UNSA

UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

FACULTAD DE INGENIERÍA Y PRODUCCIÓN DE SERVICIOS

ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

"AÑO DEL BICENTENARIO DEL PERÚ: 200 AÑOS DE INDEPENDENCIA"

ESTADÍSTICA Y PROBABILIDADES

GRUPO: B SEMESTRE: 2021 - B

Análisis Teórico de la Complejidad Computacional

INTEGRANTES:

Sumare Uscca Josue Gabriel

Gutierrez Arratia Ronald Romario

Ituccayasi Umeres Marko Marcelo

Alvarez Astete Jheeremy Manuel

DOCENTE:

Ronny Ivan Gonzales Medina

Arequipa – Perú

2021

ÍNDICE

INTRODUCCIÓN 3

CAPÍTULO I 4

DEFINICIONES Y CONCEPTOS PRINCIPALES 4

CAPÍTULO II 6

DESARROLLO DEL TEMA 6

¿Qué lenguajes de programación son los más usados? 6

TIEMPO DE EJECUCIÓN 6

CONSUMO DE MEMORIA 7

CONSUMO DE ENERGÍA 8

Complejidad en algoritmos de búsqueda de estructuras de datos 9

Búsqueda en Vectores 12

Búsqueda en listas enlazadas dobles 20

Búsqueda en Árboles RB 22

Búsqueda en Árboles AVL 24

Búsqueda en Tablas Hash 25

Búsqueda en Binary Heap 26

Búsqueda en Binomial Heap 26

Búsqueda en Fibonacci Heap 27

Cuadro de comparaciones de la complejidad algorítmica de búsqueda 28

CONCLUSIONES 29

BIBLIOGRAFÍA 30

ANEXOS 31

INTRODUCCIÓN

La teoría de la complejidad es una rama de la teoría de la computación que analiza a través de modelos matemáticos la “dificultad” de los algoritmos.

Dicha complejidad se refiere a la cantidad de pasos o líneas de código que necesita un algoritmo para funcionar, lo que trae consigo un consumo de recursos relacionados al hardware del computador proporcional a dicha cantidad.

El propósito principal de esta rama es el determinar los límites de aquello programable en una computadora, es decir aquello que una computadora puede hacer considerando los recursos disponibles. La eficiencia es un término muy importante a definir, hace referencia a una complejidad que hace uso de menos recursos.

Los lenguajes de programación tienden a ser evaluados también por diferentes parámetros con el fin de determinar el uso de recursos, la tendencia más común es que mientras dichos lenguajes sean más “fáciles” de programar el consumo de recursos es mayor, por el hecho que para que tengan estas características necesitan de pasar por varias “capas” que facilitan la generación de código.

A continuación, presentaremos parámetros relacionados a el consumo de energía de diferentes lenguajes de programación para así poder un rango de aquellos lenguajes con mayor eficiencia en esta característica.

CAPÍTULO I

DEFINICIONES Y CONCEPTOS PRINCIPALES

La ciencia de la computación es un cuerpo sistematizado del conocimiento concerniente al cálculo, que se sostiene en dos áreas fundamentales: La Teoría de la Computabilidad, basada en las ideas y los modelos fundamentales subyacentes al cálculo, y las técnicas de la ingeniería para el diseño de algoritmos: algoritmos voraces (divide y vencerás), algoritmos aleatorizados, algoritmos con retroceso, etc

Un proceso computacional, también llamado proceso algorítmico o algoritmo, es fundamental para la ciencia de la computación, puesto que un computador no puede ejecutar un problema que no tenga una solución algorítmica. Así, cualquier limitación de las capacidades de los procesos computacionales constituye también una limitación de las capacidades de la computadora [1].

. Evaluar la eficiencia de los algoritmos tiene mucho que ver con evaluar la complejidad de los mismos. Aunque esta labor pueda resultar ardua, compensa el hecho de obtener programas con alta garantía de corrección y la satisfacción intelectual de haberlos construido bien desde el principio, sin esperar el veredicto del sistema computacional.

El tema de la computabilidad tiene mucho que ver con la búsqueda de las estructuras que se requieren en un lenguaje de programación, de tal forma que se asegure que un programa expresado en dicho lenguaje pueda resolver cualquier problema que tenga una solución algorítmica [2], por ende, la Teoría de la Complejidad Computacional es la parte de la teoría de la computación que estudia los recursos requeridos durante el cálculo para resolver un problema.

Los recursos comúnmente estudiados son el tiempo (número de pasos de ejecución de un algoritmo para resolver un problema) y el espacio (cantidad de memoria utilizada para resolver un problema). Un algoritmo que resuelve un problema pero que tarda mucho en hacerlo, difícilmente será de utilidad. Igualmente, un algoritmo que necesite un gigabyte de memoria no será probablemente utilizado. A estos recursos se les puede añadir otros, tales como el número de procesadores necesarios para resolver el problema en paralelo.

Debido a que cada lenguaje es único, puede existir entre lenguajes compilados e interpretados:

COMPILADOS

Como la palabra lo dice un lenguaje que es compilado necesita un compilador si o si, este compilador podríamos decir que es el que se encarga de entender todo lo que estamos escribiendo. Los lenguajes que necesitan de un compilador trabajan de la siguiente forma: Se escribe un archivo y normalmente con una extensión dependiendo del lenguaje, por ejemplo: archivo.c, archivo.java, etc. Este archivo necesita parar por el compilador, el compilador se encargará de leer el archivo, las cantidades de veces que se lee el archivo varía.

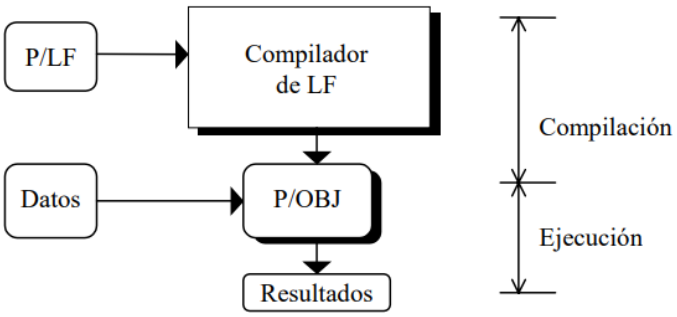

Figura 1: Esquema general de un compilador

INTERPRETADOS

El intérprete podemos decir que tiene un proceso mucho más sencillo, el intérprete lo que hace es leer el archivo y hacer la traducción(interpretar) lo que entiende, normalmente lo hace línea por línea. Con esto podemos decir que se comporta como un intérprete en verdad, las personas que están durante la transmisión de un evento deportivo, un debate o una entrega de premios y que escucha lo que dice la persona y va interpretando conforme lo va entiendo. Además, los intérpretes no generan archivos para almacenar el resultado de lo que se traduce. Como podemos leer es mucho más sencillo de describir.

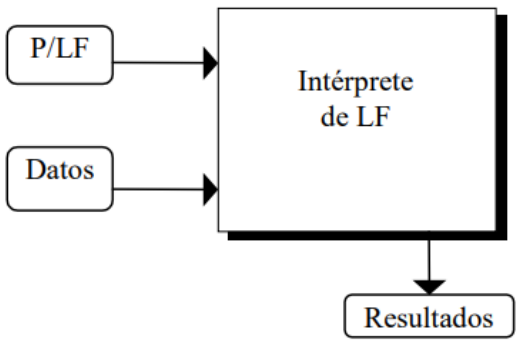

Figura 2: Esquema general de un intérprete

CAPÍTULO II

DESARROLLO DEL TEMA

¿Qué lenguajes de programación son los más usados?

Hay muchas preguntas que surgen cuando se intenta revisar el uso eficiente de energía de un lenguaje de programación. Sin embargo, comparar los lenguajes es difícil. El rendimiento de un lenguaje puede ser fácilmente mejorado sólo con la calidad del compilador o máquina virtual. El optimizar el código fuente es tan importante como las librerías mejoradas

Utilizando the Computer Benchmarks Game[3], el equipo de investigadores probó varios lenguajes compilando/ejecutando dichos programas utilizando los mejores compiladores, máquinas virtuales, intérpretes y librerías. Luego analizaron el rendimiento de las diferentes implementaciones considerando tres variables:

- Tiempo de ejecución
- Consumo de memoria
- Consumo de energía

Dentro de los lenguajes llevados analizar estas pruebas se eligieron los más usados tanto en desarrolladores así como empresas.

