Curso Introductorio al Análisis de Algoritmos con Julia

Eric S. Téllez

Tabla de contenidos

Prefacio

El *Análisis de algoritmos* es una disciplina formativa enfocada en el desempeño de los algoritmos bajo una cierta entrada. Su estudio nos permite identificar el problema algorítmico subyacente dentro de problemas reales, y por tanto, ser capaces de seleccionar, adaptar o construir una solución eficiente y eficaz para dicho problema. Una solución adecuada sobre una ingenua nos permite mejorar de manera significativa los recursos computacionales, que pueden llevar a reducción de costos de operación en un sistema o la posibilidad de procesar grandes cantidades de información de manera más eficiente.

El diseño, implementación y análisis de algoritmos es fundamental para formar el criterio del científico de datos. Los conocimientos adquiridos servirán para obtener las herramientas y la intuición necesaria para plantear la solución a un problema basado en un modelo de cómputo y resolverlo de manera eficiente y escalable cuando sea posible.

A lo largo de los temas se abordarán los algoritmos y estructuras de manera teórica y práctica, y se motivará al estudiante a realizar sus propias implementaciones. Al terminar este curso, se pretende que el alumno sea competente para seleccionar, diseñar, implementar y analizar algoritmos sobre secuencias, conjuntos y estructuras de datos para resolver problemas optimizando los recursos disponibles, en particular, memoria y tiempo de cómputo. Durante el curso se estudiaran problemas y algoritmos simples, que suelen formar parte de algoritmos más complejos, y por lo tanto, si somos capaces de seleccionar adecuadamente estos bloques más simples, afectaremos directamente el desempeño de los sistemas.

Contenido del libro

Este libro esta diseñado para ser impartido en un semestre de licenciatura o maestría con un enfoque experimental, de Ingeniería en Computación o Ciencias de la Computación, así como Ciencia de Datos. Los algoritmos que se van develando desentrañan los algoritmos clásicos de Recuperación de Información, algoritmos detrás de grandes máquinas de búsqueda, sistemas de información basados en similitud, *retrieval augmented generation* (RAG), así como de los métodos detrás de la aceleración de otras técnicas de análisis de datos como agrupamiento y reducción de dimensión no-lineal.

- El Cap. 1 se dedica a revisar el lenguaje de programación Julia, desde un punto de vista de alguien que podría no conocer el lenguaje, pero que definitivamente sabe programar y esta familiarizado con los conceptos generales de un lenguaje de programación moderno.
- El Cap. 2 introduce los conceptos de análisis asintótico y compara ordenes de crecimiento con la idea de formar intuición.
- En el Cap. 3 nos encontramos con las estructuras de datos elementales como son las estructuras de datos lineales y de acceso aleatorio, y su organización en memoria.
- El Cap. 4 esta dedicado a algoritmos de ordenamiento en el modelo de comparación, estudia algoritmos tanto de peor caso como aquellos que toman ventaja de la distribución de entrada.
- En el Cap. 5 abordamos algoritmos de búsqueda en arreglos ordenados en el modelo de comparación. De nueva cuenta se abordan algoritmos de peor caso y algoritmos que pueden sacar ventaja de instancias fáciles.
- Finalmente, el Cap. 6 estudia algoritmos de intersección de conjuntos, los cuales son la base de sistemas de información capaces de manipular cantidades enormes de datos.

Trabajo en progreso

Este libro es un trabajo en progreso, que se pretende términar durante el primer semestre de 2025, mientras se imparte el curso *Análisis de algoritmos* en la Maestría en Ciencia de Datos e Información de INFOTEC, México. El perfil de ingreso de la maestría es multidisciplinario, y esto es parte esencial del diseño de este libro.

En particular, los capítulos 1, 2, y 3 tienen un avance significativo, aunque no estan terminados. El resto de los capítulos ese encuentran en un estado incipiente.

Licencia

1 Julia como lenguaje de programación para un curso de algoritmos

Nuestro objetivo trabajar sobre algoritmos, por lo que cualquier lenguaje que pueda expresar todo lo computable, puede ser adecuado. Pero dado que nuestro enfoque será experimental, y nuestra metodología incluye medir la factibilidad y desempeño de cada algoritmo en términos reales, entonces necesitamos un lenguaje donde las instrucciones, los acceso a memoria, y la manipulación de la misma sea controlable. En este caso, y mediando con la fácilidad de aprendizaje y la productividad, este curso utiliza el lenguaje de programación Julia.[1] Pero no hay porque preocuparse por aprender un nuevo lenguaje, el curso utiliza ejemplos en Julia y utiliza una variante de su sintaxis como pseudo-código, pero las actividades se esperan tanto en Julia como en Python.

Ambos lenguajes de programación son fáciles de aprender y altamente productivos. Python es un lenguaje excelente para realizar prototipos, o para cuando existen bibliotecas que resuelvan el problema que se este enfrentando. Por otro lado, cuando se necesita control sobre las operaciones que se estan ejecutando, o la memoria que se aloja, Python no es un lenguaje que nos permita trabajar en ese sentido. Julia esta diseñado para ser veloz y a la vez mantener el dinámismo que se espera de un lenguaje moderno, adicionalmente, es posible conocer los tipos de instrucciones que realmente se ejecutan, así como también es posible controlar la alojación de memoria, ya se mediante la utilización de patrones que así nos lo permitan, o mediante instrucciones que nos lo aseguren.

[1] Se recomienda utilizar la versión 1.10 o superior, y puede obtenerse en https://julialang.org/.

Este curso esta escrito en Quarto, y se esperan reportes de de tareas y actividades tanto en Quarto https://quarto.org como en Jupyter https://jupyter.org/. La mayoría de los ejemplos estarán empotrados en el sitio, y en principio, deberían poder replicarse copiando, pegando, y ejecutando en una terminal de Julia.

Es importante clarificar que este capítulo introducirá el lenguaje de programación Julia hasta el nivel que se requiere en este curso, ignorando una gran cantidad de capacidades que no son de interés para nuestro curso. Se recomienda al alumno interesado la revisión del manual y la documentación oficial para un estudio más profundo del lenguaje.

1.1 El lenguaje de programación Julia

Julia es un lenguaje singular, es un lenguaje dinámico y de alto nivel, tiene de tipado fuerte y compila a código máquina para cada una de las instrucciones que se dan. Su interfaz más común es un REPL o *read, eval, print, loop*, esto es que puede ser utilizado de manera interactiva, además de la ejecución en scripts o notebooks como los que estaremos usando para reportar.

Es homoicónico, que significa que la manera en que se representan sus programas coincide con las estructuras de datos básicas, lo cual permite crear programas validos mediante programas. De manera práctica, también le permite la reescritura de los programas utilizando otro programa utilizando *macros*, los cuales son funciones que modifican el código y empiezan con el simbolo `@`. Estaremos viendo una serie de macros con propósitos muy específicos, crear macros y la manipulación automática de código cae fuera de nuestro curso.

El lenguaje tiene estructuras de datos básicas como rangos, vistas, tuplas, arreglos, estructuras, diccionarios, conjuntos, cadenas de caracteres, así como expresiones de código como datos y controla la ejecución mediante condicionales, ciclos y funciones. Tiene un sistema de tipos de datos muy poderoso, que le permite entre otras cosas generar código específico para dichos tipos. El código se organiza en scripts, y a nivel lógico en módulos y paquetes. Una de sus características importantes el *despacho*

múltiple en las funciones, esto es, que para cada conjunto de tipos de argumentos, compilará una función especializada. Este patrón puede ser muy poderoso para escribir código genérico que pueda ser muy eficiente, a costa de múltiples códigos de máquina para una función. Esta estrategía también viene con el problema que la primera vez que se ejecuta una función con un conjunto específico de tipos de argumentos, dicha función será especializada y compilada, lo cual puede representar un costo inicial importante en algunos casos donde no se pretenda procesar grandes cantidades de información. En particular, este problema se ha venido reduciendo en las versiones más nuevas de Julia haciendo uso una estrategía de precompilación para datos típicos.

Entre los tipos de datos es capaz de manera enteros y números de punto flotante de diferentes precisiones, caracteres, cadenas de caracteres, y simbolos. Los arreglos son realmente importantes en Julia, y soportan de manera nativa vectores, matrices y tensores, estaremos tocando apenas esta parte del lenguaje. El resto de esta unidad esta dedicada a precisar la sintaxis del lenguaje y anotaciones de importancia sobre su funcionamiento, y en particular, en el manejo que nos permitirá generar código eficiente que limite el alojamiento de memoria.

1.1.1 Funciones

Las funciones son centrales en Julia, y son definidas mediante la sintaxis

```{julia}
function nombre(arg1...)                                    ①
    ... expresiones ...
end

function nombre(arg1...; kwarg1=valor1...)                  ②
    ... expresiones ...
end

nombre(arg1, arg2...; kwarg1=valor1...) = expresion ③

(arg1, arg2...; kwarg1=valor1...) -> expresion              ④
```

```
fun() do x                                                    ⑤
    x^2
end
```

① Definición de una función simple, los tipos de los argumentos se utilizan para generar múltiples versiones de una función.
② También se soportan argumentos nombrados, los cuales van después de `;`, se debe tener en cuenta que los tipos de los argumentos nombrados no son utilizados para determinar si una función debe compilarse. Los argumentos nombrados pueden o no tener valores por omisión.
③ Si la función tiene una estructura simple, de una expresión, es posible ignorar `function` y `end`, usando '=' para definirla.
④ Muchas veces es útil definir funciones anónimas, que suelen pasarse a otras funciones de orden superior.
⑤ Un embellecedor útil para generar una función anónima (definida entre `do...end`) que se pasa como primer argumento a `fun`, e.g., es equivalente a `fun(x->x^2)`.

El *ámbito* o *scope* de las variables en Julia es sintáctico, que significa que se hereda del código donde las funciones fueron definidas, y no dinámico (que se hereda desde dónde se ejecuta la función). Aunque es el comportamiento de la mayoría de los lenguajes modernos, es importante conocerlo sobre todo para la creación de *cerraduras sintácticas* en funciones.

Una función se ejecuta con la sintaxis `nombre(arg1...)`. Conviene profundizar en las expresiones y demás componentes del lenguaje antes del ir a más ejemplos sobre funciones.

1.1.2 Hola mundo

Uno de los programas más comunes es el siguiente

```
println("¡Hola !")
```

```
¡Hola !
```

1.1.3 Expresiones y operadores

Las expresiones son la forma más genérica de expresar el código en Julia, comprenden operaciones aritméticas, asignación y declaración de variables, definiciones de bloques de código, llamadas de funciones, entre otras.

Cada linea suele ser una expresión, a menos que se extienda por múltiples lineas por medio de un agrupador de código o datos, estos pueden ser `begin...end`, `let...end`, `(...)`, `[...]`, `[...]`, `for...end`, `while...end`, `if...end`, `function...end`, `try...end`, entre las más utilizadas.

Las definiciones de variables tienen la sintaxis `variable = valor`; las variables comunmente comienzan con una letra o `_`, las letras pueden ser caracteres *unicode*, no deben contener espacios ni puntuaciones como parte del nombre; `valor` es el resultado de evaluar o ejecutar una expresión.

Los operadores más comunes son los aritméticos `+`, `-`, `*`, `/`, `÷`, `%`, `\`, `^`, con precedencia y significado típico. Existen maneras compuestas de modificar una variable anteponiendo el operador aritmético al simbolo de asignación, e.g., `variable += valor`, que se expande a `variable = variable + valor`. Esto implica que `variable` debe estar previamente definida previo a la ejecución.

Los operadores lógicos también tienen el significado esperado.

operación	descripción
`a && b`	AND lógico
`a || b`	OR lógico
`a  b`	XOR lógico
`!a`	negación lógica
`a < b`	comparación `a` es menor que `b`
`a > b`	comparación `a` es mayor que `b`
`a <= b`	comparación `a` es menor o igual que `b`
`a >= b`	comparación `a` es mayor o igual que `b`
`a == b`	comparación de igualdad
`a === b`	comparación de igualdad (a nivel de tipo)
`a != b`	comparación de desigualdad
`a !== b`	comparación de desigualdad (a nivel de tipo)

En particular `&&` y `||` implementan *corto circuito de código*, por lo que pueden usarse para el control de que operaciones se ejecutan. Cuando se compara a nivel de tipo 0 (entero) será diferente de 0.0 (real).

También hay operadores lógicos a nivel de bit, los argumentos son enteros.

operación	descripción
`a & b`	AND a nivel de bits
`a | b`	OR a nivel de bits
`a  b`	XOR a nivel del bits
`~a`	negación lógica a nivel de bits

1.1.4 Literales

Dado que existen múltiples tipos de datos existen diferentes formas de definirlas; una de ellas, probablemente la que más estaremos usando son los literales, es decir, escribir los datos directamente en el código.

