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. 1.1 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. 3 introduce los conceptos de análisis asintótico y compara ordenes de crecimiento con la idea de formar intuición.
- En el Cap. 4 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. 5 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. 6 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. 7 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 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.

Licencia

1 Julia como lenguaje de programación para un curso de algoritmos

Nuestro objetivo es 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

El código se suele organizar en scripts, módulos y paquetes. Cada uno de estos define tipos y funciones que interactuan para componer las soluciones deseadas.

El resto de esta unidad esta dedicada a precisar la sintaxis del lenguaje y anotaciones de importancia sobre su funcionamiento.

1.2 Instalación

El sitio oficial recomienda el uso de `juliaup`, una herramienta que permite manejar diferentes versiones de Julia y mantenerlas actualizarlas.

https://julialang.org/install/

Las versiones de Julia siguen el paradigma de *semantic versioning* (semver), por lo que `juliaup` permite gestionarlas de manera simple y efectiva. La versión estable es la 1.10 y las más nuevas son la 1.11 y la 1.12.

También es posible usar Colab de Google con el kernel para `Julia`; este usar julia 1.11 y hasta el momento, es el único disponible.

1.3 Manos a la obra

Una vez instalado, se puede ejecutar un REPL de Julia en la terminal ejecutando

```{bash}
$ julia
```

dado que instalamos con `juliaup` podemos mantener diferentes versiones, e.g.,

```{bash}
$ juliaup list
```

que nos mostrará una larga lista de posibles *channels* o versiones de instalación

```{bash}

$ juliaup add 1.10
$ juliaup default 1.10
```

estas instrucciones añadirán la versión 1.10 y la establecerá como versión o canal por omisión. Puedes llamar diferentes versiones ejecutando `julia +channel` como sigue:

```{bash}
$ julia +1.12
```

```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.12.1 (2025-10-17)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org release
|__/                   |

julia>
```

```

### 1.3.1 Creando el famoso "Hola mundo"

Uno de los programas más comunes es el siguiente

```
println("¡Hola !")
```

```
¡Hola !
```

1.3.2 Colab

Es posible usar Colab para reducir la complejidad de la instalación, ya que cuenta con un kernel de Julia. Como se mencionaba anteriormente, solo se soporta la versión `1.11` que es subóptima con las versiones de paquetes que usaremos más adelante. Adicionalmente, se tiene limitante de los recursos limitados que se nos proporcionen, en particular al momento de escribir estas notas, aunque los recursos que se otorgan suelen ser suficientes para pruebas, no lo son en otros ámbitos: solo se tienen 2 vcpus y tiempos de ejecución limitados.

1.3.3 Jupyter

Una vez instalado julia; debemos instalar el paquete `IJulia` que instalará todo lo necesario para correr Jupyter (ver la sección de `Pkg` al final de esta unidad para más información sobre paquetes). Una vez corriendo, se debe seleccionar crear un notebook específicando el kernel de Julia para utilizarlo.

1.3.4 Ejercicios

Según sus posibilidades de equipo:

- Instale Julia 1.10, luego instale el paquete `IJulia` y ejecute Jupyter.
- Cree un notebook con Colab con el kernel de Julia.
```

1.4 Sintaxis y estructuras

1.4.1 Funciones

Las funciones son centrales en Julia. Por ahora veremos la estructura y más adelante, definiremos algunas.

Para ejecutar una función se utiliza la sintaxis `fun(arg)`, esta regresará un valor, que depende de la función misma y muchas veces del tipo que tenga `arg`. Si fueran dos argumentos `fun(arg1, arg2)`, etc. También se soportan argumentos con nombre `fun(arg1, arg2, ...; kwarg=val)` (*kwargs* para nombrarlos de manera sintética). En este caso, los *kwargs* no influyen en los tipos de salida. Esto puede parecer extraño pero es debido a las decisiones de implementación relacionadas con el desempeño.