TIEMPO DE EJECUCIÓN

Se conoce como tiempo de ejecución (Runtime) en programación al intervalo de tiempo que va desde que el sistema operativo comienza a ejecutar las instrucciones de un determinado programa, hasta que finaliza la ejecución del mismo, ya sea porque el programa concluyó exitosamente o porque fue finalizado por el sistema operativo a causa de un fallo en tiempo de ejecución.

El hecho que el tiempo de ejecución dependa de la entrada indica que el tiempo de ejecución de un programa debe estar definido como una función de los datos de entrada, que puede especificarse con el “tamaño” de la entrada. Así, por ejemplo, en un programa de

ordenamiento el tamaño natural de medida para la entrada es el número de elementos a ordenar. Es usual indicar con T(n) el tiempo de ejecución de un programa para una entrada de tamaño n. Las unidades de T(n) pueden dejarse sin especificar, pero puede pensarse que es el número de instrucciones ejecutadas en un ordenador ideal.

Las pruebas hechas de tiempo entre distintos lenguajes de programación por excelencia entre los desarrolladores, arrojaron los siguientes resultados.

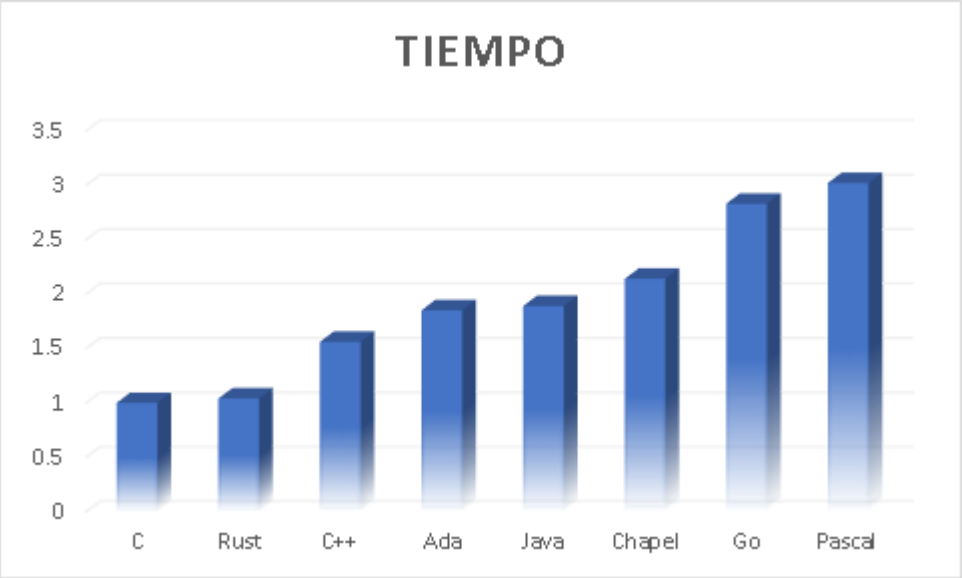

Figura 3: Esquema de tiempo de ejecución de LP

Como podemos observar en la figura 3, dentro de los lenguajes de programación más usados, el lenguaje de programación C es el que ha tenido mejores resultados en cuando al tiempo, ya que al ser un lenguaje de programación compilado, es más fácil para este poder analizar diferentes algoritmos con un tiempo inmediato al devolver el resultado.

CONSUMO DE MEMORIA

Nuestro sistema operativo utiliza la memoria para almacenar los datos y archivos de los programas que utilizamos para poder acceder a ellos de una manera más rápida, por eso la administración de memoria de una computadora es una tarea fundamental debido a que la cantidad de memoria es limitada. El sistema operativo es el encargado de administrar la memoria del sistema y compartirla entre distintos usuarios y/o aplicaciones. El RTS (Run Time System) de un lenguaje de programación administra la memoria para cada programa en ejecución.

La ejecución de un programa requiere que diversos elementos se almacenan en la memoria: Código del programa (instrucciones) Datos Permanentes Temporales Direcciones para controlar de flujo de la ejecución del programa

La asignación de memoria para algunos elementos fijos del programa que es controlada por el compilador se le llama asignación de memoria estática. A la asignación y posible recuperación de memoria durante la ejecución de un programa y bajo su control, se le llama asignación de memoria dinámica.

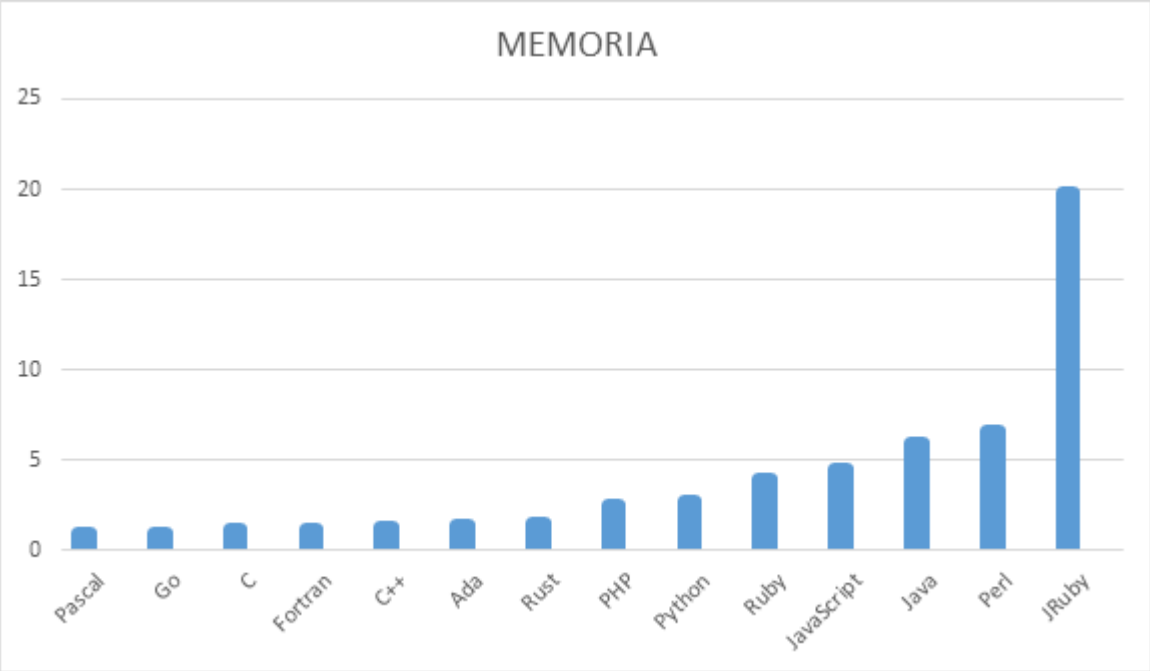

Figura 4: Esquema de consumo de Memoria en LP

CONSUMO DE ENERGÍA

Los dispositivos móviles e integrados actuales ejecutan programas escritos con diferentes lenguajes de programación. Por esta razón, también es importante considerar la eficiencia energética de un software escrito con un lenguaje de programación particular. Por eso es importante analizar el algoritmo en eficiencia energética, y su implementación eficiencia energética porque hoy en día sólo conocemos el impacto en términos de uso de memoria y tiempo de ejecución.

El siguiente diagrama de barras muestra la distribución de los lenguajes de programación más usados en cuanto a consumo energético que no solo van a depender del tiempo de compilación, sino también del consumo de memoria.