Los números enteros se definen sin punto decimal, es posible usar _ como separador y dar más claridad al código. Los enteros pueden tener 8, 16, 32, o 64 bits; por omisión, se empaquetan en variables del tipo `Int` (`Int64`). Los valores hexadecimales se interpretan como enteros sin signo, y además se empaquetan al número de bits necesario minimo para contener. El comportamiento para valores en base 10 es el de hexadecimal es congruente con un lenguaje para programación de sistemas.

```
a = 100
println((a, sizeof(a)))
b = Int8(100)
println((b, sizeof(b)))
c = 30_000_000
println((c, sizeof(c)))
d = 0xffff
println((d, sizeof(d)))
```

```
(100, 8)
```

```
(100, 1)
(30000000, 8)
(0xffff, 2)
```

Si la precisión esta en duda o el contexto lo amérita, deberá especificarlo usando el constructor del tipo e.g., `Int8(100)`, `UInt8(100)`, `Int16(100)`, `UInt16(100)`, `Int32(100)`, `UInt32(100)`, `Int64(100)`, `UInt64(100)`.

Existen números enteros de precisión 128 pero las operaciones al día de hoy no son implementadas de manera nativa por los procesadores; así mismo se reconocen números de punto flotante de precisión media `Float16` pero la mayoría de los procesadores no tienen soporte nativo para realizar operaciones con ellos, aunque los procesadores de última generación si lo tienen.

Los números de punto flotante tienen diferentes formas de definirse, teniendo diferentes efectos. Para números de precision simple, 32 bits, se definen con el sufijo `f0` como `3f0`. El sufijo `e0` también se puede usar para definir precisión doble (64 bit). El cero del sufijo en realidad tiene el objetivo de colocar el punto decimal, en notación de ingeniería, e.g., 0.003 se define como $3f-3$ o $3e-3$, dependiendo del tipo de dato que se necesite. Si se omite sufijo y se pone solo punto decimal entonces se interpretará como precision doble. Los tipos son `Float32` y `Float64`.

Los datos booleanos se indican mediante `true` y `false` para verdadero y falso, respectivamente.

Los caracteres son símbolos para índicar cadenas, se suelen representar como enteros pequeños en memoria. Se especifican con comillas simples `'a'`, `'z'`, `'!'` y soporta simbolos *unicode* `' '`.

Las cadenas de caracteres son la manera de representar textos como datos, se guardan en zonas contiguas de memoria. Se especifican con comillas dobles y también soportan símbolos unicode, e.g., `"hola mundo"`, `"pato es un  "`.

En Julia existe la noción de símbolo, que es una cadena que además solo existe en una posición en memoria se usa el prefijo `:` para denotarlos.

Julia guarda los simbolos de manera especial y pueden ser utilizados para realizar identificación de datos eficiente, sin embargo, no es buena idea saturar el sistema de manejo de símbolos por ejemplo para crear un vocabulario ya que no liberará la memoria después de definirlos ya que es un mecánismo diseñado para la representación de los programas, pero lo suficientemente robusto y bien definido para usarse en el diseño e implementación de programas de los usuarios.

```
println(:hola === :hola)
println(typeof(:hola))
println(Symbol("hola mundo"))
```

```
true
Symbol
hola mundo
```

1.1.5 Control de flujo

El control de flujo nos permite escoger que partes del código se ejecutaran como consecuencia de la evaluación de una expresión, esto incluye repeticiones.

Las condicionales son el control de flujo más simple.

```
a = 10
if a % 2 == 0                                    ①
    "par"                                        ②
else
    "impar"                                      ③
end
```

① Expresión condicional.
② Expresión a ejecutarse si (1) es verdadero.
③ Expresión a evaluarse si (1) es falso.

```
"par"
```

Se puede ignorar la clausula `else` dando solo la opción de evaluar (2) si (1) es verdadero. Finalmente, note que la condicional es una expresión y devuelve un valor.

```
a = 10
if log10(a) == 1                                 ①
    "es 10"                                      ②
end
```

```
"es 10"
```

También pueden concatenarse múltiples expresiones condicionales con `elseif` como se muestra a continuación.

```
a = 9
if a % 2 == 0
    println("divisible entre 2")
elseif a % 3 == 0
    println("divisible entre 3")
else
```

```
    println("no divisible entre 2 y 3")
end
```

```
divisible entre 3
```

Es común utilizar la sintaxis en Julia (short circuit) para control de flujo:

```
a = 9

println(a % 2 == 0 && "es divisible entre dos")    ①
println(a % 3 == 0 && "es divisible entre tres")   ②
```

① El resultado de la condición es falso, por lo que no se ejecutará la siguiente expresión.
② El resultado es verdadero, por lo que se ejecutará la segunda expresión.

```
false
es divisible entre tres
```

Fnalmente, existe una condicional de tres vias `expresion ? expr-verdadero : expr-falso`

```
a = 9

println(a % 2 == 0 ? "es divisible entre dos" : "no es divisible entre dos")
println(a % 3 == 0 ? "es divisible entre tres" : "no es divisible entre tres")
```

```
no es divisible entre dos
es divisible entre tres
```

1.1.5.1 Ciclos

Los ciclos son expresiones de control de flujo que nos permiten iterar sobre una colección o repetir un código hasta que se cumpla alguna condición. En Julia existen dos expresiones de ciclos:

- `for x in colección ...expresiones... end` y

- `while condición ...expresioens... end`

En el caso de `for`, la idea es iterar sobre una colección, esta colección puede ser un rango, i.e., `inicio:fin`, `inicio:paso:fin`, o una colección como las tuplas, los arreglos, o cualquiera que cumpla con la interfaz de colección iterable del lenguaje.

```
for i in 1:5
    println("1er ciclo: ", i => i^2)
end

for i in [10, 20, 30, 40, 50]
    println("2do ciclo: ", i => i/10)
end
```

```
1er ciclo: 1 => 1
1er ciclo: 2 => 4
1er ciclo: 3 => 9
1er ciclo: 4 => 16
1er ciclo: 5 => 25
2do ciclo: 10 => 1.0
2do ciclo: 20 => 2.0
2do ciclo: 30 => 3.0
2do ciclo: 40 => 4.0
2do ciclo: 50 => 5.0
```

Al igual que en otros lenguajes modernos, se define la variante completa o *comprehensive for* que se utiliza para transformar la colección de entrada en otra colección cuya sintaxis se ejemplifica a continuación:

```
a = [i => i^2 for i in 1:5]
println(a)
```

```
[1 => 1, 2 => 4, 3 => 9, 4 => 16, 5 => 25]
```

También es posible definir un generador, esto es, un código que puede generar los datos, pero que no los generará hasta que se les solicite.

```
a = (i => i^2 for i in 1:5)
println(a)
println(collect(a))
```

```
Base.Generator{UnitRange{Int64}, var"#3#4"}(var"#3#4"(), 1:5)
[1 => 1, 2 => 4, 3 => 9, 4 => 16, 5 => 25]
```

Otra forma de hacer ciclos de intrucciones es repetir mientras se cumpla una condición:

```
i = 0
while i < 5
    i += 1
    println(i)
end
```

```
i
```

```
1
2
3
4
5
```

```
5
```

1.1.6 Tuplas y arreglos en Julia

Una tupla es un conjunto ordenado de datos que no se puede modificar y que se desea esten contiguos en memoria, la sintaxis en memoria es como sigue:

```
a = (2, 3, 5, 7)                                    ①
b = (10, 20.0, 30f0)
c = 100 => 200
println(typeof(a))                                  ②
println(typeof(b))
println(typeof(c))
a[1], a[end], b[3], c.first, c.second               ③
```

① Define las tuplas.
② Imprime los tipos de las tuplas.
③ Muestra como se accede a los elementos de las tuplas. Julia indexa comenzando desde 1, y el término `end` también se utiliza para indicar el último elemento en una colección ordenada.

```
NTuple{4, Int64}
Tuple{Int64, Float64, Float32}
Pair{Int64, Int64}
```

```
(2, 7, 30.0f0, 100, 200)
```

La misma sintaxis puede generar diferentes tipos de tuplas. En el caso `NTuple{4, Int4}` nos indica que el tipo maneja cuatro elementos de enteros de 64 bits, los argumentos entre `{}` son parametros que especifican los tipos en cuestión. En el caso de `Tuple` se pueden tener diferentes tipos de elementos. La tupla `Pair` es especial ya que solo puede contener dos elementos y es básicamente para *embellecer* o *simplificar* las expresiones; incluso se crea con la sintaxis `key => value` y sus elementos pueden accederse mediante dos campos nombrados.

Los *arreglos* son datos del mismo tipo contiguos en memoria, a diferencia de las tuplas, los elementos se pueden modificar, incluso pueden crecer o reducirse. Esto puede implicar que se alojan en zonas de memoria diferente (las tuplas se colocan en el *stack* y los arreglos en el *heap*, ver la siguiente unidad para más información). Desde un alto nivel, los arreglos en Julia suelen estar asociados con vectores, matrices y tensores, y un arsenal de funciones relacionadas se encuentran definidas en el paquete `LinearAlgebra`, lo cual esta más allá del alcance de este curso.

```
a = [2, 3, 5, 7]                                    ①
b = [10, 20.0, 30f0]
println(typeof(a))                                  ②
println(typeof(b))
a[1], a[end], b[3], b[2:3]                          ③
```

① Define los arreglos `a` y `b`.

② Muestra los tipos de los arreglos, note como los tipos se promueven al tipo más génerico que contiene la definición de los datos.
③ El acceso es muy similar a las tuplas para arreglos unidimensionales, note que es posible acceder rangos de elementos con la sintaxis `ini:fin`.

```
Vector{Int64}
Vector{Float64}
```

```
(2, 7, 30.0, [20.0, 30.0])
```

```
a = [2 3;
     5 7]                                    ①
display(a)                                   ②
display(a[:, 1])                             ③
display(a[1, :])                             ④
```

① Definición de un arreglo bidimensional, note como se ignora la coma `,` en favor de la escritura por filas separadas por `;`.
② La variable `a` es una matriz de 2x2.
③ Es posible acceder una columna completa usando el símbolo `:` para indicar todos los elementos.
④ De igual forma, es posible acceder una fila completa.

```
2×2 Matrix{Int64}:
 2  3
 5  7
```

```
2-element Vector{Int64}:
 2
 5
```

```
2-element Vector{Int64}:
 2
 3
```

1.1.7 Diccionarios y conjuntos en Julia

Un diccionario es un arreglo asociativo, i.e., guarda pares llave-valor. Permite acceder de manera eficiciente al valor por medio de la llave, así como también verificar si hay una entrada dentro del diccionario con una llave dada. La sintaxis es como sigue:

```
a = Dict(:a => 1, :b => 2, :c => 3)          ①
a[:b] = 20                                   ②
println(a)
a[:d] = 4                                    ③
println(a)
delete!(a, :a)                               ④
a
```

① Definición del diccionario `a` que mapea simbolos a enteros.
② Cambia el valor de `:b` por 20.
③ Añade `:d => 4` al diccionario `a`.
④ Borra el par con llave `:a`.

```
Dict(:a => 1, :b => 20, :c => 3)
Dict(:a => 1, :b => 20, :d => 4, :c => 3)
```

```
Dict{Symbol, Int64} with 3 entries:
  :b => 20
  :d => 4
  :c => 3
```

Es posible utilizar diferentes tipos siempre y cuando el tipo en cuestión defina de manera correcta la función `hash` sobre la llave y la verificación de igualdad `==`.