Las funciones se definen como sigue:

```
function fun(arg1, arg2...)                                    ①
    # ... expresiones ...
end

function fun(arg1, arg2...; kwarg1=valor1, kwargs2...) ②
    # ... expresiones ...
end

fun(arg1, arg2...; kwarg1=valor1, kwargs2...) = expresion ③

(arg1, arg2...; kwarg1=valor1, kwargs2...) -> expresion ④

fun() do x                                                     ⑤
    x^2 # ... expresiones ...
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.

1.4.2 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 el in col
    ...
end

while cond
    ...
end

if cond
    ...
elseif cond
    ...
else
    ...
end

function fun(...)
    ...
end

try
    ...
catch
    ...
finally
    ...
end
```

entre las más utilizadas; claramentre hay infinidad de formas de componerlas para formar los algoritmos que se esten escribiendo.

1.4.3 Comentarios

Los comentarios en Julia se hacen por linea o por bloque.

Para comentar una linea se usa el carácter hash `#` y define una linea comentada desde ese punto hasta el salto de linea

```
# los siguientes son comentarios de linea completos, por lo que no se imprimirán
#println(:hola === :hola)
```

```
#println(typeof(:hola))
println(Symbol("hola mundo"))  # este sí se imprimirá, pero este comentario no
```

```
hola mundo
```

Para comentar por bloque, dicho bloque se encierra entre `#= ... =#`

```
println("hola mundo!" #=Symbol("hola mundo")=#)
```

```
hola mundo!
```

1.4.4 Documentación

La documentación oficial se encuentra en https://docs.julialang.org; la cuál cubre el lenguaje y las bibliotecas estandar. Fuera de eso, habrá que ver los sitios y documentaciones de cada paquete, que casi siempre estan en *github*.

Actualmente los chatbots como ChatGPT y Gemini también pueden ser una buena fuente de información sobre el API y las formas de trabajo. Nota: siempre se debe corroborar la información, ya que suelen alucinar.

La manera *empotrada* de consultar la documentación sobre una función es con el prefijo `?` en el REPL.

1.4.4.1 Ejemplo

Nota: Las ligas que salen estan rotas ya que no se adjunta la documentación en este manuscrito.

```
?println("holi")
```

```
println([io::IO], xs...)
```

Print (using `print`) `xs` to `io` followed by a newline. If `io` is not supplied, prints to the default output stream `stdout`.

See also `printstyled` to add colors etc.

2 Examples

```
julia> println("Hello, world")
Hello, world

julia> io = IOBuffer();

julia> println(io, "Hello", ',', " world.")

julia> String(take!(io))
"Hello, world.\n"
```

Esto también indica que debemos documentar nuestro código, en particular se hace de la siguiente forma

```
```{julia}

"""
    fun(...)

Esta función es un ejemplo de documentación
"""
function fun(...)
    ...
end
```
```

2.0.0.1 Definición de variables

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/memoria)
`a != b`	comparación de desigualdad
`a !== b`	comparación de desigualdad (a nivel de tipo/memoria)

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

2.0.0.2 Literales

Los valores literales son valores explicitos que Julia permite para algunos tipos de datos, y que permiten definirlos de manera simple; permitiendonos escribir 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 símbolos 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
```

2.0.1 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
```

2.0.1.1 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 ...expresiones... 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"#5#6"}(var"#5#6"(), 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
```

2.0.2 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
```

2.0.3 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])
```

2.1 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.

```
let
    e = 1.1
    println(e*e)
    @code_typed e*e
end
```

```
1.2100000000000002
```

```
CodeInfo(
1   %1 = intrinsic 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:497 within `*`
define double @"julia_*_12168"(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   "*"
    .section    .ltext,"axl",@progbits
    .globl  "julia_*_12284"                 # -- Begin function julia_*_12284
    .p2align    4, 0x90
```