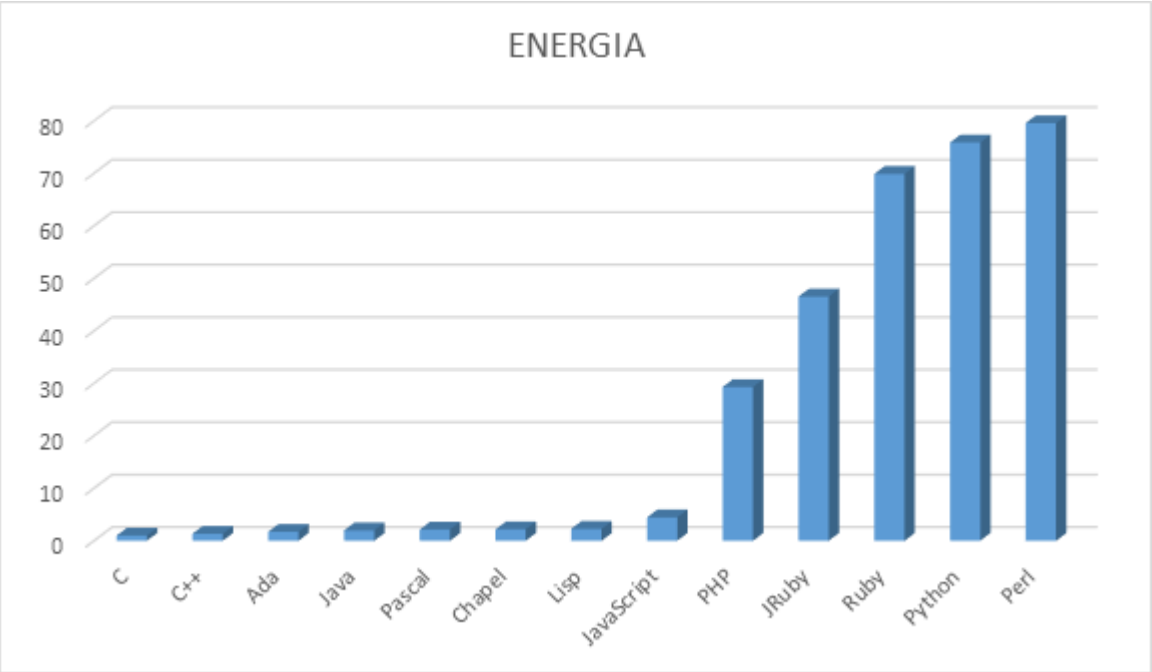

Figura 5: Esquema de consumo energético en LP

Como resultado los lenguajes C, C++ y Ada son los más eficientes energéticamente, aunque no necesariamente los que menos memoria consumen. Además, el paradigma de programación utilizado influye directamente en la eficiencia energética, siendo los más eficientes, los lenguajes imperativos, los orientados a objetos, los funcionales y finalmente los lenguajes interpretados. Pero lo más importante, en el consumo de energía lo que más influye, es el uso del procesador.

Complejidad en algoritmos de búsqueda de estructuras de datos

La complejidad algorítmica representa la cantidad de recursos (temporales) que necesita un algoritmo para resolver un problema y por tanto permite determinar la eficiencia de dicho algoritmo. Con respecto a los algoritmos y las estructuras de datos; esto puede ser por tiempo o espacio (espacio en disco) necesario para realizar una tarea específica (buscar, clasificar o acceder a los datos) en una estructura de datos determinada. La eficiencia del rendimiento de una tarea depende del número de operaciones requeridas para completarla.

Los procesos de búsqueda involucran recorrer un arreglo completo con el fin de encontrar algo. Lo más común es buscar el menor o mayor elemento (cuando se puede establecer un orden), o buscar el índice de un elemento determinado.

Existen diferentes algoritmos de búsqueda y la elección depende de la forma en que se encuentren organizados los datos: si se encuentran ordenados o si se ignora su disposición o se sabe que están al azar. También depende de si los datos a ordenar pueden ser accedidos de modo aleatorio o deben ser accedidos de modo secuencial.

Usaremos la notación denominada Big-O, y también conocida como Notación Asintótica o Notación Landau (en honor a uno de sus inventores, a principios del siglo pasado, Edmund Landau), donde mediremos el rendimiento o la complejidad de un algoritmo.

La notación Big-O nos proporciona una manera de saber cómo se va a comportar un algoritmo en función de los argumentos que le pasemos y la escala de los mismos.
Por ejemplo, imagínate una función que se utiliza para localizar un elemento dentro de una lista de elementos previamente guardados. Si la documentación de la misma nos dice que es una operación de tipo O(1), quiere decir que da igual cuántos elementos haya en la lista, la operación siempre tarda lo mismo. Para ser más exactos deberíamos decir que el esfuerzo de cómputo necesario es el mismo.

Es muy importante conocer el tipo de función/curva que se asocia con la ejecución de una función ya que nos permitirá saber de antemano si el rendimiento de nuestra aplicación se puede resentir en caso de que el tamaño de los datos a manejar aumente mucho. Así, entre dos algoritmos/funciones que hagan lo mismo, deberíamos elegir el que tenga una complejidad asintótica menor, para lo cual consultaremos la notación Big-O asociada al mismo.

Atendiendo a su complejidad, las notaciones Big-O más comunes para todo tipo de algoritmos y funciones son las que se muestran en esta lista:

- **O(1):** constante. La operación no depende del tamaño de los datos. Es el caso ideal, pero a la vez probablemente el menos frecuente. No se ve en la gráfica de más abajo porque la logarítmica le pasa justo por encima y la tapa.
- **O(n)**: lineal. El tiempo de ejecución es directamente proporcional al tamaño de los datos. Crece en una línea recta.

- **O(log n)**: logarítmica. por regla general se asocia con algoritmos que "trocean" el problema para abordarlo, como por ejemplo una búsqueda binaria.
- **O(n logn):** en este caso se trata de funciones similares a las anteriores, pero que rompen el problema en varios trozos por cada elemento, volviendo a recomponer información tras la ejecución de cada "trozo". Por ejemplo, el algoritmo de búsqueda Quick Sort
- **O(n^2)**: cuadrática. Es típico de algoritmos que necesitan realizar una iteración por todos los elementos en cada uno de los elementos a procesar. Por ejemplo, el algoritmo de burbuja. Si tuviese que hacer la iteración más de una vez serían de complejidad O(n3), O(n4), etc.… pero se trata de casos muy raros y poco optimizados.
- **O(2^n):** exponencial. Se trata de funciones que duplican su complejidad con cada elemento añadido al procesamiento. Son algoritmos muy raros pues en condiciones normales no debería ser necesario hacer algo así. Un ejemplo sería, por ejemplo, el cálculo recursivo de la serie de Fibonacci, que es muy poco eficiente (se calcula llamándose a sí misma la función con los dos números anteriores: F(n)=F(n-1)+F(n-2)).
- **O(n!):** explosión combinatoria. Un algoritmo que siga esta complejidad es un algoritmo totalmente fallido. Una explosión combinatoria se dispara de tal manera que cuando el conjunto crece un poco, lo normal es que se considere computacionalmente inviable. Solo se suele dar en algoritmos que tratan de resolver algo por la mera fuerza bruta. No deberías verlo nunca en un software "real".

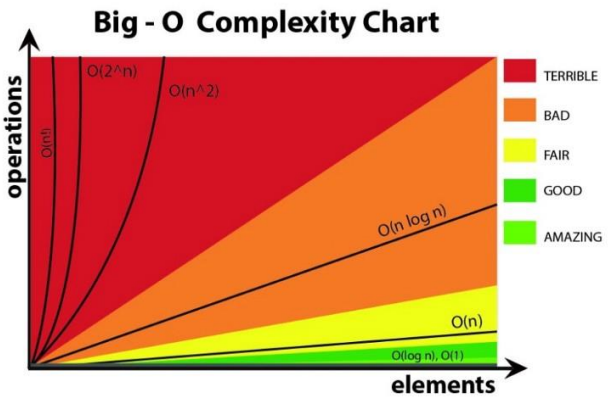

Figura 6: Esquema de la complejidad Big-O.

Como podemos observar, a medida que aumenta la complejidad, el tiempo necesario para completar la tarea crece mucho, pudiendo llegar a aumentar enormemente en algunos casos en cuanto hay más datos a manejar. Creo que verlo gráficamente es la mejor manera de darse cuenta de su importancia. Si te encuentras con una función con complejidad cuadrática (O(n^2)) o exponencial (O(2^n)) por regla general será señal de que el algoritmo necesita una revisión urgente, y mejor no utilizarlo.