Un conjunto se representa con el tipo `Set`, se implementa de manera muy similar al diccionario pero solo necesita el elemento (e.g., la llave). Como conjunto implementa las operaciones clasificación de operaciones de conjuntos

```
a = Set([10, 20, 30, 40])                    ①
println(20 in a)                             ②
push!(a, 50)                                 ③
println(a)
```

```
delete!(a, 10)                                    ④
println(a)
println(intersect(a, [20, 35]))                   ⑤
union!(a, [100, 200])                             ⑥
println(a)
```

① Definición del conjunto de números enteros.
② Verificación de membresia al conjunto `a`.
③ Añade 50 al conjunto.
④ Se borra el elemento 10 del conjunto.
⑤ Intersección de `a` con una colección, no se modifica el conjunto `a`.
⑥ Unión con otra colección, se modifica `a`.

```
true
Set([50, 20, 10, 30, 40])
Set([50, 20, 30, 40])
Set([20])
Set([50, 200, 20, 30, 40, 100])
```

1.2 El flujo de compilación de Julia

Basta con escribir una linea de código en el REPL de Julia y esta se compilará y ejecutará en el contexto actual, usando el ámbito de variables. Esto es conveniente para comenzar a trabajar, sin embargo, es importante conocer el flujo de compilación para tenerlo en cuenta mientras se códifica, y así generar código eficiente. En particular, la creación de funciones y evitar la *inestabilidad* de los tipos de las variables es un paso hacia la generación de código eficiente. También es importante evitar el alojamiento de memoria dinámica siempre que sea posible. A continuación se mostrará el análisis de un código simple a diferentes niveles, mostrando que el lenguaje nos permite observar la generación de código, que últimadamente nos da cierto control y nos permite verificar que lo que se esta implementando es lo que se específica en el código. Esto no es posible en lenguajes como Python.

```
1 let
2     e = 1.1
```

```
3       println(e*e)
4       @code_typed e*e
5   end
```

```
1.2100000000000002
```

```
CodeInfo(
1   %1 = Base.mul_float(x, y)::Float64
        return %1
) => Float64
```

En este código, se utiliza la estructa de agrupación de expresiones `let...end`. Cada expresión puede estar compuesta de otras expresiones, y casi todo es una expresión en Julia. La mayoria de las expresiones serán finalizadas por un salto de linea, pero las compuestas como `let`, `begin`, `function`, `if`, `while`, `for`, `do`, `module` estarán finalizadas con `end`. La indentación no importa la indentación como en Python, pero es aconsejable para mantener la legibilidad del código. La linea 2 define e inicializa la variable `e`; la linea 3 llama a la función `println`, que imprimirá el resultado de `e*e` en la consola. La función `println` esta dentro de la biblioteca estándar de Julia y siempre esta *visible*. La linea 4 es un tanto diferente, es una macro que toma la expresión `e*e` y realiza algo sobre la expresión misma, en particular `@code_type` muestra como se reescribe la expresión para ser ejecutada. Note como se hará una llamada a la función `Base.mul_float` que recibe dos argumentos y que regresará un valor `Float64`. Esta información es necesaria para que Julia pueda generar un código veloz, el flujo de compilación llevaría esta información a generar un código intermedio de *Low Level Virtual Machine* (LLVM), que es el compilador empotrado en Julia, el cual estaría generando el siguiente código LLVM (usando la macro `@code_llvm`):

```
; Function Signature: *(Float64, Float64)
;  @ float.jl:493 within `*`
define double @"julia_*_10689"(double %"x::Float64", double %"y::Float64") #0 {
top:
  %0 = fmul double %"x::Float64", %"y::Float64"
  ret double %0
}
```

Este código ya no es específico para Julia, sino para la maquinaría LLVM. Observe la especificidad de los tipos y lo corto del código. El flujo de compilación requeriría generar el código nativo, que puede ser observado a continuación mediante la macro `@code_native`:

```
    .text
    .file   "*"
    .globl  "julia_*_10882"                 # -- Begin function julia_*_10882
    .p2align    4, 0x90
    .type   "julia_*_10882",@function
"julia_*_10882":                        # @"julia_*_10882"
; Function Signature: *(Float64, Float64)
;  @ float.jl:493 within `*`
# %bb.0:                                # %top
;  @ float.jl within `*`
    #DEBUG_VALUE: *:x <- $xmm0
    #DEBUG_VALUE: *:y <- $xmm1
    push    rbp
    mov rbp, rsp
;  @ float.jl:493 within `*`
    vmulsd  xmm0, xmm0, xmm1
    pop rbp
    ret
.Lfunc_end0:
    .size   "julia_*_10882", .Lfunc_end0-"julia_*_10882"
;
                                        # -- End function
    .type   ".L+Core.Float64#10884",@object # @"+Core.Float64#10884"
    .section    .rodata,"a",@progbits
    .p2align    3, 0x0
".L+Core.Float64#10884":
    .quad   ".L+Core.Float64#10884.jit"
    .size   ".L+Core.Float64#10884", 8

.set ".L+Core.Float64#10884.jit", 140348701361632
    .size   ".L+Core.Float64#10884.jit", 8
    .section    ".note.GNU-stack","",@progbits
```

En este caso podemos observar código específico para la computadora que esta generando este documento, es posible ver el

manejo de registros y el uso de instrucciones del CPU en cuestión.

Este código puede ser eficiente dado que los tipos y las operaciones son conocidos, en el caso que esto no puede ser, la eficiencia esta perdida. Datos no nativos o la imposibilidad de determinar un tipo causarían que se generará más código nativo que terminaría necesitanto más recursos del procesador. Una situación similar ocurre cuando se aloja memoria de manera dinámica. Siempre estaremos buscando que nuestro código pueda determinar el tipo de datos para que el código generado sea simple, si es posible usar datos nativos, además de no manejar o reducir el uso de memoría dinámica.

1.3 Ejemplos de funciones

Las funciones serán una parte central de nuestros ejemplos, por lo que vale la pena retomarlas y dar ejemplos.

1.4 Recursos para aprender Python y Julia

1.4.1 Python

- Python, se recomieda utilizar la distribución de https://www.anaconda.com/download/
- Documentación oficial, comenzar por el tutorial https://docs.python.org/3/
- Documentación oficial https://docs.julialang.org/en/stable/

1.4.2 Julia

- Información sobre como instalar Julia y flujos de trabajo simples (e.g., REPL, editores, etc.) para trabajar con este lenguaje de programación: *Modern Julia Workflows* https://modernjuliaworkflows.github.io/.

- Libro sobre julia *Think Julia: How to Think Like a Computer Scientist* https://benlauwens.github.io/ThinkJulia.jl/latest/book.html.
- Curso *Introduction to computational thinking* https://computationalthinking.mit.edu/Fall20/

1.5 Licencia

2 Introducción al análisis de algoritmos con Julia

Este capítulo introduce los fundamentos de análisis de algoritmos. Se introduce el concepto de modelo de cómputo, se itroduce y se motiva la notación asintótica, ya que es el lenguaje común en el análisis de algoritmos. También se mostrarán algunos de los ordenes de crecimiento más representativos, que nos permitirán comparar algoritmos que resuelvan una tarea dada, así como permitirnos catalogarlos con respecto a los recursos de computo necesarios para ejecutarlos.

2.1 Concepto de algoritmo y estructura de datos

Los algoritmos son especificaciones formales de los pasos u operaciones que deben aplicarse a un conjunto de entradas para resolver un problema, obteniendo una solución correcta a dicho problema. Establecen los fundamentos de la programación y de la manera en como se diseñan los programas de computadoras. Dependiendo del problema, pueden existir múltiples algoritmos que lo resuelvan, cada uno de ellos con sus diferentes particularidades. Así mismo, un problema suele estar conformado por una cantidad enorme de instancias de dicho problema, por ejemplo, para una lista de n números, existen $n!$ formas de acomodarlos, de tal forma que puedan ser la entrada a un algoritmo cuya entrada sea una lista de números donde el orden es importante. En ocasiones, los problemas pueden tener infinitas de instancias. En este curso nos enfocaremos en problemas que pueden ser simplificados a una cantidad finita instancias.

Cada paso u operación en un algoritmo esta bien definido y puede ser aplicado o ejecutado para producir un resultado. A su vez,

cada operación suele tener un costo, dependiente del módelo de computación. Conocer el número de operaciones necesarias para transformar la entrada en la salida esperada, i.e., resolver el problema, es de vital importancia para seleccionar el mejor algoritmo para dicho problema, o aun más, para instancias de dicho problema que cumplen con ciertas características.

Una estructura de datos es una abstracción en memoria de entidades matemáticas y lógicas que nos permite organizar, almacenar y procesar datos en una computadora. El objetivo es que la información representada puede ser manipulada de manera eficiente en un contexto específico, además de simplificar la aplicación de operaciones para la aplicación de algoritmos.

2.2 Modelos de cómputo

Un modelo de cómputo es una abstracción matemática de una computadora o marco de trabajo algorítmico que nos permite estudiar y medir los costos de los algoritmos funcionando en este modelo de tal forma que sea más simple que una computadora física real. Ejemplos de estos modelos son las máquinas de Turing, las funciones recursivas, el cálculo lambda, o la máquina de acceso aleatorio. Todas estos modelos son *equivalentes* en sus capacidades, pero sus diferentes planteamientos permiten enfocarse en diferentes aspectos de los problemas.

- La máquina de Turing. Es un módelo creado por Alan Turing a principios del siglo XX; la idea es un dispositivo que podría ser implementada de manera mecánica si se tuvieran recursos infinitos; esta máquina puede leer y escribir en una cinta *infinita* una cantidad de símbolos predeterminada para cada problema siguiendo una serie de reglas simples sobre lo que lee y escribe, dichas reglas y la cienta, forman una máquina de estados y memoria, que pueden realizar cualquier cálculo si el tiempo no fuera un problema.
- Funciones recursivas. Se basa en funciones que trabajan sobre los números naturales y que definen en conjunto el espacio de funciones computables. Son una herramienta

abstracta que permite a los teóricos de la lógica y computación establecer los límites de lo computable.

- Cálculo lambda. Es un módelo creado por Alonzo Church y Stephen Kleene a principios del siglo XX, al igual que las funciones recursivas, se fundamenta en el uso de funciones y es una herramienta abstracta con própositos similares, sin embargo el cálculo lambda no se limita a recursiones, y se enfoca en diferentes reglas de reducción y composición de funciones, y es natural la inclusión de operadores de alto nivel, aunque estos mismos sean definidos mediante un esquema funcional.
- Máquina de acceso aleatorio (RAM). Es un módelo que describe una computadora con registros. Adiferencia de una computadora física, no tienen limitación en su capacidad, ni en la cantidad de registros ni en la precisión de los mismos. Cada registro puede ser identicado de manera única y su contenido leído y escrito mediante reglas o instrucciones formando un programa. En particular reconoce las diferencias entre registros de los programas y registros de datos, i.e., arquitectura harvard. Existe un número mínimo de instrucciones necesarias (i.e., incremento, decremento, poner a cero, copiar, salto condicional, parar) pero es común construir esquemas más complejos basados en estas primitivas. Se necesita un registro especial que indica el registro de programa siendo ejecutado. Los accesos a los registros tienen un tiempo constante a diferencia de otros esquemas; es el modelo más cercano a una implementación moderna de computadora.

Una computadora moderna difiere de muchas formas de una máquina RAM. De entrada, las limitaciones físicas requieren memorias finitas y registros con valores mínimos y máximos. También se debe trabajar con una jerarquía de memoria con diferentes niveles, donde los niveles más rápidos también son los más escasos; por tanto, es importante sacar provecho de esta jerarquía siempre que sea posible. Las operaciones también tienen costos diferentes, dependiendo de su implementación a nivel de circuitería, así como también existe cierto nivel de paralelización que no esta presente en una máquina RAM, tanto a nivel de procesamiento de datos como lectura de datos y el programa, esto sin tener en cuenta la arquitecturas multitarea

que ya es común en el equipo actual.

En este curso nos enfocaremos en especificaciones de alto nivel, donde los algoritmos pueden ser implementados en una computadora física, y estaremos contando operaciones de interés pensando en costos constantes en el acceso a memoria y en una selección de operaciones, al estilo de una máquina RAM.

La selección de operaciones de interés tiene el espíritu de simplificar el análisis, focalizando nuestros esfuerzos en operaciones que acumulan mayor costo y que capturan la dinámica del resto. Adicionalmente al conteo de operaciones nos interesa el desempeño de los algoritmos en tiempo real y en la cantidad de memoria consumida, por lo que se aboradará el costo realizando mediciones experimentales, contrastando con el análisis basado en conteo de operaciones siempre que sea posible.

2.3 Tipos de análisis

La pregunta inicial sería ¿qué nos interesa saber de un algoritmo que resuelve un problema? probablemente, lo primero sería saber si produce resultados correctos. Despues, entre el conjunto de las alternativas que producen resultados correctos, es determinante obtener su desempeño para conocer cuál es más conveniente para resolver un problema.

En ese punto, es necesario reconocer que para un problema, existen diferentes instancias posibles, esto es el espacio de instancias del problema, y que cada una de ellas exigirían soluciones con diferentes costos para cada algoritmo. Por tanto existen diferentes tipos de análisis y algoritmos.

- *Análisis de mejor caso.* Obtener el mínimo de resolver cualquier instancia posible, puede parecer poco útil desde el punto de vista de decisión para la selección de un algoritmo, pero puede ser muy útil para conocer un problema o un algoritmo.
- *Análisis de peor caso.* Obtener el costo máximo necesario para resolver cualquier instancia posible del problema con un algoritmo, este es un costo que si nos puede apoyar en la decisión de selección de un algoritmo; sin embargo, en

muchas ocasiones, puede ser poco informativo o innecesario ya que tal vez hay pocas instancias que realmente lo ameriten.
- *Análisis promedio.* Se enfoca en obtener un análisis promedio basado en la población de instancias del problema para un algoritmo dado.
- *Análisis amortizado.* Se enfoca en análisis promedio pero para una secuencia de instancias.
- *Análisis adaptativo.* Para un subconjunto *bien caracterizado* del espacio de instancias de un problema busca análizar los costos del algoritmo en cuestión. La caracterización suele estar en términos de una medida de complejidad para el problema; y la idea general es medir si un algoritmo es capaz de sacar provecho de instancias *fáciles*.

