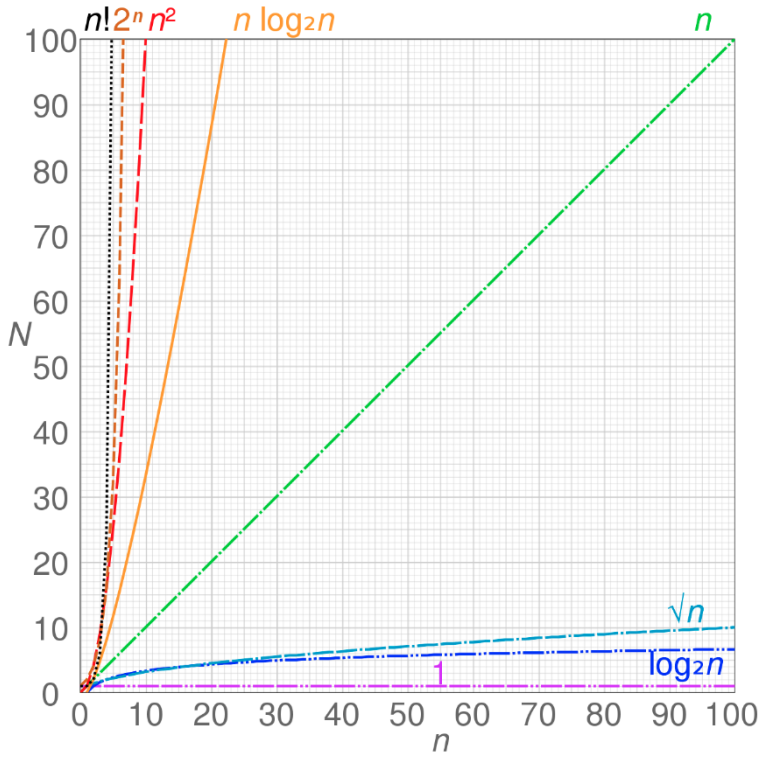
Análisis de algoritmos

- Cómo medir la velocidad de un algoritmo --> Análisis de las instrucciones del algoritmo en base a los datos con los que está trabajando.
- Notación **Big O** o crecimiento de funciones: nos indica qué tan rápido es un algoritmo,

No tiene en cuenta los coeficientes (desviaciones), sólo el factor dominante (elemento que más crece).

O(n) *siendo n el número de elementos*



ESTRUCTURAS DE DATOS (JAVA)

- **Arrays estáticos**
  - Almacena de forma contigua los valores en la memoria.
  - Rellena todas las posiciones con un valor por defecto (int con 0, objetos con null, ...) definidos por Java
  - ADT (operaciones):
    - Crear(tamaño)
      - Coste O(N). Recorre cada posición de la memoria con el valor por defecto
    - EstablecerValorEn(valor, posición)
      - Coste O(1)
    - Obtener(posición)
      - A través de la posición de memoria donde empieza el array, y el tamaño de cada objeto, accede a la posición
      - Coste O(1)
    - Búsqueda

        - Coste O(N), porque en el peor caso estará en la última posición
    - Ordenar
        - Coste O(NLogN)
        - Arrays.sort(array)
    - Búsqueda (en array ordenado)
        - Coste O(LogN)
        - Arrays.binarySearch(array, key)
    - Copiar un rango
        - Arrays.copyOfRange(array, desde incluido, hasta no incluido)
        - Coste O(N), en el peor caso copia todas las posiciones

- **Arrays dinámicos** (ArrayList)
  - Son un envoltorio a un array estático, y hace operaciones sobre éste para modificar su tamaño de forma óptima.

    Internamente son una serie de operaciones para crear y borrar (copia) arrays estáticos en base a nuestras operaciones
  - ADT
    - Crear()
        - Por defecto, array vacío con capacidad de 10. Coste: O(N)
    - Agregar(elemento)
        - Primero asegura capacidad interna del array. Si sobrepasa, crea un nuevo array con mayor capacidad (+/- el doble) en el que copia todos los elementos del viejo.
        - Amortizar coste de inserción -> O(1) = establecer
        - Coste O(N) en el peor caso
    - Agregar(elemento, posición)
        - Coste O(N) en el peor caso, insertando en 0
    - Borrar(posición)
        - Coste O(N) igual que el caso anterior
    - Obtener(posición)
        - Coste O(1) = estáticos
    - Establecer(elemento, posición)
        - Coste O(1)