En la siguiente tabla muestra el tiempo que tardaría un computador en realizar f(n) operaciones, para distintas funciones f y distintos valores de n, suponiendo que se realizan un millón de operaciones por segundo.

n / f(n)	10	20	30	40	50	70	100
n	0.00001 s	0.00002 s	0.00003 s	0.00004 s	0.00005 s	0.00007 s	0.0001 s
n log(n)	0.00003 s	0.00008 s	0.00014 s	0.00021 s	0.00028 s	0.00049 s	0.0006 s
n^2	0.0001 s	0.0004 s	0.0009 s	0.0016 s	0.0025 s	0.0049 s	0.01 s
n^3	0.001 s	0.008 s	0.027 s	0.064 s	0.125 s	0.343 s	1 s
n^4	0.01 s	0.16 s	0.81 s	2.56 s	6.25 s	24 s	1.6 min
n^5	0.1 s	3.19 s	24.3 s	1.7 s	5.2 min	28 min	2.7 horas
$n^{Log(n)}$	0.002 s	4.1 s	17.7 s	5 min 35 s	1 h 4min	2.3 días	224 días
2^n	0.001 s	1.04 s	17 min	12 dias	35.6 a.	37 Ma	2.6 MEU
3^n	0.059 s	58 min	6.5 a.	385495 a.	22735 Ma	52000 Ma	10^{18} MEU
n!	3.6 s	77054 a.	564 MEU	$1.6\ 10^{18}$ME	$6\ 10^{34}$	$2.4\ 10^{70}$M	2
n^n	2.7 horas	219 EU	$4.2\ 10^{14}$ME	$2.4\ 10^{28}$ME	$1.8\ 10^{67}$ME	$3\ 10^{111}$ M	$2\ 10^{182}$ M

s: segundos min: minutos h: horas a: añosMa: millones de años
EU: edad del universo MEU: millones de veces la edad del universo

Figura 7: Tiempo en segundos que tarda en realizar f(n) operaciones, un millón por segundo.

Búsqueda en Vectores

Para buscar un dato en un vector existen dos métodos: búsqueda secuencial o lineal y búsqueda binaria. El primero es más fácil de implementar, pero puede tomar más tiempo, el segundo es más eficiente, pero requiere que el vector esté ordenado.

➢ **Búsqueda Lineal:**

- Búsqueda del menor

```
menor = a[0];
for (i=1;i<n;i++)
        if ( a[i]<menor )
                menor=a[i];
```

- Búsqueda del mayor

```
mayor= a[n-1];
for (i=0;i<n-1;i++)
        if ( a[i]>mayor )
           mayor=a[i];
```

- Búsqueda de elemento

```
 encontrado=-1;
for (i=0;i<n;i++)
        if ( a[i]==elemento_buscado )
           encontrado=i;
```

Figura 8: Algoritmo de búsqueda lineal

Este método consiste en tomar un dato clave que identifica al elemento que se busca y hacer un recorrido a través de todo el arreglo comparando el dato de referencia con el dato de cada posición. Supóngase que se tienen una lista de estudiantes y se desea ubicar al que se identifica con el número 27844562. La búsqueda consiste en comparar dicho número con la identificación de cada estudiante de la lista.

La búsqueda terminará en el evento de encontrar una coincidencia en los números o si al llegar al final de la lista no se encontró identificación igual al número buscado, en cuyo caso se concluye que el dato no existe en el vector.

En la figura 9 se presenta el diagrama del flujo de una función con los pasos generales para hacer una búsqueda secuencial en un vector

La función recibe como parámetros el vector y la clave del elemento a buscar, recorre el vector hasta que se encuentre el elemento, en cuyo caso la posición se registra en la variable pos, o hasta que se llegue al final del vector. Al finalizar el algoritmo devuelve la variable pos, si el dato fue encontrado ésta contiene la posición que ocupa en el vector, en caso contrario, reporta el valor menos uno (-1) indicando que el dato no está en el vector.

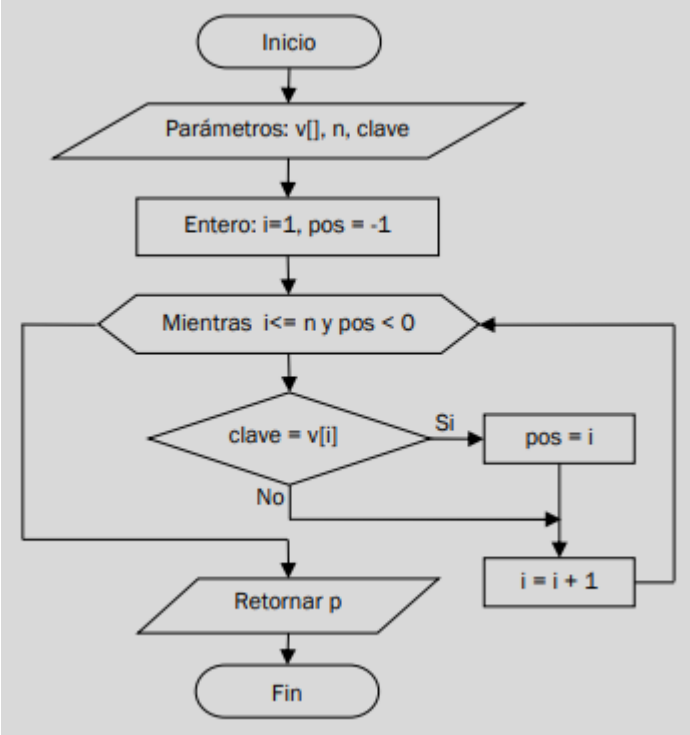

Figura 9: Diagrama de flujo de algoritmo de búsqueda lineal

Para buscar un estudiante, un profesor guarda los datos de sus estudiantes en tres vectores: código, nombre y nota, como se muestra en la figura 112. Se requiere un algoritmo para consultar la nota de un estudiante a partir de su código.

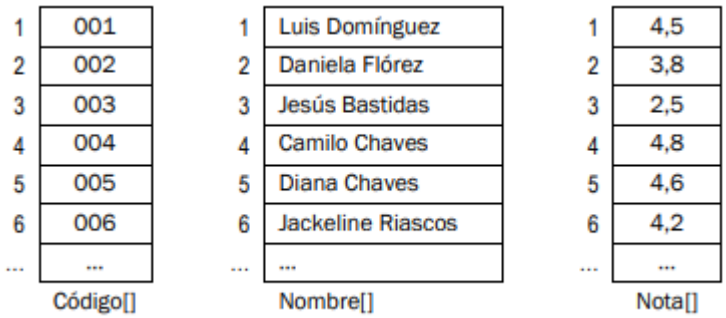

Figura 10: Datos de estudiantes almacenados en vectores

Haciendo uso de la función búsqueda lineal() y pasándole como parámetros el vector código, el tamaño del mismo y el código del estudiante, se puede conocer en qué posición de los vectores se encuentran los datos del estudiante. El algoritmo se presenta en la figura 8, en éste se supone la existencia de un procedimiento para llenar los datos en los vectores.

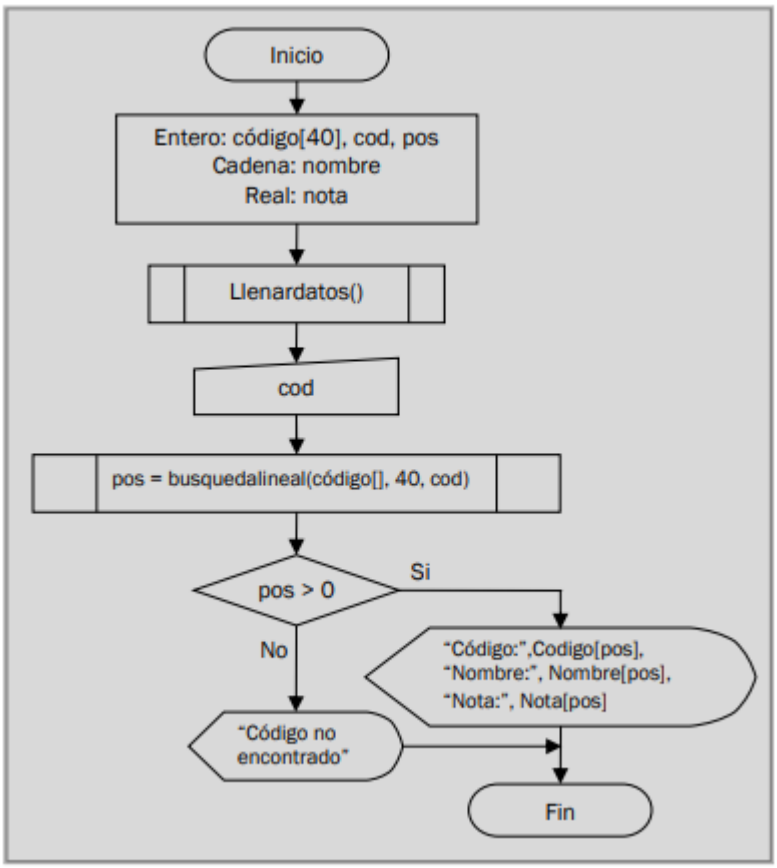

Figura 11: Diagrama de flujo del algoritmo de buscar estudiante

Eficiencia y Complejidad

Considerando la Cantidad de Comparaciones

1 : Mejor Caso: • El elemento buscado está en la primera posición. Es decir, se hace una sola comparación

n : Peor Caso: • El elemento buscado está en la última posición. Necesitando igual cantidad de comparaciones que de elementos el arreglo

n/2 : En Promedio: • El elemento buscado estará cerca de la mitad. Necesitando en promedio, la mitad de comparaciones que de elementos.