2.4 Notación asintótica

Realizar un conteo de operaciones y mediciones es un asunto complejo que requiere focalizar los esfuerzos. Para este fin, es posible contabilizar solo algunas operaciones de importancia, que se supondrían serían las más costosas o que de alguna manera capturan de manera más fiel la dinámica de costos.

El comportamiento asintótico es otra forma de simplificar y enfocarnos en los puntos de importancia.

2.4.1 Apoyo audio-visual

En los siguientes videos se profundiza sobre los modelos de cómputo y los diferentes tipos de análisis sobre algoritmos.

- Parte 1:
- Parte 2:
- Parte 3:

2.4.2 Ordenes de crecimiento

Dado que la idea es realizar un análisis asintótico, las constantes suelen ignorarse, ya que cuando el tamaño de la entrada

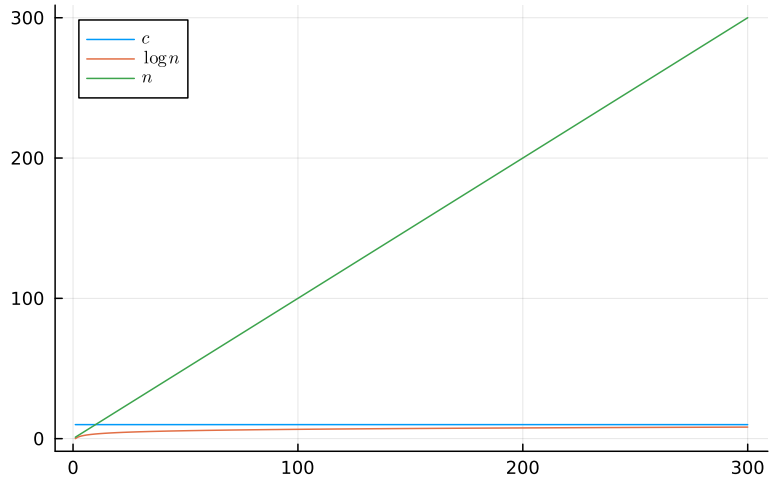
es suficientemente grande, los términos con mayor orden de magnitud o crecimiento dominarán el costo. Esto es, es una simplificación necesaría.

Los ordenes de crecimiento son maneras de categorizar la velocidad de crecimiento de una función, y para nuestro caso, de una función de costo. Junto con la notación asimptótica nos permite concentrarnos en razgos gruesos que se mantienen para entradas grandes, más que en los detalles, y no perder el punto de interés. A continuación veremos algunas funciones con crecimientos paradigmáticos; las observaremos de poco en poco para luego verlos en conjunto.

2.4.2.1 Costo constante, logaritmo y lineal

La siguiente figura muestra un crecimiento nulo (constante), logaritmico y lineal. Note como la función logarítmica crece lentamente.

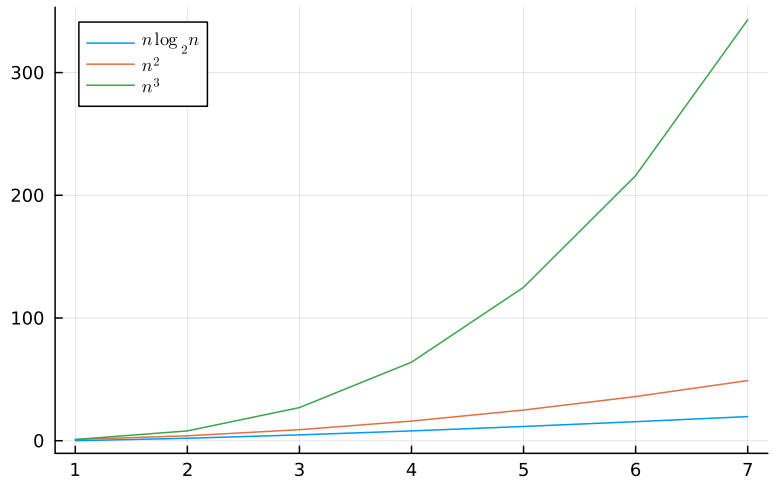
```
using Plots, LaTeXStrings
n = 300 # 300 puntos

plot(1:n, [10 for x in 1:n], label=L"c")
plot!(1:n, [log2(x) for x in 1:n], label=L"\log{n}")
plot!(1:n, [x for x in 1:n], label=L"n")
```



2.4.2.2 Costo $n \log n$ y polinomial

A continuación veremos tres funciones, una función con $n \log n$ y una función cuadrática y una cúbica. Note como para valores pequeños de n las diferencias no son tan apreciables para como cuando comienza a crecer n; así mismo, observe los valores de n de las figuras previas y de la siguiente, este ajuste de rangos se hizo para que las diferencias sean apreciables.

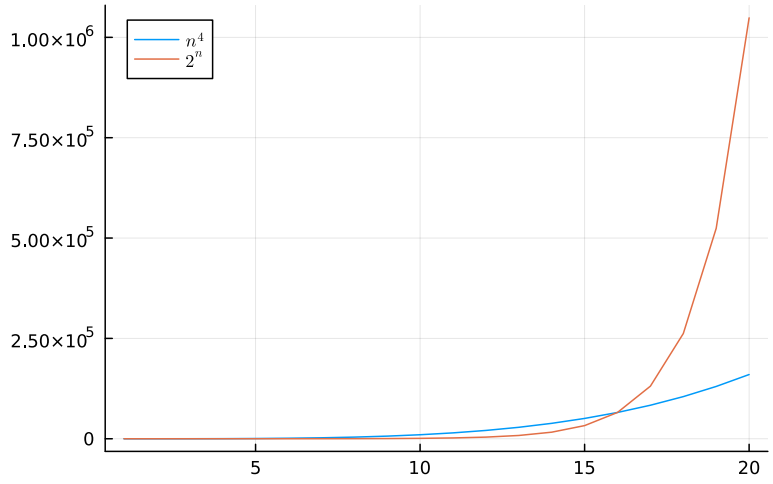
```
using Plots, LaTeXStrings
n = 7 # note que se usan menos puntos porque 300 serían demasiados para el rango

plot(1:n, [x * log2(x) for x in 1:n], label=L"n\log_2{n}")
plot!(1:n, [x^2 for x in 1:n], label=L"n^2")
plot!(1:n, [x^3 for x in 1:n], label=L"n^3")
```



2.4.2.3 Exponencial

A continuación se compara el crecimiento de una función exponencial con una función polinomial. Note que la función polinomial es de grado 4 y que la función exponencial tiene como base 2; aún cuando para números menores de aproximadamente 16 la función polinomial es mayor, a partir de ese valor la función 2^n supera rapidamente a la polinomial.

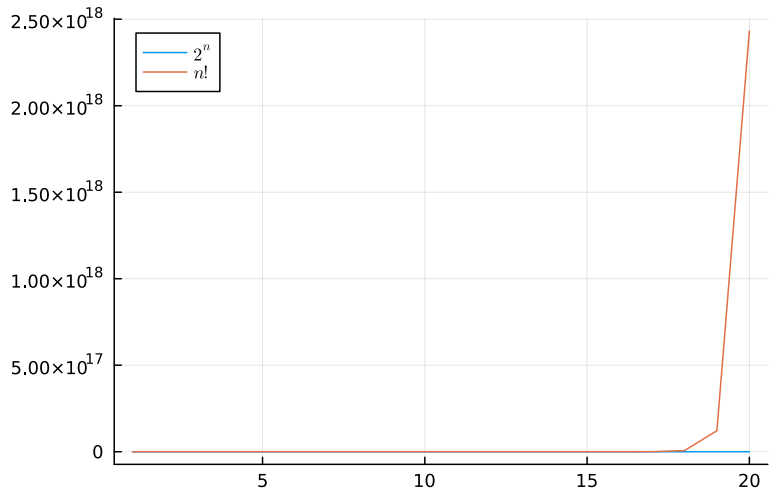
```
using Plots, LaTeXStrings
n = 20

plot(1:n, [x^4 for x in 1:n], label=L"n^4")
plot!(1:n, [2^x for x in 1:n], label=L"2^n")
```



2.4.2.4 Crecimiento factorial

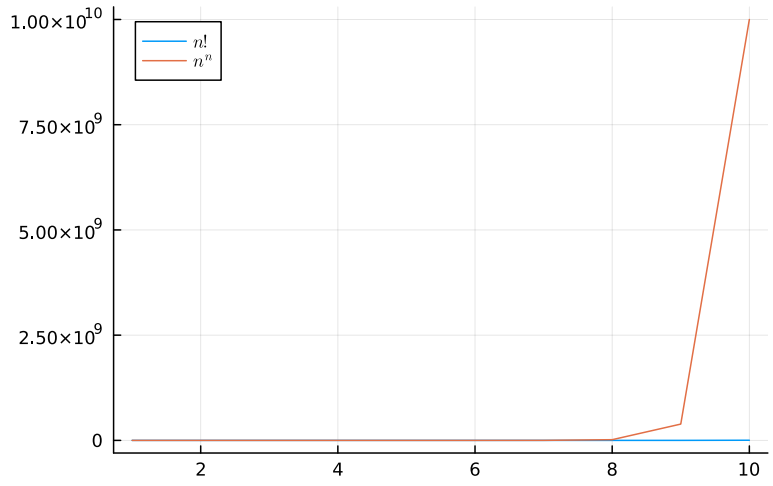
Vease como la función factorial crece mucho más rápido que la función exponencial para una n relativamente pequeña. Vea las magnitudes que se alcanzan en el *eje* y, y comparelas con aquellas con los anteriores crecimientos.

```
using Plots, LaTeXStrings
n = 20

plot(1:n, [2^x for x in 1:n], label=L"2^n")
plot!(1:n, [factorial(x) for x in 1:n], label=L"n!")
```



2.4.2.5 Un poco más sobre funciones de muy alto costo

```
n = 10

plot(1:n, [factorial(x) for x in 1:n], label=L"n!")
plot!(1:n, [x^x for x in Int128(1):Int128(n)], label=L"n^n")
```



Vea la figura anterior, donde se compara $n!$ con n^n, observe

como es que cualquier constante se vuelve irrelevante rapidamente; aun para n^n piense en n^{n^n}.

Note que hay problemas que son realmente costosos de resolver y que es necesario conocer si se comporta así siempre, si es bajo determinado tipo de entradas. Hay problemas en las diferentes áreas de la ciencia de datos, donde veremos este tipo de costos, y habrá que saber cuando es posible solucionarlos, o cuando se deben obtener aproximaciones que nos acerquen a las respuestas correctas con un costo manejable, es decir, mediar entre exactitud y costo. En este curso se abordaran problemas con un costo menor, pero que por la cantidad de datos, i.e., n, se vuelven muy costosos y veremos como aprovechar supuestos como las distribuciones naturales de los datos para mejorar los costos.

2.5 Conclusiones

Es importante conocer los ordenes de crecimiento más comunes de tal forma que podamos realizar comparaciones rápidas de costos, y dimensionar las diferencias de recursos entre diferentes tipos de costos. La notación asintótica hace uso extensivo de la diferencia entre diferentes ordenes de crecimiento para ignorar detalles y simplificar el análisis de algoritmos.

2.6 Actividades

Comparar mediante simulación en un notebook de Jupyter o Quarto los siguientes órdenes de crecimiento:

- $O(1)$ vs $O(\log n)$
- $O(n)$ vs $O(n \log n)$
- $O(n^2)$ vs $O(n^3)$
- $O(a^n)$ vs $O(n!)$
- $O(n!)$ vs $O(n^n)$

- Escoja los rangos adecuados para cada comparación, ya que como será evidente después, no es práctico fijar los rangos.
- Cree una figura por comparación, i.e., cinco figuras. Discuta lo observado por figura.
- Cree una tabla donde muestre tiempos de ejecución simulados para algoritmos ficticios que tengan los órdenes de crecimiento anteriores, suponiendo que cada operación tiene un costo de 1 nanosegundo.
  - Use diferentes tamaños de entrada $n = 100$, $n = 1000$, $n = 10000$ y $n = 100000$.
  - Note que para algunas fórmulas, los números pueden ser muy grandes, tome decisiones en estos casos y defiendalas en el reporte.
- Discuta las implicaciones de costos de cómputo necesarios para manipular grandes volúmenes de información, en el mismo notebook.

