubicación de almacenamiento principal. El almacenamiento en caché permite reutilizar de forma eficaz los datos recuperados o procesados anteriormente.



Lista enlazada doble

- **Listas dobles lineales:** tanto el puntero izquierdo del primer nodo (cabeza) como el derecho del último nodo (fin de la lista) son NULOS.

- **Listas dobles circulares:** el puntero izquierdo del primer nodo apunta al último nodo de la lista y el puntero derecho del último nodo apunta al primer nodo de la lista.

Nodo

En este caso el nodo cambia en su estructura, ya que deberá contar con dos punteros que indiquen la posición tanto del nodo siguiente como del nodo anterior

```python
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.siguiente = None

# Ejemplo de uso
nodo1 = Nodo(10)
nodo2 = Nodo(20)
nodo3 = Nodo(30)

nodo1.siguiente = nodo2
nodo2.siguiente = nodo3

nodo_actual = nodo1
while nodo_actual is not None:
    print(nodo_actual.dato)
    nodo_actual = nodo_actual.siguiente

```

```python
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.siguiente = None
        self.anterior = None

nodo1 = Nodo(10)
nodo2 = Nodo(20)
nodo3 = Nodo(30)

nodo1.siguiente = nodo2
nodo2.anterior = nodo1
nodo2.siguiente = nodo3
nodo3.anterior = nodo2

nodo_actual = nodo1
while nodo_actual is not None:
    print(nodo_actual.dato)
    nodo_actual = nodo_actual.siguiente

print()

nodo_actual = nodo3
while nodo_actual is not None:
    print(nodo_actual.dato)
    nodo_actual = nodo_actual.anterior
```

¿Circular?

Ejercicios

- Escribe una función que tome una lista de números como entrada y devuelva la suma de todos los elementos de la lista.
- Escribe una función que tome una lista y me diga cuales son los elementos duplicados en la lista
- Escribe una función que tome una lista doblemente enlazada y devuelva una nueva lista con los elementos en orden inverso.
- Escribe una función que tome una lista enlazada doble y elimine el último elemento de la lista.