Por lo tanto, la velocidad de ejecución depende linealmente del tamaño del arreglo

- **Búsqueda Binaria**

  Este método es más eficiente que la búsqueda secuencial pero sólo se puede aplicar sobre vectores o listas de datos ordenados.

En la búsqueda binaria no se hace un recorrido de principio a fin, sino que se delimita progresivamente el espacio de búsqueda hasta llegar al elemento buscado. La primera comparación se hace con el elemento de la mitad del arreglo, si aquel no es el dato buscado, se decide si buscar en la mitad inferior o en la mitad superior según la clave sea menor o mayor del elemento de la mitad. Se toma como espacio de búsqueda la mitad del vector que corresponda y se procede de igual forma, se compara con el elemento del centro, si ese no és el que se busca, se toma un nuevo espacio de búsqueda correspondiente a la mitad inferior o superior del espacio anterior, se compara nuevamente con el elemento del centro, y así sucesivamente hasta que se encuentre el elemento o el espacio de búsqueda se haya reducido un elemento.

En la búsqueda binaria el número de elementos que conforman el campo de búsqueda se reduce a la mitad en cada iteración, siendo: n para la primera iteración (n = tamaño del vector), n/2 para la segunda, $n/2^2$ en la tercera y $/2^3$ para la cuarta.

En general, el espacio de búsqueda en la i-ésima iteración está conformado por n/2i-1 elementos.

Supóngase un vector v de 20 elementos ordenado de forma ascendente, como se muestra en la figura 114, en el que se busca el número 10. La primera iteración toma como espacio de búsqueda el vector completo y se ubica en el elemento de la mitad.

Para determinar cuál es elemento de la mitad se suma el índice del extremo inferior con el índice del extremo superior del espacio de búsqueda y se divide para dos.

(1 + 13) / 2 = 7

Se averigua si la clave está en el elemento del centro: clave = v[7]? 10 = 14?

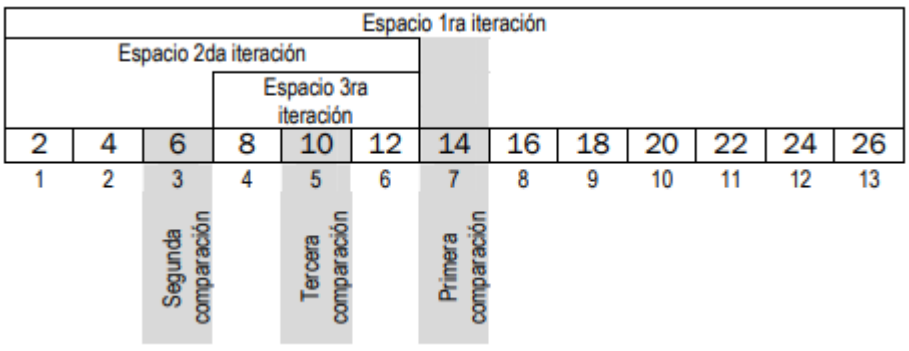

Figura 12. Diagrama de flujo del algoritmo de búsqueda lineal

Como el dato no está en posición 7 se define un nuevo espacio de búsqueda correspondiente a la mitad inferior del vector, dado que la clave (clave = 10) es menor que el elemento del centro (v[7] = 14). El espacio para la segunda comparación está entre las posiciones 1 y 6.

Se calcula nuevamente el centro:
(1 + 6) / 2 = 3
Ahora el elemento central es el de la posición 3, se averigua si ese es el dato que se busca:

clave = v[3]?
10 = 6?

Como en esta posición tampoco está el dato buscado se hace una nueva búsqueda, esta vez se toma la mitad superior del espacio, ya que la clave es mayor que el dato del centro: 10 > 6.
El nuevo espacio de búsqueda comprende los elementos entre las posiciones 4 y 6. Se calcula nuevamente el elemento del centro:

(4 + 6) / 2 = 5

Se compara la clave con el elemento de la posición central

clave = v[5]?
10 = 10?

Los valores son iguales, en esta iteración el dato fue encontrado. La búsqueda termina. Fueron necesarias 3 comparaciones para encontrar el elemento.

La función para realizar la búsqueda binaria recibe como parámetros el vector con los datos, el tamaño del vector y el dato que se busca o clave de búsqueda. El pseudocódigo de ésta se muestra en el cuadro 15

```
1   Entero busquedabinaria(entero v[], entero n, entero clave)
2       Entero inf = 1, sup = n, centro, pos = -1
3       Mientras inf <= sup y pos < 0 hacer
4           centro = (inf + sup) / 2
5           Si v[centro] = clave entonces
6               pos = centro
7           Si no
8               Si clave > v[centro] entonces
9                   inf = centro + 1
10              Si no
11                  sup = centro – 1
12              Fin si
13          Fin si
14      Fin mientras
15      Retornar pos
16  Fin busquedabinaria
```

Figura 13. Pseudocódigo de la función búsqueda binaria

Complejidad y Eficiencia

Contando Comparaciones

1: Mejor Caso: • El elemento buscado está en el centro. Por lo tanto, se hace una sola comparación

Log(n): Peor Caso: • El elemento buscado está en una esquina. Necesitando log2(n) cantidad de comparaciones

log(n/2): En Promedio: • Serán algo como log2(n/2)

Por lo tanto, la velocidad de ejecución depende logarítmicamente del tamaño del arreglo

Búsqueda en listas enlazadas simples

La lista ordenada de números enteros dada anteriormente (17, 26, 31, 54, 77 y 93) puede ser representada por una estructura enlazada como se muestra en la siguiente figura. De nuevo, la estructura de nodo y enlace es ideal para representar el posicionamiento relativo de los ítems.

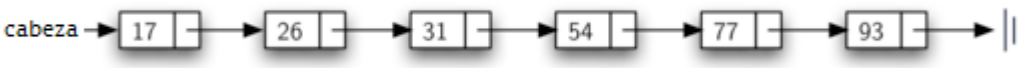

Figura 14: Una lista enlazada ordenada

La búsqueda en una lista enlazada no ordenada requería que recorrieran los nodos de uno en uno hasta encontrar el ítem que buscábamos o quedarnos sin nodos (None). Resulta que el mismo enfoque realmente funciona con la lista ordenada y de hecho, en el caso en que encontramos el ítem, es exactamente lo que necesitamos. Sin embargo, en el caso en que el ítem no esté en la lista, podemos aprovechar el ordenamiento para detener la búsqueda tan pronto como sea posible.

Por ejemplo, la Figura 16 muestra la lista enlazada ordenada a medida que se busca el valor 45. A medida que avanzamos, comenzando en la cabeza de la lista, primero comparamos contra 17. Dado que 17 no es el ítem que estamos buscando, nos movemos al siguiente nodo, en este caso 26. De nuevo, este valor no es el que queremos, así que pasamos a 31 y luego a 54. Ahora, en este punto, algo es diferente. Puesto que 54 no es el ítem que buscamos, nuestra estrategia anterior sería seguir adelante. Sin embargo, debido al hecho de que se trata de una lista ordenada, continuar ya no será necesario.

Una vez que el valor en el nodo sea mayor que el ítem que estamos buscando, la búsqueda puede detenerse y devolver False. No hay manera de que el ítem pueda existir más adelante en la lista enlazada.