```
    .type   "julia_*_12284",@function
"julia_*_12284":                        # @"julia_*_12284"
; Function Signature: *(Float64, Float64)
;   @ float.jl:497 within `*`
# %bb.0:                                # %top
    #DEBUG_VALUE: *:x <- $xmm0
    #DEBUG_VALUE: *:y <- $xmm1
    push    rbp
    mov rbp, rsp
    vmulsd  xmm0, xmm0, xmm1
    pop rbp
    ret
.Lfunc_end0:
    .size   "julia_*_12284", .Lfunc_end0-"julia_*_12284"
;
                                        # -- End function
    .type   ".L+Core.Float64#12286",@object # @"+Core.Float64#12286"
    .section    .lrodata,"al",@progbits
    .p2align    3, 0x0
".L+Core.Float64#12286":
    .quad   ".L+Core.Float64#12286.jit"
    .size   ".L+Core.Float64#12286", 8

.set ".L+Core.Float64#12286.jit", 128506429123056
    .size   ".L+Core.Float64#12286.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.

2.2 Ejemplos de funciones

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

```
function f(x)
    x^2
end
```

```
f (generic function with 1 method)
```

Siempre regresan el valor de la última expresión; note como el tipo (y no solo el valor) de retorno depende del tipo de la entrada, e.g., si `x` es un entero entonces `x^2` será un entero, pero si `x` es una matriz, `x^2` será una matriz.

Hay valores opcionales y kwargs, ambas tienen características diferentes:

```
function f(x, t=1)
    (x+t)^2
end

function g(x; t=1)
    (x+t)^2
end
```

```
g (generic function with 1 method)
```

2.3 Definición de estructuras

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

La idea suele ser que todo se use de manera armoniosa

```
"""
  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)))
```

```
(2.0f0, 1.4142135f0)
```

2.4 Arreglos

Una matriz aleatoria de 4×6 se define como sigue

```
A = rand(Float32, 4, 6)
```

```
4×6 Matrix{Float32}:
 0.13498   0.269399  0.60038   0.164158  0.201783  0.220873
 0.550795  0.721219  0.402523  0.374848  0.48225   0.669627
 0.240371  0.666591  0.443133  0.456626  0.551546  0.411505
 0.197902  0.462914  0.140391  0.811499  0.179973  0.604056
```

Un vector aleatorio de 6 dimensiones sería como sigue:

```
x = rand(Float32, 4)
```

```
4-element Vector{Float32}:
 0.09749919
 0.87706774
 0.6801365
 0.7423233
```

entonces podriamos multiplicar `x` con `A` como sigue:

```
y = x' * A
```

```
1×6 adjoint(::Vector{Float32}) with eltype Float32:
 0.806637  1.45583  0.817183  1.25773  0.951365  1.33713
```

```
y'
```

```
6-element Vector{Float32}:
 0.8066374
 1.4558284
 0.8171829
 1.257734
 0.9513647
 1.3371279
```

También existen otras formas para realizarla, aunque no suelen ser la mejor idea si se tienen alternativas canónicas:

```
using LinearAlgebra
```

```
dot.(Ref(x), eachcol(A))
```

```
6-element Vector{Float32}:
 0.8066374
 1.4558284
 0.8171829
 1.257734
 0.9513647
 1.3371278
```

Este ejemplo muestra la técnica *broadcasting* que aplica una función a una colección; se indica añadiendo un punto al final del nombre de la función. Adicionalmente, hay una serie de reglas que se deben seguir para el manejo de las colecciones. La función `eachcol` crea un iterador sobre cada columna de la matriz `A` y `Ref(x)`, nos permite que el `broadcasting` reconozca al vector `x` como un único elemento en lugar de una colección de valores.

2.4.1 Ejercicios

Dados dos vectores, cree las siguientes funciones:

1. Calcule el coseno entre dos vectores u, v, de dimensión d:
   - $cos(u, v) = \frac{\langle u,v \rangle}{\|u\|\|v\|}$; donde $\langle u, v \rangle = \sum_i^d u_i \cdot v_i$ y $\|\cdot\|$ es la norma de un vector.
2. Calcule la distancia Euclidea entre dos vectores u, v:
   - $euclidean(u, v) = \sqrt{\sum_i^d (u_i - v_i)^2}$.