2.6.1 Entregable

Su trabajo se entregará en PDF y con el notebook fuente; deberá estar plenamente documentado, con una estructura que permita a un lector interesado entender el problema, sus experimentos y metodología, así como sus conclusiones. Tenga en cuenta que los notebooks pueden alternar celdas de texto y código.

No olvide estructurar su reporte, en particular el reporte debe cubrir los siguientes puntos: - Título del reporte, su nombre. - Introducción. - Código cercano a la presentación de resultados. - Figuras y comparación de los órdenes de crecimiento. - Análisis y simulación de costo en formato de tabla. - Conclusión. Debe abordar las comparaciones hechas y la simulación; también toque el tema de casos extremos y una n variable y asintóticamente muy grande. - Lista de referencias. Nota, una lista de referencias que no fueron utilizadas en el cuerpo del texto será interpretada como una lista vacía.

2.7 Bibliografía

Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2022). Introduction to Algorithms (2nd ed.). MIT Press.

- Parte I: Cap. 1, 2, 3

3 Estructuras de datos elementales

Objetivo

Implementar, aplicar y caracterizar el desempeño de algoritmos en peor caso y adaptativos para búsqueda en arreglos ordenados. Se discutirán estructuras de datos básicas que serán de gran utilidad al momento de construir programas y de resolver problemas más complejos; nos enfocaremos en las estructuras de datos .

3.1 Introducción

En esta unidad se discutirán las propiedades y operaciones básicas de estructuras como conjuntos, listas, pilas, colas, arreglos, vectores, matrices y matrices dispersas. La intención es utilizar código en el lenguaje de programación Julia, que pueda ser traducido fácilmente en otros lenguajes de programación; así como explicar las particularidades de las estructuras.

3.2 Conjuntos

Los *conjuntos* son estructuras abstractas que representan una colección de elementos, en particular, dado las posibles aplicaciones un conjunto puede tener contenido inmutable o mutable, esto es que puede aceptar modificaciones a dicha colección. Un conjunto puede estar vacio ($\emptyset$) o contener elementos, e.g., $\{a, b, c\}$. Un conjunto puede unirse con otro conjunto, e.g., $\{a, b\} \cup \{c\} = \{a, b, c\}$, así como puede intersectarse con otros

conjuntos, e.g. $\{a, b, c\} \cap \{b, d\} = \{b\}$. El tamaño de una colección lo representamos con barras, e.g., $|\{a, b\}| = 2$. También es útil consultar por membresia $a \in \{a, b, c\}$ o por la negación de membrsia, i.e., $a \notin \{a, b, c\}$. En contraste con la definición matemática de conjunto, es común necesitar conjuntos mutables en diferentes algoritmos, esto es, que permitan inserciones y borrados sobre la misma estructura. Esto es sumamente útil ya que nos permite hacer una representación en memoria que no requiera realizar copias y gestionar más memoria. Suponga el conjunto $S = \{a, b, c\}$, la función $pop!(S, b)$ resultaría en $\{a, c\}$, y la función $push!(S, d)$ resultaría en $\{a, c, d\}$ al encadenar estas operaciones. Note que el símbolo ! solo se esta usando en cooncordancia con el lenguaje de programación Julia para indicar que la función cambiaría el argumento de entrada, y es solo una convención, no un operador en sí mismo. Así mismo, note que estamos usando una sintaxis muy sencilla $fun(arg1, arg2, ...)$ para indicar la aplicación de una función u operación a una serie de argumentos.

Es importante hacer notar, que aunque es uno de los conceptos fundamentales, no existe una única manera de representar conjuntos, ya que los requerimientos de los algoritmos son diversos y tener la representación correcta puede ser la diferencia. Las implementaciones y algoritmos alrededor pueden llegar a ser muy sofisticados, dependiendo de las características que se desean, algunas de las cuales serán el centro de estudio de este curso.

3.3 Tuplas y estructuras

Las *tuplas* son colecciones abstractas ordenadas, donde incluso puede haber repetición, pueden verse como una secuencia de elementos, e.g., $S = (a, b, c)$; podemos referirnos a la $i$ésima posición de la forma S_i, o incluso $S[i]$, si el contexto lo amérita, e.g., pseudo-código que pueda ser transferido a un lenguaje de programación más fácilmente. Es común que cada parte de la tupla pueda contener cierto tipo de dato, e.g., enteros, números de punto flotante, símbolos, cadenas de carácteres, etc. Una tupla es muy amena para ser representada de manera contigua

en memoria. En el lenguaje de programación Julia, las tuplas se representan entre paréntesis, e.g., $(1, 2, 3)$.

```
t = (10, 20, 30)
```

```
t[1] * t[3] - t[2]
```

```
280
```

Definición y acceso a los campos de una tupla en Julia

Una *estructura* es una tupla con campos nombrados; es muy útilizada en lenguajes de programación, por ejemplo, en Julia la siguiente estructura puede representar un punto en un plano:

Dado que es amena para representarse de manera contigua en memoria, en los lenguajes de programación que aprovechen este hecho, una tupla puede enviarse como *valor* (copiar) cuando se utiliza en una función; por lo mismo, puede guardarse en el *stack*, que es la memoria *inmediata* que se tiene en el contexto de ejecución de una función. En esos casos, se puede optimizar el manejo de memoria (alojar y liberar), lo cuál puede ser muy beneficioso para un algoritmo en la práctico. El otro esquema posible es el *heap*, que es una zona de memoria que debe gestionarse (memoria dinámica); es más flexible y *duradera* entre diferentes llamadas de funciones en un programa. Los patrones esperados son dispersos y puede generar fragmentación

```
struct Point
  x::Float32
  y::Float32
end
```

Note la especificación de los tipos de datos que en conjunto describirán como dicha estructura se maneja por una computadora, y que en términos prácticos, es determinante para el desempeño. Es común asignar valores satelitales en programas o algoritmos, de tal forma que un elemento simple sea manipulado o utilizado de manera explicita en los algoritmos y tener asociados elementos secundarios que se vean afectados por las operaciones. Los conjuntos, tuplas y las estructuras son excelentes formas de representar datos complejos de una manera sencilla.

En Julia, es posible definir funciones o métodos al rededor del tipo de tuplas y estructuras.

Es importante saber que si algunos de los campos o datos de una tupla o estructura estan en el *heap* entonces solo una parte estará en el *stack*; i.e., en el caso extremo solo serán referencias a datos en el *heap*. Esto puede llegar a complicar el manejo de memoria, pero también puede ser un comportamiento sobre el que se puede razonar y construir.

```
"""
  Calcula la norma de un vector representado
  como un tupla
"""
function norm(u::Tuple)
  s = 0f0
  for i in eachindex(u)
```

```
    s = u[i]^2
  end
  sqrt(s)
end

"""
  Calcula la norma de un vector de 2 dimensiones
  representado como una estructura
"""
function norm(u::Point)
  sqrt(u.x^2 + u.y^2)
end

(norm((1, 1, 1, 1)), norm(Point(1, 1)))
```

```
(1.0, 1.4142135f0)
```

Funciones sobre diferentes tipos de datos

Note que la función es diferente para cada tipo de entrada; a este comportamiento se le llamada despacho múltiple y será un concepto común este curso. En otros lenguajes de programación se implementa mediante orientación a objetos.

3.4 Arreglos

Los *arreglos* son estructuras de datos que mantienen información de un solo tipo, tienen un costo constante $O(1)$ para acceder a cualquier elemento (también llamado acceso aleatorio) y tipicamente se implementan como memoria contigua en una computadora. Al igual que las tuplas, son colecciones ordenadas, las estaremos accediendo a sus elementos con la misma notación. En este curso usaremos arreglos como colecciones representadas en segmentos contiguos de memoria con dimensiones lógicas fijas. A diferencia de las tuplas, es posible reemplazar valores, entonces $S_{ij} \leftarrow a$, reemplazará el contenido de S en la celda especificada por a.

A diferencia de las tuplas, pueden tener más que una dimensión. La notación para acceder a los elementos se extiende, e.g. para

Julia tiene un soporte para arreglos excepcional, el cual apenas trataremos ya que se enfoca en diferentes áreas del cómputo numérico, y nuestro curso esta orientado a algoritmos. En Python, estructuras similares se encuentra en el paquete *Numeric Python* o *numpy*; tenga en cuenta que las afirmaciones sobre el manejo de memoria y representación que estaremos usando se apegan a estos modelos, y no a las *listas* nativas de Python.

una matriz S (arreglo bidimensional) S_{ij} se refiere a la celda en la fija i columna j, lo mismo que $S[i, j]$. Si pensamos en datos numéricos, un arreglo unidimensional es útil para modelar un *vector* de múltiples dimensiones, un arreglo bidimensional para representar una *mátriz* de tamaño $m \times n$, y arreglos de dimensión mayor pueden usarse para tensores. Se representan en memoria en segmentos contiguos, y los arreglos de múltiples dimensiones serán representados cuyas partes pueden ser delimitadas mediante aritmética simple, e.g., una matriz de tamaño $m \times n$ necesitará una zona de memoria de $m \times n$ elementos, y se puede acceder a la primera columna mediante en la zona $1, \ldots, m$, la segunda columna en $m + 1, \ldots, 2m$, y la $i$ésima en $(i - 1)m + 1, \ldots, im$; esto es, se implementa como el acceso en lotes de tamaño fijo en un gran arreglo unidimensional que es la memoria.

memoria RAM																
otros datos	columna 1 - x[:, 1]				columna 2 - x[:, 2]				columna 3 - x[:, 3]				columna 4 - x[:, 4]				otros datos
	x[1,1]	x[2,1]	x[3,1]	x[4,1]	x[1,2]	x[2,2]	x[3,2]	x[4,2]	x[1,3]	x[2,3]	x[3,3]	x[4,3]	x[1,4]	x[2,4]	x[3,4]	x[4,4]	

Figura 3.1: Esquema de una matriz en memoria.

Esta es la manera que en general se manejan los datos en una computadora, y conocerlo de manera explícita nos permite tomar decisiones de diseño e implementación.

La representación precisa en memoria es significativa en el desempeño de operaciones matriciales como pueden ser el producto entre matrices o la inversión de las mismas. La manera como se acceden los datos es crucial en el diseño de los algoritmos.

El siguiente ejemplo define un vector u de m elementos y una matriz X de tamaño $m \times n$, ambos en un cubo unitario de 4 dimensiones, y define una función que selecciona el producto punto máximo del vector u a los vectores columna de X:

```
function mydot(u, x)
  s = 0f0
  for i in eachindex(u, x)
    s += u[i] * x[i]
  end
  s
end

function getmaxdot(u::Vector, X::Matrix)
  maxpos = 1
```

```
  # en la siguiente linea, @view nos permite controlar que
  # no se copien los arreglos, y en su lugar, se usen referencias
  maxdot = mydot(u, @view X[:, 1])
  # obtiene el número de columnas e itera apartir del 2do indice
  mfilas, ncols = size(X)
  for i in 2:ncols
    d = mydot(u, @view X[:, i])
    if d > maxdot
      maxpos = i
      maxdot = d
    end
  end

  (maxpos, maxdot)
end

getmaxdot(rand(Float32, 4), rand(Float32, 4, 1000))
```

```
(353, 1.5240753f0)
```

En este código puede verse como se separa el cálculo del producto punto en una función, esto es porque en sí mismo es una operación importante; también podemos aislar de esta forma la manera que se accede (el orden) a los vectores. La idea fue acceder columna a columna, lo cuál asegura el uso apropiado de los accesos a memoria. En la función *getmaxdot* se resuelve el problema de encontrar el máximo de un arreglo, y se puede observar que sin conocimiento adicional, este requiere $O(n)$ comparaciones, para una mátriz de n columnas. Esto implica que cada producto punto se cuenta como $O(1)$, lo cual simplifica el razonamiento. Por la función *mydot* podemos observar que el producto punto tiene un costo de $O(m)$, por lo que la *getmaxdot* tiene un costo de $O(mn)$ operaciones lógicas y aritméticas.