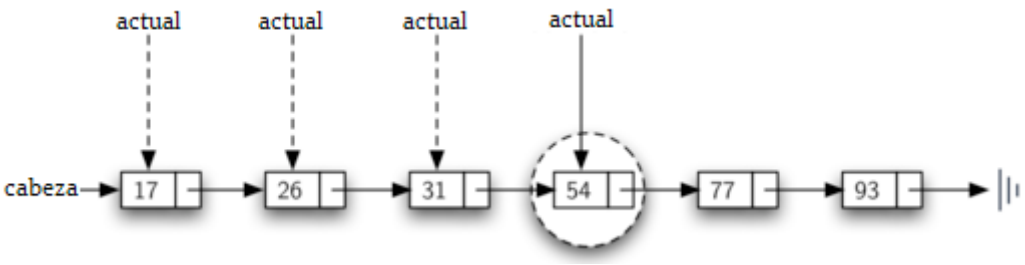

Figura 15: Búsqueda en una lista enlazada ordenada

Supongamos que tenemos la lista ordenada compuesta por los números 17, 26, 54, 77 y 93 y que queremos agregar el valor 31. El método agregar debe decidir que el nuevo ítem debe estar entre 26 y 54. La Figura 18 muestra la configuración que necesitamos. Como explicamos anteriormente, necesitamos recorrer la lista enlazada buscando el lugar donde se agrega el nuevo nodo. Sabemos que hemos encontrado ese lugar cuando nos quedamos sin nodos (actual se convierte en none) o el valor del nodo actual llega a ser mayor que el ítem que deseamos agregar. En nuestro ejemplo, ver el valor 54 nos detendrá.

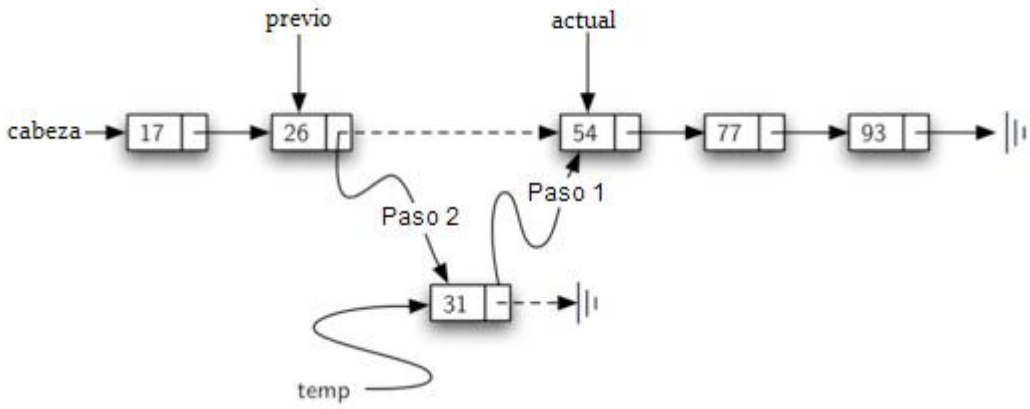

Figura 16: Agregar un ítem a una lista enlazada ordenada

Complejidad en las listas Enlazadas Simples

Para analizar la complejidad de las operaciones de lista enlazadas, necesitamos considerar si se requiere recorrerlas. Considere una lista enlazada que tiene n nodos. El método está vacío, es

- **O(1)** ya que requiere un paso para comprobar si la referencia de la cabeza es none, tamaño por otro lado, siempre requerirán pasos ya que no hay forma de saber cuántos nodos hay en la lista enlazada sin recorrerla desde la cabeza hasta el final. Por lo tanto, tamaño es
- **O(n)**. Agregar un ítem a una lista no ordenada siempre será
- **O(1)** ya que simplemente colocamos el nuevo nodo en la cabeza de la lista enlazada. Sin embargo, buscar y remover, así como agregar para una lista ordenada, requieren el proceso de recorrido. Aunque en promedio pueden necesitar recorrer sólo la mitad de los nodos, estos métodos son todos
- **O(n)** ya que en el peor de los casos procesarán cada nodo de la lista.

Búsqueda en listas enlazadas dobles

Las listas enlazadas dobles son un tipo de lista lineal en la que cada nodo tiene dos enlaces, uno que apunta al nodo siguiente, y el otro que apunta al nodo anterior. Las listas doblemente enlazadas no necesitan un nodo específico para acceder a ellas, ya que presentan una gran ventaja comparada con las listas enlazadas y es que pueden recorrer en ambos sentidos a partir de cualquier nodo de la lista, ya que siempre es posible desde cualquier nodo alcanzar

cualquier otro nodo de la lista, hasta que se llega a uno de los extremos. La representación gráfica de la lista enlazada doble se muestra de la siguiente manera:

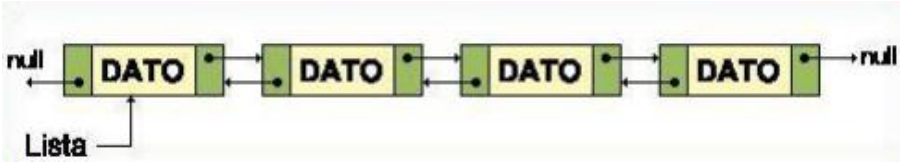

Figura 17: Nodos de una lista enlazada doble

BUSCAR O RECORRER

Para recorrer una lista procederemos de un modo parecido al que usábamos con las listas abiertas, ahora no necesitamos un puntero auxiliar, pero tenemos que tener en cuenta que Lista no tiene por qué estar en uno de los extremos:

Retrocedemos hasta el comienzo de la lista, asignamos a lista el valor de lista->anterior mientras lista->anterior no sea NULL.

Abriremos un bucle que al menos debe tener una condición, que el índice no sea NULL.

Dentro del bucle asignaremos a lista el valor del nodo siguiente al actual.

Por ejemplo, para mostrar todos los valores de los nodos de una lista, podemos usar el siguiente bucle en C:

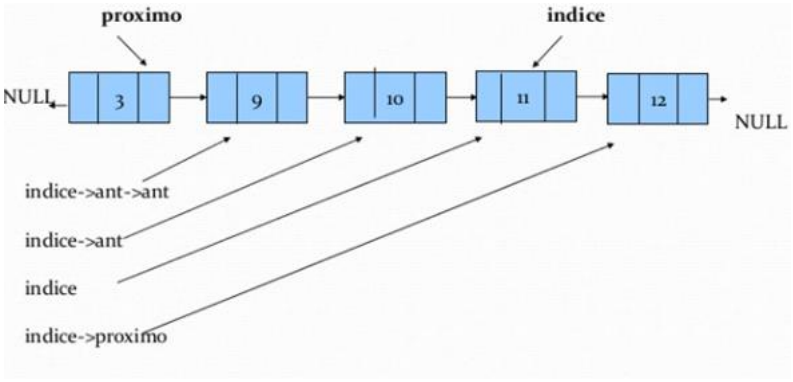

Figura 18: Lista enlazada doble

Búsqueda en Arboles binarios

Una búsqueda más eficiente puede hacerse sobre un arreglo ordenado. Una de éstas es la Búsqueda Binaria. La Búsqueda Binaria, compara si el valor buscado está en la mitad superior o inferior. En la que está, subdivido nuevamente, y así sucesivamente hasta encontrar el valor.