2.5 Paquetes y módulos

El ecosistema de paquetes de Julia es una de sus mayores fortalezas, impulsado por usu gestor de paquetes *Pkg*, el cual viene integrado en el REPL y en la su instalación mínima; es muy robusto. Se encarga de la instalación y actualización de librerías, así como también garantiza la reproducibilidad de los proyectos. Cada ambiente de trabajo en Julia utiliza archivos como `Project.toml` y `Manifest.toml` para registrar la paquetería usada, así como las versiones necesarias de todas las dependencias.

2.5.1 Pkg en REPL

Para entrar al modo Pkg desde cualquier sesión de Julia en el REPL se debe teclear corchete que cierra `]`.

El prompt del REPL cambiará:

Modo	Prompt
Normal (Julia)	`julia>`
Modo Pkg	`(@v1.10) pkg>`

Ahora se puede ver qué paquetes están instalados en tu entorno actual.

Comando	Acción
`st` o `status`	Muestra la lista de todos los paquetes instalados y sus versiones específicas.

Para añadir una *paquete* a tu entorno, usa el comando `add`.

Comando	Acción
`add Paquete`	Descarga e instala el paquete.

En el caso de que un paquete ya no sea necesario, se puede desinstalar con el comando `rm` (de *remove*).

Comando	Acción
`rm Paquete`	Elimina el paquete del entorno actual.

Finalmente, es posible actualizar paquetes de manera individual o colectiva usando el comando `up`.

Comando	Acción
`up` o `update`	*Actualiza* todos los paquetes instalados a su última versión compatible.
`up Paquete` o `update Paquete`	Actualiza `Paquete` a su última versión compatible.

2.5.1.1 Manejo de ambientes

El entorno o ambiente (*environment*) se puede especificar de manera global o por diretorio, y nos sirve para aislar las aplicaciones y no entrar en dificultades por versiones.

Comando	Acción
`activate dir`	Activa el directorio `dir` como ambiente.

Supongamos que nos comparten un proyecto escrito en julia, lo primero que debemos hacer es activar e instanciar el ambiente; la instanciación es como sigue

Comando	Acción
`instantiate`	Se instalan todos los paquetes indicados por el ambiente.

Muchas veces también cambiamos paquetes locales que requiren reactualizar el ambiente, eso se consigue con `] resolve` que actualizará las nuevas dependencias que cambiaron.

2.5.1.2 Saliendo del modo Pkg

Para volver al modo de ejecución de código normal de Julia, se debe presionar *backspace*.

El prompt cambiará de nuevo a `julia>` y podrás usar los paquetes que instalaste con el comando `using`.

```
using DataFrames, CSV
```

esto traera el paquete al entorno en memoria haciendo accesibles sus métodos y estructuras públicas.

2.5.2 Usando `Pkg` desde el modo normal de Julia (fuera del modo Pkg del REPL)

Existe un paquete interno de las instalaciones de julia llamado `Pkg` que es el que maneja todo lo anterior, este puede ser utilizado como cualquier paquete. Basicamente tiene funciones similares a las del modo `Pkg` (con nombres completos).

Ejemplos:

```
julia> import Pkg
julia> Pkg.add("PlotlyLight")
julia> Pkg.add(["CSV", "DataFrames"])
julia> Pkg.rm("PlotlyLight")
julia> Pkg.update()
julia> Pkg.status()
```

Ahora para el manejo de los ambientes:

```
julia> import Pkg
julia> Pkg.activate(".")
julia> Pkg.instantiate()
julia> Pkg.add("Statistics")
```

2.6 Otras estrategias para la organización de código

La función `include("nombre_archivo.jl")` es el método más simple en Julia para organizar código en múltiples archivos. Su función es equivalente a *copiar y pegar* el contenido del archivo especificado directamente en la línea donde se llama a `include`.

Sirve para estructurar grandes *scripts* en archivos más pequeños y manejables; el código incluido se ejecuta en el mismo *alcance* (scope) donde se llamó a `include`. Si llamas a `include` en el alcance global, las funciones y variables definidas en el archivo incluido se vuelven globales. Si lo llamas dentro de un módulo, se vuelven parte de ese módulo.