El producto entre matrices es un caso paradigmático por su uso en la resolución de problemas prácticos, donde hay una gran cantidad de trabajo al rededor de los costos necesarios para llevarlo a cabo. En particular, el algoritmo naïve, es un algoritmo con costo cúbico, como se puede ver a continuación:

```
function myprod(A::Matrix, B::Matrix)
  mA, nA = size(A)
  mB, nB = size(B)
  @assert nA == mB
  C = Matrix{Float32}(undef, mA, nB)

  for i in 1:mA
    for j in 1:mB
      rowA = @view A[i, :]
      colB = @view B[:, i]
      C[i, j] = mydot(rowA, colB)
    end
  end

  C
end

A = rand(Float32, 5, 3)
B = rand(Float32, 3, 5)
C = myprod(A, B)
display(C)
```

```
5×5 Matrix{Float32}:
 0.984554  0.984554  0.984554   4.5869f-41   -4.36406f-29
 1.76572   1.76572   1.76572   -4.36406f-29   4.5869f-41
 0.397248  0.397248  0.397248   4.5869f-41   -4.36406f-29
 1.52667   1.52667   1.52667   -9.08707f-26   4.5869f-41
 0.975039  0.975039  0.975039   4.5869f-41   -4.36679f-29
```

Funciones sobre diferentes tipos de datos

Se pueden ver dos ciclos iterando a lo largo de filas y columnas, adicionalmente un producto punto, el cual tiene un costo lineal en la dimensión del vector, por lo que el costo es cúbico. Esta implementación es directa con la definición misma del producto matricial. Dado su implacto, existen diferentes algoritmos para hacer esta operación más eficiente, incluso hay áreas completas dedicadas a mejorar los costos para diferentes casos o características de las matrices.

3.5 Listas

Las *listas* son estructuras de datos ordenadas lineales, esto es, no se asume que los elementos se guardan de manera contigua y los accesos al i-ésimo elemento cuestan $O(i)$. Se soportan inserciones y borrados. Por ejemplo, sea $L = [a, b, c, d]$ una lista con cuatro elementos, $L_2 = b$, $insert!(L, 2, z)$ convertirá $L = [a, z, b, c, d]$ (note que b se desplazó y no se reemplazó como se esperaría en un arreglo). La operación $deleteat!(L, 2)$ regresará la lista a su valor previo a la inserción. Estas operaciones que modifican la lista también tienen diferentes costos dependiendo de la posición, e.g., donde el inicio y final de la secuencia (también llamados *cabeza* y *cola*) suelen ser más eficientes que accesos aleatorios, ya que se tienen referencias a estas posiciones en memoria. Es de especial importancia la navegación por la lista mediante operaciones de sucesor *succ* y predecedor *pred*, que pueden encadenarse para obtener acceso a los elementos. A diferencia de un arreglo, las listas no requieren una notación simple para acceso a los elementos y sus reemplazos, ya que su aplicación es diferente.

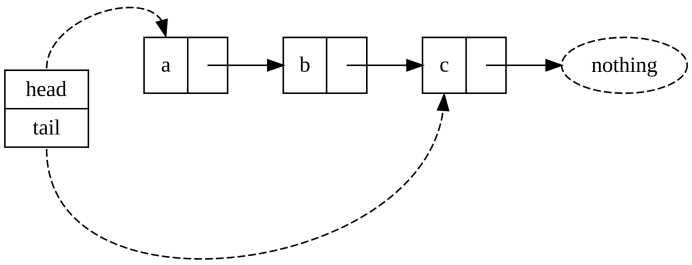


Figura 3.2: Una lista ligada simple

La Figura 3.2 muestra una lista ligada, que es una implementación de lista que puede crecer fácilmente, funciona en el *heap* de memoria por lo que cada bloque requiere memoria dinámica. Cada bloque es una estructura; se pueden distinguir dos tipos, la lista que contiene referencias al primer nodo y al último nodo. Los *nodos de de datos* contienen los elementos de la colección y referencias al siguiente nodo, también llamado *sucesor*. El nodo

nothing es especial y significa que no hay más elementos.

El siguiente código muestra como la definición de lista ligada.

Listado 3.1 Código para una lista ligada simple

```
struct Nodo
  data::Int
  next::Union{Nodo,Nothing}
end

nodo = Nodo(10, Nodo(20, Nodo(30, nothing)))

println(nodo)
(nodo.data, nodo.next.data, nodo.next.next.data)
```

```
Nodo(10, Nodo(20, Nodo(30, nothing)))

(10, 20, 30)
```

En el Listado 3.1 se ignora la referencia a *tail* (*head* se guarda en *nodo*), por lo que las operaciones sobre *tail* requieren recorrer la lista completa, costando $O(n)$ en el peor caso para una lista de n elementos.

Por su manera en la cual son accedidos los datos, se tienen dos tipos de listas muy útiles: las *colas* y las *pilas*. Las *colas* son listas que se acceden solo por sus extremos, y emulan la política de *el primero en entrar es el primero en salir* (first in - first out, FIFO), y es por eso que se les llama colas haciendo referencia a una cola para realizar un trámite o recibir un servicio. Las *pilas* o *stack* son listas con la política *el último en entrar es el último en salir* (last in - first out, LIFO). Mientras que cualquier lista puede ser útil para implementarlas, algunas maneras serán mejores que otras dependiendo de los requerimientos de los problemas siendo resueltos; sin embargo, es importante recordar sus políticas de acceso para comprender los algoritmos que las utilicen.

En este curso, se tienen en cuenta las siguientes operaciones, nombrando diferente cada operación:

- *push!(L, a)*: insertar a al final de la lista L.
- *pop!(L)*: remueve el último elemento en L.
- *deleteat!(L, pos)*: remueve el elemento en la posición *pos*, se desplazan los elementos.
- *insert!(L, pos, valor)*: inserta *valor* en la posición *pos* desplazando los elementos anteriores.

3.5.0.1 Ejercicios

- Implemente *insert!* y *deleteat!*
- ¿Cuál sería la implementación de *succ* y *pred* en una lista ligada?
- ¿Cuales serían sus costos?
- Añadiendo más memoria, como podemos mejorar *pred*?

3.5.1 Grafos

Otras estructuras de datos elementales son los *grafos*. Un grafo $G = (V, E)$ es una tupla compuesta por un conjunto de vertices V y el conjunto de aristas E. Por ejemplo, el grafo con $A = (\{a, b, c, d\}, \{(a, b), (b, c), (c, d), (d, a)\})$

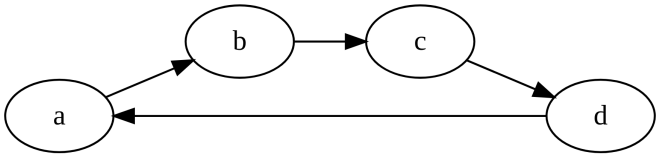


Figura 3.3: Un grafo dirigido simple

Los grafos son herramientas poderosas para representar de manera abstracta problemas que implican relaciones entre elementos. En algunos casos es útil asociar funciones a los vértices y las aristas. Tenga en cuenta los siguientes ejemplos:

- $peso : V \to \mathbb{R}$, la cual podría usarse como $peso(a) = 1.5$.

- $costo : V \times V \rightarrow \mathbb{R}$, la cual podría usarse como $costo(a, b) = 2.0$.

La estructura del grafo puede accederse mediante las funciones:

- $in(G, v) = \{u \mid (u, v) \in E\}$
- $out(G, u) = \{v \mid (u, v) \in E\}$

así como el número de vertices que entran y salen como:

- $indegree(G, v) = |in(G, v)|$.
- $outdegree(G, u) = |out(G, u)|$.

Un grafo puede tener aristas no dirigidas, el grafo con $B = (\{a, b, c, d\}, \{\{a, b\}, \{b, c\}, \{c, d\}, \{d, a\}\})$, no reconocerá orden en las aristas.

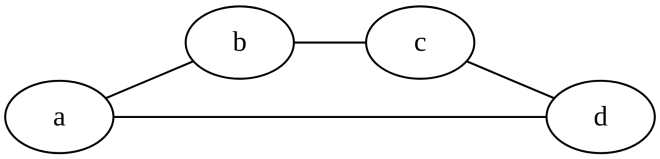


Figura 3.4: Un grafo cuyas aristas no estan dirigidas

Por lo tanto, podremos decir que $(a, b) \in E_A$ pero $(b, a) \notin E_A$. Por otro lado tenemos que $\{a, b\} \in E_B$, y forzando un poco la notación, $(a, b) \in E_B$, $(b, a) \in E_B$; para los conjuntos de aristas de A y B. La estructura puede ser accedida mediante $neighbors(G, u) = \{v \mid \{u, v\} \in E\}$.

Un grafo puede estar representado de diferentes maneras, por ejemplo, un arreglo bidimensional (matriz), donde $S_{ij} = 1$ si hay una arista entre los vértices i y j; y $S_{ij} = 0$ si no existe una arista. A esta representación se le llama matriz de adjacencia. Si el grafo tiene pocos 1's vale la pena tener una representación diferente; este es el caso de las listas de adjacencia, donde se representa cada fila o cada columna de la matriz de adjacencia como una lista de los elementos diferentes de cero.

Existen otras representaciones como la lista de coordenadas, *coordinate lists* (COO), o las representaciones dispersas compimidas, *sparse row* (CSR) y *compressed sparse column* (CSC) (Scott y Tůma 2023). Todas estas representaciones tratan de disminuir el uso de memoria y aprovechar la gran dispersión para realizar operaciones solo cuando sea estrictamente necesario.

Un *árbol* es un grafo en el cual no existen ciclos, esto es, no existe forma que en una caminata sobre los vértices, a traves de las aristas y prohibiendo revisitar aristas, es imposible regresar a un vértice antes visto.

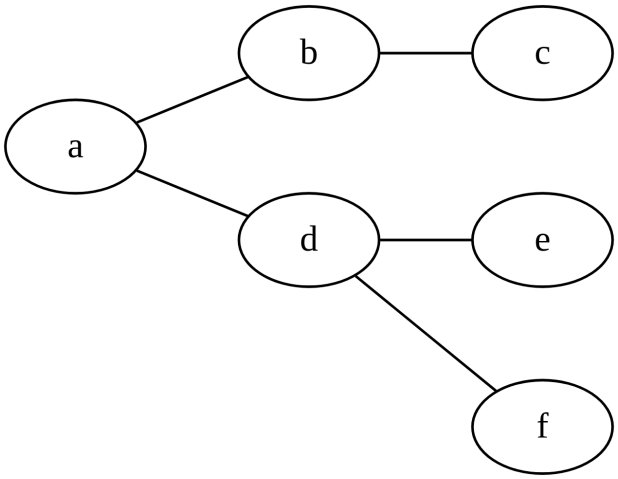


Figura 3.5: Árbol con aristas no dirigidas

En algunos casos, es conveniente identificar vértices especiales en un árbol $T = (V, E)$. Un vértice es la raíz del árbol, $root(T)$, es especial ya que seguramente se utilizará como acceso al árbol y por tanto contiene un camino a cada uno vértices en V. Cada vértice puede tener o no hijos, $children(T, u) = \{v \mid (u, v) \in E\}$. Se dice que u es un hoja (leaf) si $children(T, u) = \emptyset$, e interno (inner) si no es ni raíz ni hoja.

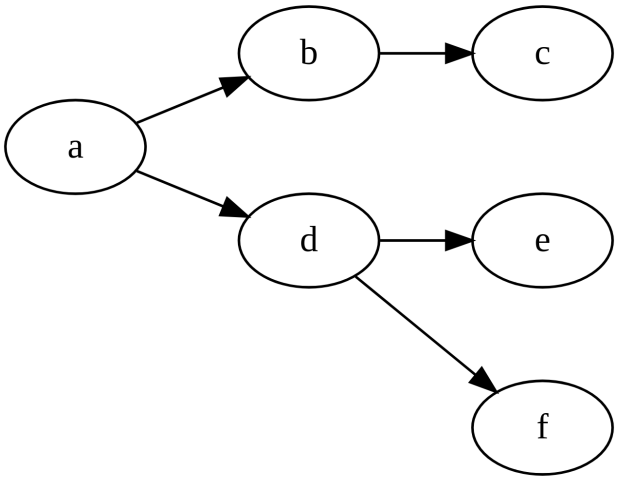


Figura 3.6: Árbol con aristas dirigidas, note que es fácil saber si hay un vértice o nodo que se distinga como raíz, o nodos que sean hojas.

Al igual que en los grafos más generales, en los árboles es útil definir funciones sobre vértices y aristas, así como marcar tipos de vértices, e.g., posición u color, que simplifiquen el razonamiento para con los algoritmos asociados.

Los nodos y las aristas de un grafo pueden *recorrerse* de diferentes maneras, donde se aprovechan las relaciones representadas. En un grafo general podría ser importante solo visitar una vez cada vértice, o guiarse en el recorrido por alguna heurística o función asociada a vértices o aristas.

El recorrido *primero a lo profundo*, Depth First Search (DFS), comienza en un nodo dado y de manera *voraz* avanzará recordando orden de visita y avanzando al ver un nuevo nodo repitiendo el procedimiento hasta que todos los vértices alcanzables sean visitados. El siguiente pseudo-código lo implementa:

```
#| lst-label: lst-dfs
#| lst-cap: Psudo-código DFS

function operación!(vértice)
  #... operaciones sobre el vértice siendo visitado ...
end

function DFS(grafo, vértice, visitados)
  operación!(vértice)
  push!(visitados, vértice)
  for v in neighbors(grafo, vértice)
    if v   visitados
      operación!(v)
      push!(visitados, v)
      DFS(grafo, v, visitados)
    end
  end
end

# ... código de preparación del grafo
visitados = Set()
DFS((vértices, aristas), vérticeinicial, visitados)
# ... código posterior a la visita DFS
```

Las llamadas recursivas a DFS tienen el efecto de *memorizar*

el orden de visita anterior y regresarlo cuando se sale de este, por lo que hay una memoria implicita utilizada, implementanda por el *stack* de llamadas. La función *operación!* es una abstracción de cualquier cosa que deba hacerse sobre los nodos siendo visitados.