Ejemplo:

Supuesto: Arreglo con datos ordenados en forma ascendente: i<k a[i]<a[k]

• Estamos buscando la posición del valor 17

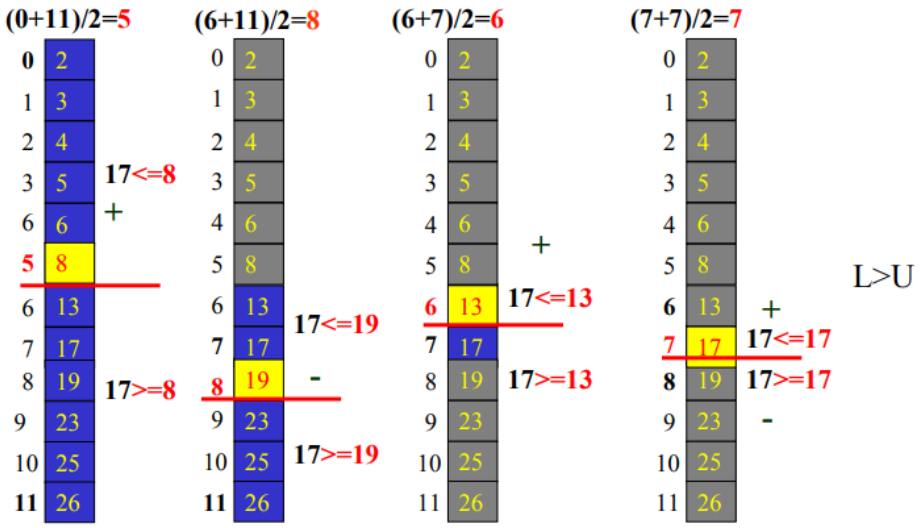

Figura 19: Búsqueda binaria

- Mejor Caso: El elemento buscado está en el centro. Por lo tanto, se hace una sola comparación O(1).
- Peor Caso: El elemento buscado está en una esquina. Necesitando log(n) cantidad de comparaciones
- En Promedio: Serán algo como log(n/2)

Por lo tanto, la velocidad de ejecución depende logarítmicamente del tamaño del arreglo

Búsqueda en Árboles RB

Un árbol rojo-negro es un tipo especial de árbol binario usado en informática para organizar información compuesta por datos comparables (por ejemplo, números). En los árboles rojo-negro las hojas no son relevantes y no contienen datos.

En los árboles rojo-negro, como en todos los árboles binarios de búsqueda, es posible moverse ordenadamente a través de los elementos de forma eficiente si hay forma de

localizar el padre de cualquier nodo. El tiempo de desplazarse desde la raíz hasta una hoja a través de un árbol equilibrado que tiene la mínima altura posible es de O(log *n*).

- Caso promedio:

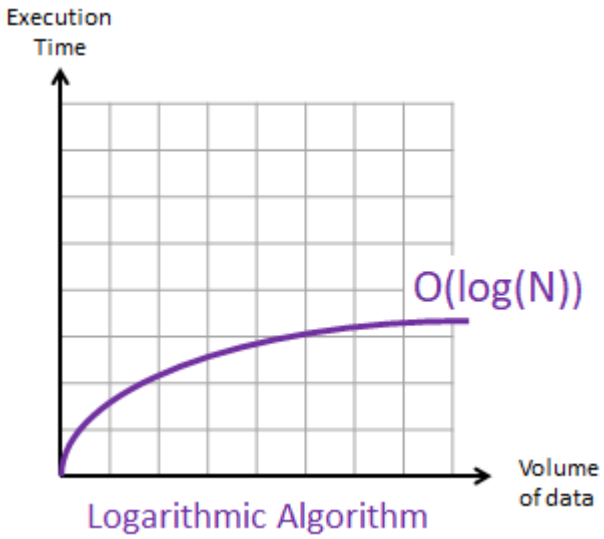

Figura 20: Complejidad O(log (N))

- Peor caso:

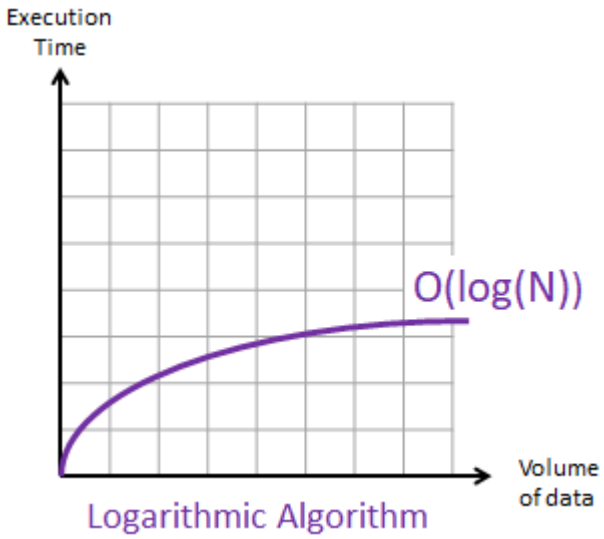

Figura 21: Complejidad O(log (N))

Búsqueda en Árboles AVL

Los árboles AVL se mantienen equilibrados de tal modo que, para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha o viceversa. Esta forma de balanceo, la complejidad de una búsqueda en uno de estos árboles se mantiene siempre en orden de complejidad O(log n).

- Caso promedio:

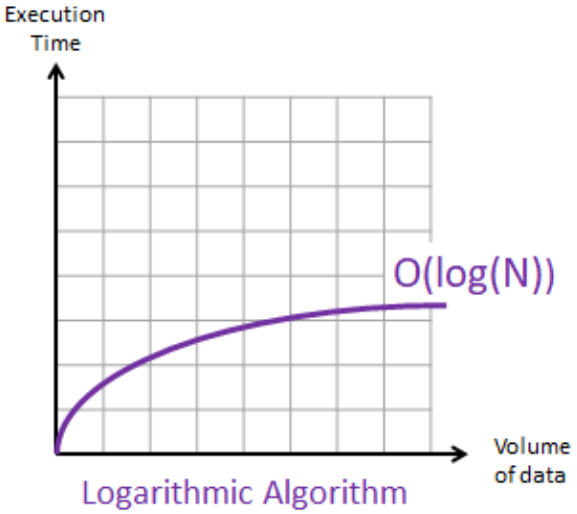

Figura 22: Complejidad O(log (N))

- Peor caso:

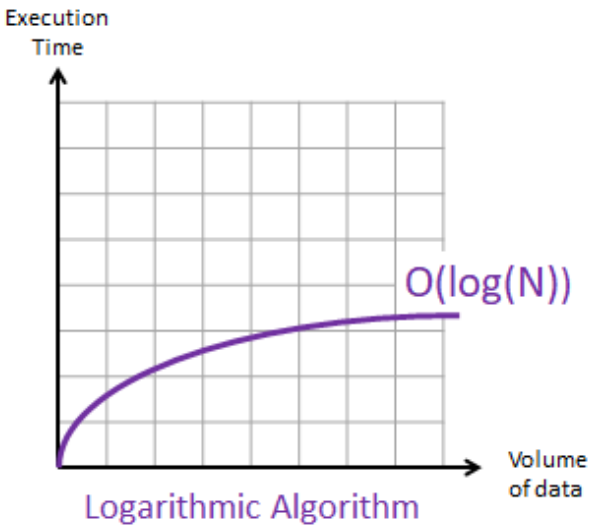

Figura 23: Complejidad O(log (N))

Búsqueda en Tablas Hash

Es una estructura de datos que vincula claves con valores. La operación principal que soporta de manera eficiente es la búsqueda: permite el acceso a los elementos (teléfono y dirección, por ejemplo) almacenados a partir de una clave generada, transforma la clave con una función hash que genera un número que identifica la posición (casilla o cubeta) donde la tabla hash localiza el valor deseado.

- Caso promedio:

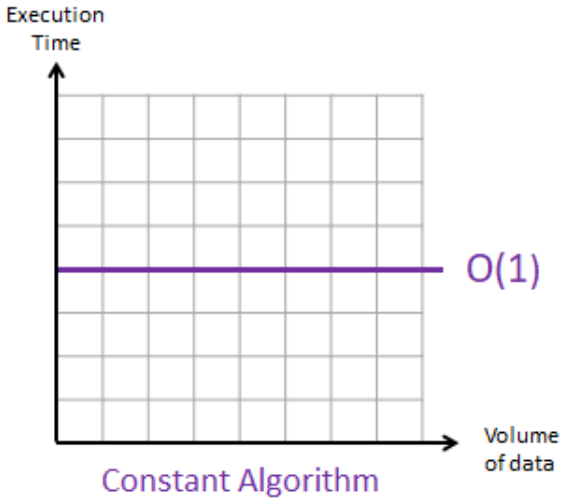

Figura 24: Complejidad O(1)

- Peor caso:

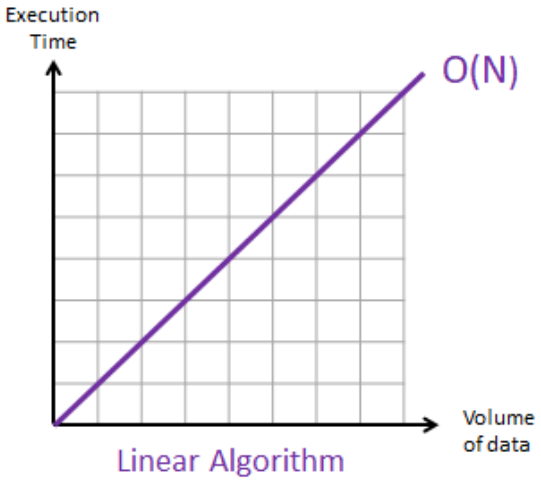

Figura 25: Complejidad O(N)

En el caso de Binary Heap, Binomial Heap y Fibonacci Heap su búsqueda es ineficiente, ya que llevaría un tiempo encontrar un nodo con una clave dada, por eso en este caso, no será una búsqueda en si, sino seria encontrar la clave mínima.