Es simple, pero no proporciona aislamiento, y puede generar conflictos de nombres si no se usa de manera adecuada.

Por otro lado, los *módulos* permmiten crear *espacios de nombres (namespaces)* aislados y bien definidos, utiles para organizar proyectos grandes y complejos.

2.6.1 Aislamiento y alcance (Scoping)

Un módulo actúa como una *caja* que encierra sus funciones y variables. Todo lo que se define dentro de un módulo es *privado* por defecto para evitar conflictos de nombres con código externo.

```
module MiCalculadora
    # Esta función es PRIVADA
    function interna(x)
        return x * 2
    end

    # Esta función se hace PÚBLICA con 'export'
    export sumar

    function sumar(a, b)
        return a + b
    end
end
```

Para que las funciones, tipos o constantes dentro de un módulo sean accesibles desde afuera, deben ser explícitamente *exportadas* utilizando la palabra clave `export`.

Los modulos pueden anidarse.

Para utilizar las funciones de un módulo en otro script o en el REPL, se usan dos comandos principales:

Comando	Acción
`using NombreModulo`	Importa solo los símbolos que han sido **exportados** por el módulo.

Comando	Acción
`import NombreModulo`	Importa el módulo completo. Para usar sus funciones, debes prefijarlas (ej: `NombreModulo.sumar(1, 2)`).

En la práctica, un paquete o un proyecto grande de Julia casi siempre usa tanto `include` como módulos. De esta manera, `include` ayuda a la organización de archivos, mientras que el bloque `module` garantiza que todo el código esté contenido en un espacio de nombres único y limpio, evitando colisiones.

En particular, los *paquetes* pueden verse como la preparación de un módulo para su distribución, indicando los paquetes que usan (dependencias) y sus versiones especificas para los cuales fueron diseñados. También suelen incluir documentación y pruebas unitarias.

2.7 Recursos para aprender más sobre el lenguaje

- 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/

3 Introducción al análisis de algoritmos con Julia

Este capítulo introduce a los fundamentos de análisis de algoritmos. Se introduce el concepto de modelo de cómputo y la notación asintótica, preparandonos para usar 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 catalogarlos con respecto a los recursos de computo necesarios para ejecutarlos.

3.1 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 delinean la manera en como se diseñan los programas de computadoras.

Es común encontrar que un problema puede ser resuelto por múltiples algoritmos, 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.

3.2 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:

- La máquina de Turing.
- Funciones recursivas.
- Cálculo lambda.
- Máquina de acceso aleatorio (RAM).

Todas estos modelos son *equivalentes* en sus capacidades, pero sus diferentes planteamientos permiten enfocarse en diferentes aspectos de los problemas.

3.2.1 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 mediante un cabezal en una cinta *infinita* (ver Figura 3.1) 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.

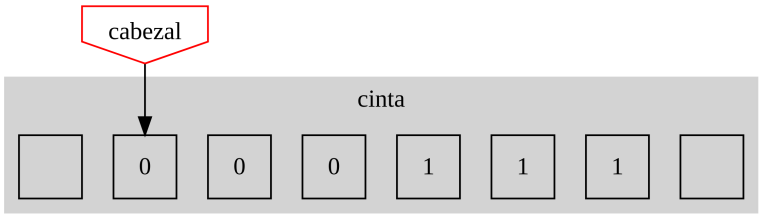


Figura 3.1: Cabezal y cinta.

