

Análisis de complejidad computacional

¿Qué es un algoritmo?

- Informalmente, un algoritmo es cualquier procedimiento computacional bien definido que toma algún valor, o un conjunto de valores como entrada y produce algún valor, o conjunto de valores como salida.
- Un algoritmo es por lo tanto una secuencia de pasos computacionales que transforma la entrada en la salida.
- También podemos ver un algoritmo como una herramienta para resolver un problema computacional bien especificado.

Características de un algoritmo

- Un algoritmo debe de poseer las siguientes características:
 - **Precisión:** Un algoritmo debe expresarse sin ambigüedad.
 - **Determinismo:** Todo algoritmo debe de responder del mismo modo ante las mismas condiciones.
 - **Finito:** La descripción de un algoritmo debe de ser finita.

Cualidades de un algoritmo

- Un algoritmo debe de ser además:
 - **General:** Es deseable que un algoritmo sea capaz de resolver una clase de problemas lo más amplia posible.
 - **Eficiente:** Un algoritmo es eficiente cuantos menos recursos en tiempo, espacio (de memoria) y procesadores consume.

Problema computacional

- Un problema computacional es un problema que una computadora podría resolver o una pregunta que una computadora podría responder.
- Un problema computacional puede verse como una colección infinita de instancias junto con un conjunto de soluciones, posiblemente vacío, para cada instancia.
- **Instancia:** Una sucesión finita de números enteros (a_1, a_2, \dots, a_n)
- **Solución:** Una permutación $(a'_1, a'_2, \dots, a'_n)$ de la sucesión de entrada tal que $a'_1 \leq a'_2 \leq \dots \leq a'_n$

- El campo de la teoría de la complejidad computacional intenta determinar la cantidad de recursos (complejidad computacional) que requerirá resolver un problema dado y explicar por qué algunos problemas son intratables o indecidibles.
- Los problemas computacionales pertenecen a clases de complejidad que definen ampliamente los recursos:
 - Tiempo
 - espacio / memoria
 - Energía
 - profundidad del circuito


que se necesitan para calcularlos (resolverlos) con varias máquinas abstractas.

Tipos de problemas computacionales

- **Problema de decisión.**
 - Si
 - No
- **Problemas de búsqueda.** Las respuestas pueden ser cadenas arbitrarias.
 - Se representa como una relación que consta de todos los pares instancia-solución.
 - $P(i, s)$ determina si s es una solución de i .
- **Problema de conteo** pide el número de soluciones a un problema de búsqueda dado.
- **Problema de optimización**
 - No solo se busca una solución, sino que se busca "*la mejor*" de todas.
- En un **problema de función** se espera una única salida (de una función total) para cada entrada, pero la salida es más compleja que la de un problema de decisión, es decir, no es solo "sí" o "no".

Complejidad algorítmica

- Si dos algoritmos diferentes resuelven el mismo problema entonces los llamamos algoritmos equivalentes.
- La complejidad algorítmica permite establecer una comparación entre algoritmos equivalentes para determinar en forma teórica, cuál tendrá mejor rendimiento en condiciones extremas y adversas.

- 
- Para esto se trata de calcular cuántas instrucciones ejecutará el algoritmo en función del tamaño de los datos de entrada.
 - Llamamos “instrucción” a la acción de asignar valor a una variable y a la realización de las operaciones aritméticas y lógicas.

- Como resultaría imposible medir el tiempo que demora una computadora en ejecutar una instrucción, simplemente diremos que cada una demora 1 unidad de tiempo en ejecutarse.
- Luego, el algoritmo más eficiente será aquel que requiera menor cantidad de unidades de tiempo para concretar su tarea.

Ejemplo. Búsqueda secuencial de un elemento dentro de un *array* de longitud len.

```
int busquedaSecuencial(int arr[], int len, int v)
{
    int i = 0;
    while( i<len && arr[i]!=v )
    {
        i = i + 1;
    }
    return i<len?i:-1;
}
```

Ejemplo. Búsqueda secuencial de un elemento dentro de un *array* de longitud `len`.

```
int busquedaSecuencial(int arr[], int len, int v)
{
    int i = 0;
    while( i < len && arr[i] != v )
    {
        i = i + 1;
    }
    return i < len ? i - 1;
}
```

4

$$1 + 4 + 2 = 7$$

2

Ejemplo. Búsqueda secuencial de un elemento dentro de un *array* de longitud *len*.

```
int busquedaSecuencial(int arr[], int len, int v)
```

```
{
```

```
    int i = 0;
```

```
    while( i<len && arr[i]!=v )
```

```
    {
```

```
        i = i + 1;
```

```
    }
```

```
    return i<len?i:-1;
```

```
}
```

4

Ciclo While

2

Donde *n* es la cantidad de iteraciones que realiza el *while*

Instrucción	$i = i + 1$	$i < len \ \&\& \ arr[i] \neq v$
Unidades de tiempo	$2n$	$4n$

Ejemplo. Búsqueda secuencial de un elemento dentro de un *array* de longitud *len*.

$$f(n) = 6n + 7$$

- Si el elemento que buscamos se encuentra en la primera posición del *array* entonces el algoritmo no ingresará al *while* y solo requerirá 7 unidades de tiempo. **Mejor de los casos.**
- **Peor de los casos.** Si el valor que buscamos no existe en el *array* o cuando se encuentre en la última posición.

7 unidades de tiempo fijas + $6n$, siendo n igual a *len*