- **Listas ligadas**
  - Es una forma de crear una lista con los datos, de tal manera que un dato está enlazado con el siguiente.
  - Datos no contiguos. Guarda dato y la referencia (enlace) al siguiente en una posición de memoria.
  - Clase Nodo genérica con Dato y Nodo al Siguiente
  - Nodo principio apuntando a Nodo A y un final apuntando al Nodo final (Siguiente = Null)
  - ADT:
    - Insertar al principio

- Coste O(1) ---> Nuevo Nodo B -> B.siguiente = Principio -> Principio = B
- Insertar al final
  - Coste O(1) con lista vacía --> Nuevo Nodo A -> Principio y Final apuntando a A
  - Coste O(1) en lista con objetos --> Nuevo Nodo B -> Final.sig = B -> Final = B
- Insertar en posición
  - Coste: O(N)
    - Nuevo Nodo, sig = null
    - Recorrer lista con dos nodos: Anterior mantendrá nodo anterior y Actual en nodo que se está comprobando si es el buscado
    - Anterior.sig = Nuevo
    - Nuevo.sig = Actual
- Eliminar al principio
  - Coste O(1) ---> Principio = Nodo A.sig
- Eliminar en posición o al final
  - Coste O(N) ---> igual que Insertar en posición -> Anterior.sig = Actual.sig
- Obtener elemento del principio
  - Coste O(1)
- Obtener elemento del final
  - Coste O(1)
- Acceso a posición
  - Coste O(N) -> Recorrer al menos todo. Posición de memoria desconocida

- **Lista ligada doble**
  - Cada Nodo tiene dos enlaces, al siguiente y al anterior.
  - No necesario Nodo Anterior y Actual para operar.

  En Java -> clase LinkedList

- **Pilas (Stack)**
  - LIFO (Last in, Last out)
  - Implementar acción deshacer (apilando acciones) o búsqueda en profundidad
  - Guardar llamadas de las funciones recursivas en el orden en que se deben ejecutar.
  - ADT: Crear, Apilar (push), Desapilar (pop), Ver la cima, Tamaño, Ver si está vacía

  - Implementar pila con lista ligada. ADT:
    - Apilar/Desapilar -- Insertar/Eliminar (SE QUITA) al principio. Coste O(1)
    - Ver la cima Coste O(1)

  - Implementar pila con array dinámico
    - Cima variable top inicializada a -1
    - Apilar: top++ --> Insertar elemento en top. Coste O(1)
    - Desapilar: top-- Coste O(1)
    - Ver la cima con top. Coste O(1)

- **Colas(Queue)**
  - FIFO (First in, First Out)
  - Transferir datos entre procesos asíncronos, cola de entrada para servidor, búsqueda de anchura...
  - ADT: crear, encolar, desencolar, "" = pila

  - Implementar cola con lista ligada.
    - Encolar -- Insertar al final. Coste O(1)
    - Desencolar -- Eliminar (SE QUITA) al inicio. Coste O(1)
    - Ver al frente con top. 1er elemento lista. Coste O(1)

- **Montículos**
  - Retiramos valores de forma ordenada (el array no debe estar necesariamente ordenado) con un coste bajo
  - Es un tipo de árbol binario, tiene un nodo y este dos hijos (puede ser altura n)
  - Dos tipos: max (orden mayor a menor, padre > hijos) y min (p < h)
  - Ha de rellenarse de izq a der sin dejar ningún nodo en medio sin hijos
  - Implementar algoritmo de Prim, de Dijkstra, lista ordenada de inserción muy rápida y eficiente
  - ADT de una implementación cola de prioridad:
    - Crear: array dinámico
    - Insertar elem: última pos array. Método flotar -> mover elem a pos correspondiente
      Acceder a padre a O(1) -> si k>0, padre está en pos (k-1)/2
      Nodos = 2^(altura+1)- 1 == Altura = Log2(Nodos) --> Altura = Log2(lista.size) --> Coste O(LogN)
    - Retirar elemento max/min: quitamos 1er elem, el último se coloca en la 1a pos y usamos método hundir: comprobación con hijos más peq/may -> (2*k)+1 hijo izq, (2*k)+2 hijo der --> O(LogN)

  - HeapSort: algoritmo de ordenamiento que se basa simplemente en ir retirando todos los elementos de un montículo (heap) e irlo insertando por ese orden en el array
    - Coste O(NLogN)