El *recorrido a lo ancho*, Breadth First Search (BSF), visita los vértices locales primero que los alejados contrarío al avance voraz utilizado por DFS.

```
#| lst-label: lst-bfs
#| lst-cap: Psudo-código BFS

function BFS(grafo, vértice, visitados, cola)
  operación!(vértice)
  push!(visitados, vértice)
  push!(cola, vértice)

  while length(cola) > 0
    u = popfirst!(cola)
    for v in neighbors(grafo, u)
      if v   visitados
        operación!(v)
        push!(visitados, v)
        push!(cola, v)
      end
    end
  end
end

# ... código de preparación del grafo
visitados = Set()
BFS((vértices, aristas), vérticeinicial, visitados)
# ... código posterior a la visita BFS
```

El BFS hace uso explícito de la memoria para guardar el orden en que se visitarán los vértices (*cola*); se utiliza un conjunto para marcar vértices ya visitados (*visitados*) con la finalidad de evitar un recorrido infinito.

3.5.1.1 Ejercicios

- Implemente un grafo dirigido mediante listas de adyacencia.
- Implemente un grafo no dirigido mediante lista de adyacencia.
- Implemente el algoritmo de recorrido DFS y BFS con implementaciones de grafos.

3.6 Actividades

Implementar los siguientes algoritmos sobre matrices. - Multiplicación de matrices - Eliminación gaussiana / Gauss-Jordan Compare los desempeños de ambos algoritmos contando el número de operaciones y el tiempo real para matrices aleatorias de tamaño (n $\times$n) para (n= 100, 300, 1000). Maneje de manera separada los datos de conteo de operaciones (multiplicaciones y sumas escalares) y las de tiempo real. Discuta sus resultados experimentales; ¿qué puede concluir? ¿Cuál es el impacto de acceder los elementos contiguos en memoria de una matriz? ¿Qué cambiaría si utiliza matrices dispersas? ¿Cuáles serían los costos?

Entregable

Su trabajo se entregará en PDF y con el notebook fuente; deberá estar plenamente documentado, con una estructura que permita a un lector interesado entender el problema, sus experimentos y metodología, así como sus conclusiones. Tenga en cuenta que los notebooks pueden alternar celdas de texto y código.

No olvide estructurar su reporte, en particular el reporte debe cubrir los siguientes puntos:

- Título del reporte, su nombre.
- Introducción.
- Código cercano a la presentación de resultados.
- Figuras y tablas
- Análisis de los resultados
- Conclusión, discusiones de las preguntas

- Lista de referencias. Nota, una lista de referencias que no fueron utilizadas en el cuerpo del texto será interpretada como una lista vacía.

3.7 Bibliografía

Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2022). Introduction to Algorithms (2nd ed.). MIT Press.

- Parte III: Cap 10 Elementary Data Structures.
- Parte VI: Cap 22 Elementary Graph Algorithms.
- Parte VII: Cap 28 Matrix Operations.

4 Algoritmos de ordenamiento

Objetivo

Implementar, utilizar y caracterizar el desempeño de algoritmos peor caso y adaptables a la distribución para ordenamiento de arreglos.

4.1 Introducción

En este tema se aborda el ordenamiento basado en comparación, esto es, existe una función $\leq$. Recuerde que se cumplen las siguientes propiedades:

- si $u \leq v$ y $v \leq w$ entonces $u \leq w$ (transitividad)
- tricotomia:
  - si $u \leq v$ y $v \leq u$ entonces $u = v$ (antisimetria)
  - en otro caso, $u \leq v$ o $v \leq u$

La idea es entonces, dado un arreglo $A[1, n] = a_1, a_2, \cdots, a_n$ obtener una permutación π tal que $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. Si se asegura que en el arreglo ordenado se preserven el orden original de los indices cuando $u = v$, entonces se tiene un ordenamiento estable.

En terminos prácticos, la idea es reorganizar A, mediante el cálculo implicito de π, de tal forma que después de terminar el proceso de ordenamiento se obtenga que A esta ordenado, i.e., $a_i \leq a_{i+1}$. En sistemas reales, el alojar memoria para realizar el ordenamiento implica costos adicionales, y es por esto que es común modificar directamente A. Utilizar π solo es necesario cuando no es posible modificar A. También es muy común utilizar datos *satélite* asociados con los valores a comparar, de esta manera es posible ordenar diversos tipos de datos.

En esta unidad se tendrá atención especial a aquellos algoritmos oportunistas que son capaces de obtener ventaja en instancias sencillas.

4.1.1 Lecturas

Las lecturas de este tema corresponden al capítulo 5 de (Knuth 1998), en específico 5.2 *Internal sorting*. También se recomienda leer y comprender la parte II de (Cormen et al. 2022), que corresponde a *Sorting and order statistics*, en partícular Cap. 6 y 7, así como el Cap. 8.1. El artículo de wikipedia https://en.wikipedia.org/wiki/Sorting_algorithm también puede ser consultado con la idea de encontrar una explicación rápida de los algoritmos.

En la práctica, pocos algoritmos son mejores que *quicksort*. En (Loeser 1974) se detalla una serie de experimentos donde se compara quicksort contra otros algoritmos relacionados; por lo que es una lectura recomendable.

La parte adaptable, esto es para algoritmos *oportunistas* que toman ventaja de instancias simples, esta cubierta por el artículo (Estivill-Castro y Wood 1992). En especial, es muy necesario comprender las secciones 1.1 y 1.2, el resto del artículo debe ser leído aunque no invierta mucho tiempo en comprender las pruebas expuestas si no le son claras. En especial, en las secciones indicadas se establecen las medidas de desorden contra las cuales se mide la complejidad. En (Cook y Kim 1980) realiza una comparación del desempeño de varios algoritmos para ordenamiento de listas casi ordenadas, esto es, en cierto sentido donde los algoritmos adaptables tienen sentido. Este artículo es anterior a (Estivill-Castro y Wood 1992) pero tiene experimentos que simplifican el entendimiento de los temas.

4.2 Material audio-visual sobre algoritmos de ordenamiento

4.3 Actividades

4.3.1 Actividad 0 [sin entrega]

Realizar las actividades de lectura y comprensión. - De preferencia realice los ejercicios de los capítulos y secciones relacionadas.

4.3.2 Actividad 1 [con reporte]

1. Implemente los algoritmos, bubble-sort, insertion-sort, merge-sort y quick-sort. Explíquelos.
2. Carge los archivos `unsorted-list-p=*.json`, los cuales corresponden al archivo `listas-posteo-100.json` perturbado en cierta proporción: $p = 0.01, 0.03, 0.10, 0.30$.

- En el notebook `perturbar-listas.ipynb` se encuentran el procedimiento que se utilizó para la perturbación.
- Nota: puede usar sus propias listas de posteo perturbadas para la actividad siempre y cuando sean comparables en tamaño.
- Recuerde que la unidad anterior se dió un notebook con el código para crear índices invertidos y las listas de posteo.

3. Para cada archivo de listas desordenadas con cierta perturbación, realice el siguiente experimento:

- Ordene con los algoritmos implementados para cada valor de p con cada.
- Grafique el número de comparaciones necesarias para ordenar las 100 listas.
- Grafique el tiempo en segundos necesario para ordenar las 100 listas.

4. Muestre de manera agregada la información de todos los experimentos en una tabla.

- Nota 1: Recuerde copiar o cargar cada lista para evitar ordenar conjuntos completamente ordenados.
- Nota 2: Repita varias veces las operaciones de ordenamiento, esto es muy importante sobre para la estabilidad de los tiempos en segundos (vea Nota 1).
- Nota 3: En las implementaciones podrá usar cualquier comparación que le convenga, i.e., $<, \leq$, $cmp \to \{-1, 0, 1\}$, etc.
- Nota 4: Tome en cuenta que varios lenguajes de programación (Python y Julia) hacen copias de los arreglos cuando se usa *slicing*, i.e., `arr[i:j]` creará un nuevo arreglo y eso implica costos adicionales innecesarios:
  - Python: use índices o arreglos de `numpy`.
  - Julia: use índices o vistas, i.e., `@view`.

4.3.3 Entregable

El reporte deberá ser en formato notebook y el PDF del mismo notebook. El notebook debe contener las implementaciones de los algoritmos solicitados. Recuerde que el reporte debe llevar claramente su nombre, debe incluir una introducción, la explicación de los experimentos realizados, las observaciones, conclusiones y bibliografía.

Para generar el PDF primero guarde el notebook como HTML y luego genere el PDF renderizando e imprimiendo el HTML con su navegador. En lugar de imprimir, seleccione guardar como PDF.

5 Algoritmos de búsqueda en el modelo de comparación

Esta unidad esta dedicada a la implementación y análisis de algoritmos de búsqueda sobre arreglos ordenados, esto es que presentan un orden total. Un arreglo es una estructura lineal de elementos contiguos en memoría donde la posición es importante. En esta unidad se estudian algoritmos que para localizar elementos que cumplan con predicados simples de orden. Como restricción adicional, se limita la duplicidad de elementos en los arreglos, esto sin reducir la generalidad de los algoritmos estudiados. Para cualquier tripleta de elementos a, b, c en el a arreglo se cumple lo siguiente:

- reflexividad: $a \leq a$.
- transitividad: $a \leq b$ y $b \leq c$ entonces $a \leq c$.
- anti-simetría: $a \leq b$ y $b \leq a$ entonces $a = b$.
- completitud: $a \leq b$ o $b \leq a$.

Note que dada la condición de arreglo consecutivo en memoria, para dos elementos u_i y u_j, donde i y j son posiciones:

- $u_i < u_j \iff i < j$; note que el comparador es estricto.
- $u_i = u_j \iff i = j$.

Los algoritmos tomarán ventaja de este hecho para localizar de manera precisa y eficiente elementos deseados, descritos mediante los mismos operadores.

5.0.1 Listas ordenadas

En esta unidad se aborda la búsqueda en arreglos ordenados, y abusando del término, muchas veces les llamaremos *listas ordenadas*. Recuerde que a lo largo de este curso, esta será nuestra representación para conjuntos.

En la literatura es común que se aborde el tema con un modelo de costo basado en comparaciones, esto es, cada comparación $\leq$ provoca costo constante $O(1)$. Este curso no es la excepción. La comparación como unidad de costo es un excelente factorizador de las operaciones satelitales en los algoritmos de búsqueda; esto debería quedar claro una vez que se comprendan los algoritmos.

Utilizaremos como base el artículo (Jon Louis Bentley y Yao 1976), que es de lectura forzosa. Nos apoyaremos en una serie de lecturas adicionales para comprender y madurar el concepto.

5.1 Material audio-visual

En el siguiente video se adentraran en diferentes estrategías de búsqueda, notoriamente aquellas que llamaremos oportunistas o adaptables (adaptative). Estas técnicas nos permitirán tomar provecho de instancias sencillas de problemas e incrementar el desempeño en ese tipo de instancias.

Tenga en cuenta que, honrando la literatura, usaremos de forma indiscriminada listas ordenadas como sinónimo de arreglos ordenados.

5.1.1 Búsqueda

5.2 Actividades

5.2.1 Actividad 0 [sin entrega]

Realizar las actividades de lectura y comprensión, apoyosé en el video de esta unidad. De preferencia realice los ejercicios de las secciones relacionadas.

- El artículo sobre búsqueda no acotada, como representativo sobre búsqueda adaptativa (Jon Louis Bentley y Yao 1976).
- Cap. 12 de (Sedgewick 1998), en partícular Sec. 12.3 y 12.4.

- Cap. 6 de (Knuth 1998), en partícular Sec. 6.1 y 6.2.
- El artículo sobre búsqueda adaptativa secuencial (Jon L. Bentley y McGeoch 1985).
- Recuerde la referencia básica para la notación y conceptos es (Cormen et al. 2022).