Búsqueda en Binary Heap

Un montón binario está diseñado para que el elemento mínimo pueda ser localizado en tiempo constante porque es el primer elemento del montón. Esto quiere decir que su tiempo de complejidad se mantiene en O(1).

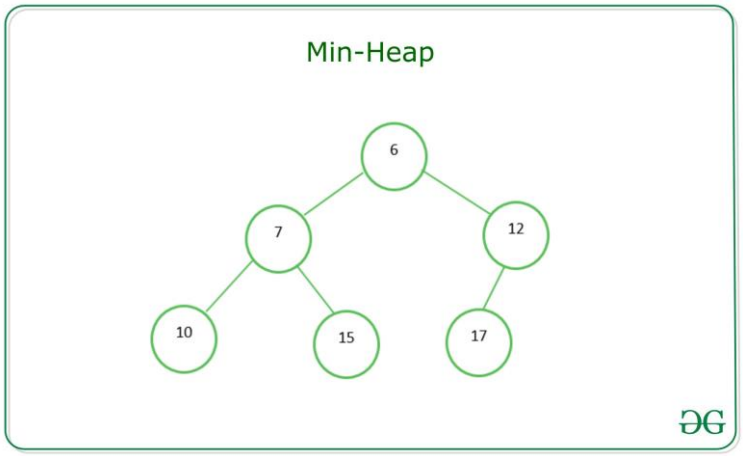

Figura 26: Minimum Binary Heap

Búsqueda en Binomial Heap

Para encontrar el mínimo, es simple ya que solo retorna el puntero al nodo del monton H cuya clave es mínima. El tiempo de complejidad se mantiene en O(lg n).

```
BINOMIAL-HEAP-MINIMUM(H)
1  y ← NIL
2  x ← head[H]
3  min ← ∞
4  while x ≠ NIL
5        do if key[x] < min
6              then min ← key[x]
7                   y ← x
8           x ← sibling[x]
9  return y
```

Figura 27: Function BINOMIAL-HEAP-MINIMUM

El procedimiento BINOMIAL-HEAP-MINIMUM comprueba todas las raíces, que son como máximo lg n + 1, guardando el mínimo actual en min y un puntero al mínimo actual en y. Cuando se llama en el montón binomial de la figura mostrada, BINOMIAL-HEAP-MINIMUM devuelve un puntero al nodo con la clave 1. Como hay como máximo lg n + 1 raíces que comprobar, el tiempo de ejecución de BINOMIAL-HEAP-MINIMUM es O(lg n).

Búsqueda en Fibonacci Heap

La operación Encontrar Mínimo es trivial porque guardamos el puntero al nodo que lo contiene. Esto no cambia el Potencial del Montículo, ya que el coste actual y amortizado es constante. Quiere decir que su tiempo de complejidad se mantiene en O(1).

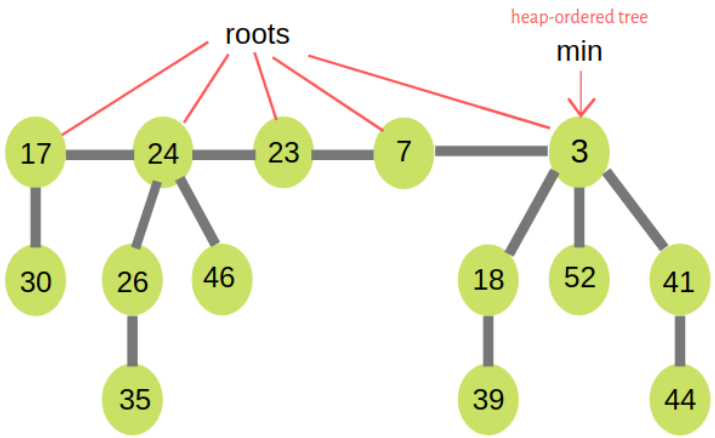

Figura 28: Fibonacci Heaps

Cuadro de comparaciones de la complejidad algorítmica de búsqueda

Estructuras de datos	Complejidad algorítmica en Búsqueda de una elemento		
	Peor Caso	Caso Promedio	Mejor Caso
Vectores	***O(n/2) || O(logn n/2)***	***O(n) || O(log n)***	***O(1)***
Lista enlazada Simple	***O(n)***	***O(n)***	***O(1)***
Lista enlazada Doble	***O(n)***	***O(n)***	***O(1)***
Arboles binarios	***O(log n)***	***O(logn n/2)***	***O(1)***
Árboles RB	***O(log n)***	***O(log n)***	***O(log n)***
Arboles AVL	***O(log n)***	***O(log n)***	***O(log n)***
Tablas Hash	***O(n)***	***O(1)***	***O(1)***
Binary Heap	***O(1)***	***O(1)***	***O(1)***
Binomial Heap	***O(log n)***	***O(log n)***	***O(log n)***
Fibonacci Heap	***O(1)***	***O(1)***	***O(1)***

Figura 29: Cuadro comparativo de las estructuras

CONCLUSIONES

Gran parte de la eficiencia está determinada por el tiempo de ejecución, es decir, la complejidad computacional de la solución, aunque aproximadamente una mínima parte está determinada por un patrón de consumo de energía.

El análisis en los distintos lenguajes proporciona la información básica para elegir una implementación de clasificación para hacer más eficiente un programa o acomodarnos más a un lenguaje.

¿Se puede elegir un lenguaje de programación basado en la energía, tiempo y memoria? Si. “C” es el lenguaje que más destaca analizando los 3 campos.

Pero no solo debemos de tener una perspectiva de los lenguajes, unos cumplen una función mejor que otras, en cuanto a paradigmas como POO con java en el apogeo por su eficiencia energética y en rapidez.

Como trabajo adicional se planea investigar con más detalle los factores de hardware que afectan el consumo de energía, memoria y complejidad, cómo están relacionados con distintos lenguajes de programación.

BIBLIOGRAFÍA

- [1] BROOKSHEAR J. Glean. Teoría de la computación. Addisson Wesley Iberoamericana Wilmington Delaware 1993.
- [2] PeñA M., Ricardo. Diseño de programas. Formalismo y abstracción. Prentice Hall Madrid 1998.
- [3] Colaboradores de Wikipedia. El juego de comparativas del lenguaje informático Wikipedia, La enciclopedia libre, 10 de octubre de 2021. Disponible en: https://en.wikipedia.org/wiki/The_Computer_Language_Benchmarks_Game
- Jose D. L.*Lenguajes compilados y lenguajes interpretados. (2017).*ED Team .https://ed.team/blog/lenguajes-compilados-vs-lenguajes-interpretados
- [4] Mohammad Rashid, Luca Ardito, Marco Torchiano. Energy Consumption Analysis of Algorithms Implementations. 2015.
- Fundamentos de Informática y programación. Disponible en: http://informatica.uv.es/docencia/fguia/TI/Libro/PDFs/CAPI7.pdf
- Jose M. Rendimiento de Algoritmos y notación Big-O. Campus MVP. (2016, 14 de junio). Disponible en: https://www.campusmvp.es/recursos/post/Rendimiento-de-algoritmos-y-notacion-Big-O.aspx
- Teddy A. Algoritmo de búsqueda y ordenamiento. Departamento de informática. Universidad Técnica Federico Santa María. Disponible en: https://www.inf.utfsm.cl/~noell/IWI-131-p1/Tema8b.pdf
- H. Cormen Thomas, E. Leiserson Charles, L.Rivest Ronald y S. Clifford.(2009) Introduction to Algorithms.MIT Press. Disponible en: https://books.google.com.pe/books?id=aefUBQAAQBAJ&printsec=frontcover&dq=Introduction+To+Algorithms&hl=es&sa=X&redir_esc=y#v=onepage&q=Introduction%20To%20Algorithms&f=false

ANEXOS