5.2.2 Actividad 1 [con reporte]

Realice y reporte el siguiente experimento:

- Use el archivo `listas-posteo-100.json`, contiene las 100 listas de posteo más frecuentes, se encuentran en formato JSON.
- Utilice las listas (sin el término asociado).
- Los usuarios de Julia deberán asegurar que los tipos de los arreglos es `Int` y no `Any` para asegurar la velocidad adecuada
- Seleccione 1000 identificadores de documentos al azar, entre 1 y n, recuerde que $n = 50,000$.
- Grafique el tiempo promedio de *buscar* los 1000 identificadores en todas las listas (un solo número que represente las 100×1000 búsquedas). Nota: lo que determinará al buscar es la *posición de inserción* que se define como el lugar donde debería estar el identificador si se encontrara en la lista.
- Los algoritmos que caracterizará son los siguientes (nombres con referencia a (Jon Louis Bentley y Yao 1976)):
  - Búsqueda binaria acotada
  - Búsqueda secuencial B_0
  - Búsqueda no acotada B_1
  - Búsqueda no acotada B_2
  - *Importante: Tal vez deba repetir varias veces cada búsqueda si los tiempos son muy pequeños.*
- Bosqueje en pseudo-código la implementación de la búsqueda casí optima B_k

5.2.3 Entregable

El reporte deberá ser en formato notebook y el PDF del mismo notebook. El notebook debe contener las implementaciones de los algoritmos solicitados. Recuerde que el reporte debe llevar claramente su nombre, debe incluir una introducción, la explicación de los experimentos realizados, las observaciones, conclusiones y bibliografía.

Nota: En las implementaciones podrá usar comparación $<, \leq$, o incluso $cmp \rightarrow \{-1, 0, 1\}$, teniendo en cuenta que cmp es común en lenguajes modernos, solo debe indicarlo.

5.2.4 Actividad 2 [sin entrega]

Revisar el notebook `crear-indice-invertido.ipynb` para los detalles de como se generó la lista de posteo. Usted puede crear nuevas listas de posteo si lo desea usando los conjuntos de datos disponibles (listados en dicho notebook), y a su vez utilizarlas en las actividades de este Unidad. Solo deberá indicarlo; recuerde que los números de documentos y tamaño de vocabulario cambiarán.

5.2.5 Leyendo las listas de posteo

Usted no necesita generar las listas de posteo, solo leer las que se le han proporcionado en el archivo `listas-posteo-100.json` que corresponden a las 100 listas de posteo más pobladas (100 terminos más usados en el conjunto de datos). En el archivo `listas-posteo-100.json` , cada linea un JSON valido, donde se tiene el término y la lista de posteo.

- En el notebook `lectura-listas-de-posteo.ipynb` se muestra como se leen las listas de posteo desde Julia

6 Algoritmos de intersección de conjuntos con representación de listas ordenadas

Objetivo

Implementar y comparar algoritmos de intersección de conjuntos representados como listas ordenadas, utilizando una variedad de algoritmos de búsqueda que dan diferentes propiedades a los algoritmos de intersección.

6.1 Introducción

En este tema se conocerán, implementarán y compararán algoritmos de intersección de listas ordenadas. El cálculo de la intersección es un proceso costoso en una máquina de búsqueda, sin embargo, es un procedimiento esencial cuando se trabaja con grandes colecciones de datos.

El índice invertido tal y como lo hemos creado, es capaz de manejar una cantidad razonablemente grande de documentos. Para asegurarnos del escalamiento con la cantidad de documentos, es necesario utilizar algoritmos de intersección que sean eficientes. Entonces, dadas las listas ordenadas $L_1, \cdots, L_k$ (e.g, correspondientes a las listas de posteo en un índice invertido), tomará dichas listas y producirá $L^* = \bigcap_i L_i$, esto es, si $u \in L^*$ entonces $u \in L_i$ para $1 \leq i \leq k$.

Existen varios algoritmos prominentes para llevar a cabo esta operación. Uno de los trabajos seminales viene de Hwang &

Lin, en su algoritmo de *merge* entre dos conjuntos (Hwang y Lin 1971). En este trabajo se replantea el costo como encontrar los *puntos* de unión entre ambos conjuntos, esto se traslada de manera inmediata al problema de intersección. El problema correspondiente para intersectar dos conjuntos cualesquiera representados como conjuntos ordenados es entonces $\log \binom{n+m}{m}$, que usando la aproximación de Stirling se puede reescribir como

$$n \log \frac{n}{m} + (n-m) \log \frac{n}{n-m},$$

donde n y m corresponden a al número de elementos en cada conjunto.

Un algoritmo *naïve* para realizar la intersección, puede ser buscar todos los elementos del conjunto más pequeño en el más grande. Si para la búsqueda se utiliza *búsqueda binaria*, tenemos un costo de $m \log n$.

Esta simple idea puede ser explotada y mejorada para obtener costos más bajos, por ejemplo, si en lugar de buscar sobre la lista más grande directamente, esta se divide en bloques de tamaño m para encontrar el bloque que contiene cada elemento (recuerde que el arreglo esta ordenado), para después buscar dentro del bloque. Haciendo esto, el costo se convierte en

$$m \log \frac{n}{m} + m \log m$$

cuyo costo se ajusta mejor al costo del problema. Este es el algoritmo propuesto, a groso modo, en (Hwang y Lin 1971).

Cuando $k > 2$, la intersección se puede realizar usando las k listas a la vez, o se puede hace por pares. Se puede observar que la intersección de dos conjuntos da como resultado un conjunto igual o más pequeño que el más pequeño de los conjuntos intersectados. Adicionalmente, los conjuntos pequeños son "más faciles" de intersectar con un algoritmo *na"ive*. Por tanto, una estrategía que funciona bien en el peor caso es intersectar los 2 arreglos más pequeños cada vez. Esta una idea muy popular llamada *Small vs Small (SvS)*.

Existe otra familia de algoritmos, basados en búsquedas adaptativas que pueden llegar a mejorar el desempeño bajo cierto tipo de entradas. En (Demaine, López-Ortiz, y Ian Munro

2001), (Barbay, López-Ortiz, y Lu 2006), (Barbay et al. 2010), y (Baeza-Yates y Salinger 2005) se muestran comparaciones experimentales de diversos algoritmos de intersección, entre ellos adaptables, que utilizan de manera creativa algoritmos de búsqueda adaptables para aprovechar instancias simples. Estos estudios se basan en contribuciones teoricas de los mismos autores (Demaine, López-Ortiz, y Munro 2000), (Demaine, López-Ortiz, y Ian Munro 2001), (Barbay y Kenyon 2002), (Baeza-Yates 2004).

6.2 Recursos audio-visuales de la unidad

Parte 1: Algoritmos de intersección (y unión) de listas ordenadas

Parte 2: Algoritmos de intersección y algunas aplicaciones

6.3 Actividades

Implementación y comparación de diferentes algoritmos de intersección de conjuntos.

Lea cuidadosamente las instrucciones y desarrolle las actividades. Entregue el reporte correspondiente en tiempo.

6.3.1 Actividad 0 [Sin entrega]

1. Lea y comprenda los artículos relacionados (listados en la introducción).

6.3.2 Actividad 1 [Con reporte]

1. Cargue el archivo `listas-posteo-100.json` del tema 3. Si lo desea, puede usar listas de posteo generadas con otros conjuntos de datos, usando los scripts de las unidades pasadas. Si es necesario, repase los temas anteriores para recordar la naturaleza y propiedades de las listas.

- Sea $P^{(2)}$ el conjunto de todos los posibles pares de listas entre las 100 listas de posteo. Seleccione de manera aleatoria $A \subset P^{(2)}$, $|A| = 1000$.
- Sea $P^{(3)}$ el conjunto de todas las posibles combinaciones de tres listas de posteo entre las 100 listas disponibles, Seleccione de manera aleatoria $B \subset P^{(3)}$, $|B| = 1000$.
- Sea $P^{(4)}$ el conjunto de todas las posibles combinaciones de cuatro listas de posteo entre las 100 listas disponibles. Seleccione de manera aleatoria $C \subset P^{(4)}$, $|C| = 1000$.

2. Implemente los algoritmos de las secciones 3.1 *Melding Algorithms* y 3.2 *Search algorithms* (en especial 3.2.1 y 3.2.2) de (Barbay et al. 2010).

3. Realice y reporte los siguientes experimentos:

- Intersecte cada par de listas $a, b \in A$, y reporte de manera acumulada el tiempo en segundos y el número de comparaciones.
- Intersecte cada tripleta de listas $a, b, c \in B$, y reporte de manera acumulada el tiempo en segundos y el número de comparaciones.
- Intersecte cada tetrapleta de listas $a, b, c, d \in C$, y reporte de manera acumulada el tiempo en segundos y el número de comparaciones.
- Cree una figura `boxplot` que describa el tiempo en segundos para los tres experimentos.
- Cree una figura `boxplot` que describa el número de comparaciones para los tres experimentos.
- Cree una figura `boxplot` que describa las longitudes de las intersecciones resultantes para A, B, C.

6.3.3 Entregable

El reporte deberá ser en formato notebook y el PDF del mismo notebook. El notebook debe contener las implementaciones. Recuerde que el reporte debe llevar claramente su nombre, debe incluir una introducción, la explicación de los métodos usados, la explicación de los experimentos realizados, la discusión de los resultados, y finalizar con sus observaciones y conclusiones.

Nota sobre la generación del PDF: Jupyter no genera el PDF directamente, a menos que se tengan instalados una gran cantidad de paquetes, entre ellos una instalación completa de LaTeX. En su lugar, para generar el PDF en Jupyter primero guarde el notebook como HTML y luego genere el PDF renderizando e imprimiendo el HTML con su navegador. En lugar de imprimir, seleccione guardar como PDF.

References

Baeza-Yates, Ricardo. 2004. «A fast set intersection algorithm for sorted sequences». En *Combinatorial Pattern Matching: 15th Annual Symposium, CPM 2004, Istanbul, Turkey, July 5-7, 2004. Proceedings 15*, 400-408. Springer.

Baeza-Yates, Ricardo, y Alejandro Salinger. 2005. «Experimental analysis of a fast intersection algorithm for sorted sequences». En *International Symposium on String Processing and Information Retrieval*, 13-24. Springer.

Barbay, Jérémy, y Claire Kenyon. 2002. «Adaptive intersection and t-threshold problems». En *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 390-99. SODA '02. USA: Society for Industrial; Applied Mathematics.

Barbay, Jérémy, Alejandro López-Ortiz, y Tyler Lu. 2006. «Faster adaptive set intersections for text searching». En *Experimental Algorithms: 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006. Proceedings 5*, 146-57. Springer.

Barbay, Jérémy, Alejandro López-Ortiz, Tyler Lu, y Alejandro Salinger. 2010. «An experimental investigation of set intersection algorithms for text searching». *Journal of Experimental Algorithmics (JEA)* 14: 3-7.

Bentley, Jon L., y Catherine C. McGeoch. 1985. «Amortized analyses of self-organizing sequential search heuristics». *Commun. ACM* 28 (4): 404-11. https://doi.org/10.1145/3341.3349.

Bentley, Jon Louis, y Andrew Chi-Chih Yao. 1976. «An almost optimal algorithm for unbounded searching». *Information Processing Letters* 5 (3): 82-87. https://doi.org/https://doi.org/10.1016/0020-0190(76)90071-5.

Cook, Curtis R, y Do Jin Kim. 1980. «Best sorting algorithm for nearly sorted lists». *Communications of the ACM* 23 (11): 620-24.

Cormen, Thomas H, Charles E Leiserson, Ronald L Rivest, y Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

Demaine, Erik D, Alejandro López-Ortiz, y J Ian Munro. 2001. «Experiments on adaptive set intersections for text retrieval systems». En *Algorithm Engineering and Experimentation: Third International Workshop, ALENEX 2001 Washington, DC, USA, January 5–6, 2001 Revised Papers 3*, 91-104. Springer.

Demaine, Erik D, Alejandro López-Ortiz, y J Ian Munro. 2000. «Adaptive set intersections, unions, and differences». En *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, 743-52.

Estivill-Castro, Vladmir, y Derick Wood. 1992. «A survey of adaptive sorting algorithms». *ACM Computing Surveys (CSUR)* 24 (4): 441-76.

Hwang, Frank K., y Shen Lin. 1971. «Optimal merging of 2 elements with n elements». *Acta Informatica* 1 (2): 145-58.

Knuth, Donald. 1998. *The Art Of Computer Programming, vol. 3 (2nd ed): Sorting And Searching*. Vol. 3. Redwood City, CA, USA.: Addison Wesley Longman Publishing Co. Inc.

Loeser, Rudolf. 1974. «Some performance tests of "quicksort" and descendants». *Communications of the ACM* 17 (3): 143-52.

Scott, Jennifer, y Miroslav Tůma. 2023. «An Introduction to Sparse Matrices». En *Algorithms for Sparse Linear Systems*, 1-18. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-031-25820-6_1.

Sedgewick, Robert. 1998. *Algorithms in c++, parts 1-4: fundamentals, data structure, sorting, searching*. Addison-Wesley-Longman, 1998.