Una máquina de Turing se puede escribir como una tupla de 7 elementos $M = (Q, \Sigma, \Gamma, s, \epsilon, F, \delta)$ donde:

- Q es un conjunto finito de estados.
- Σ es el alfabeto de entrada, i.e., un conjunto finito de símbolos que no incluye el espacio en blanco.
- Γ es el alfabeto de la cinta , i.e., un conjunto finito de símbolos $\Sigma \subseteq \Gamma$.
- $s \in Q$ es el estado inicial.
- $\epsilon \in \Gamma$ es un símbolo especial denominado *blanco*; y puede llenar la cinta al infinito.
- $F \subseteq Q$ conjunto de estados finales de aceptación.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, es la función de transición, i.e., una función parcial que índica que se debe hacer al leer un símbolo en la posición actual de lectura, esto es, qué se escribe, hacia que estado se cambia, y la dirección de movimiento de la cinta (siempre se mueve en una celda).

Hay muchas variantes de la definición de máquina de Turing, por ejemplo, una cinta especial para lectura y una para escritura, diferentes formas de definir las transiciones, símbolos para no moverse, etc. Hasta ahora, todas las formas son a lo más equivalentes en términos de poder de cómputo, la diferencia viene en la expresividad para definir soluciones.

Es cómun plantear los problemas en forma de lenguajes; esto es, en términos de *dada la cadena S ¿la máquina M la acepta?*, dónde *acepta* quiere decir que la máquina es capaz de ir desde el estado inicial al estado de aceptación leyendo/transformando

S. Si M no termina en un estado de aceptación, entonces se implica el fallo o respuesta negativa.

> **Nota**
>
> Algunas causas de fallo son que la función de transición no este definida para un estado y símbolo particular y el no alcanzar un estado de salida aceptación.

Por ejemplo, sea M_{01} una máquina capaz de detectar $n \geq 0$ ceros terminados por un único 1. Entonces

$$M_{01} = (\{q_0, q_1, q_2\}, \{\mathtt{0}, \mathtt{1}\}, \{\mathtt{0}, \mathtt{1}, \mathtt{X}, \}, q_0, , \{q_2\}, \delta_{01})$$

donde la función de transición se define como:

$$\delta_{01}(q_0, \mathtt{0}) = (q_0, \mathtt{X}, \mathtt{R}) \tag{3.1}$$
$$\delta_{01}(q_0, \mathtt{1}) = (q_1, \mathtt{X}, \mathtt{R}) \tag{3.2}$$
$$\delta_{01}(q_1,) = (q_2, , \mathtt{R}) \tag{3.3}$$
$$\tag{3.4}$$

Dada la regularidad de la función de transición, es común escribirla como una tabla:

entrada	salida
$(q_0, \mathtt{0})$	$(q_0, \mathtt{X}, \mathtt{R})$
$(q_0, \mathtt{1})$	$(q_1, \mathtt{X}, \mathtt{R})$
$(q_1,)$	$(q_2, , \mathtt{R})$

Una máquina de Turing se puede representar mediante un autómata

Supongamos ahora el problema de reconocer cadenas 0^n1^n; para esto podemos definir la siguiente máquina de Turing

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{\mathtt{0}, \mathtt{1}\}, \{\mathtt{0}, \mathtt{1}, \mathtt{X}, \mathtt{Y}, \}, q_0, , \{q_4\}, \delta),$$

donde la función de transición es como sigue:

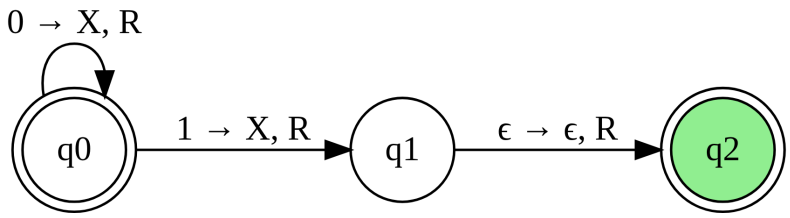


Figura 3.2: Ejemplo de máquina de Turing que reconoce cadenas 0^n1

$$
\begin{aligned}
\delta(q_0, \texttt{0}) &= (q_1, \texttt{X}, \texttt{R}) \\
\delta(q_0, \texttt{Y}) &= (q_3, \texttt{Y}, \texttt{R}) \\
\delta(q_1, \texttt{0}) &= (q_1, \texttt{0}, \texttt{R}) \\
\delta(q_1, \texttt{1}) &= (q_2, \texttt{Y}, \texttt{L}) \\
\delta(q_1, \texttt{Y}) &= (q_1, \texttt{Y}, \texttt{R}) \\
\delta(q_2, \texttt{0}) &= (q_2, \texttt{0}, \texttt{L}) \\
\delta(q_2, \texttt{X}) &= (q_0, \texttt{X}, \texttt{R}) \\
\delta(q_2, \texttt{Y}) &= (q_2, \texttt{Y}, \texttt{L}) \\
\delta(q_3, \texttt{Y}) &= (q_3, \texttt{Y}, \texttt{R}) \\
\delta(q_3,) &= (q_4, , \texttt{R})
\end{aligned}
$$

Todo inicia con una cadena de la forma esperada, e.g., `000111`, la idea general del algoritmo es aparear `0`'s y `1`'s, ya que solo es valido si ambas subcadenas tienen igual longitud. Para esto se marcan los `0`'s vistos con `X` y los `1`'s con `Y`. La máquina estará entonces marcando las primeras ocurrencias y moviendose a traves de la cinta para encontrar los correspondientes.

> **!** Importante
>
> Las máquinas de Turing son capaces del representar cualquier problema cómputable con una analogía mecánica, lo cual hace evidente su implementación en el mundo físico.

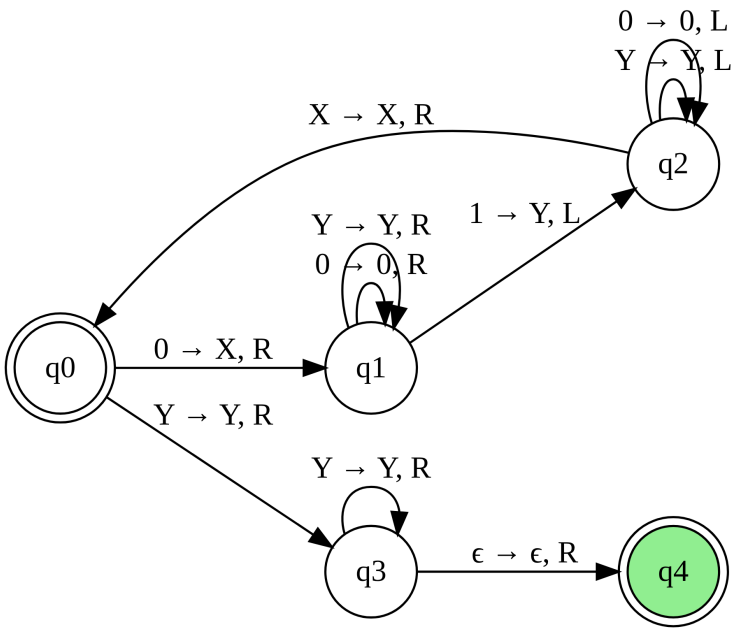


Figura 3.3: Ejemplo de máquina de Turing que reconoce cadenas $0^n 1^n$

3.2.2 Modelos funcionales

- 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.

3.2.3 Máquina de acceso aleatorio (RAM)

Es un módelo que describe una computadora con registros. A diferencia 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 como funciona una computadora moderna entre los que se hemos revisado.

Una computadora moderna difieres de una máquina RAM, por ejemplo, a diferencia de una computadora física se suponen infinitos registros con precisión infinita. Debido a los costos de los semiconductores y la energía necesaría, es conveniente

construir computadoras con una jerarquía de memoria: los niveles con mayores prestaciones (e.g., rápidez) son los más escasos. 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 y lectura de datos.

3.3 Sobre la importancia del modelo de cómputo en el curso

En este curso nos enfocaremos en especificaciones de alto nivel, donde los algoritmos son convenientes para una computadora física. Sin embargo, 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 como magnitud física medible, así como en la cantidad de memoria consumida, por lo que se aboradará el costo realizando mediciones experimentales. Se contrastará con el análisis basado en conteo de operaciones siempre que sea posible.

3.4 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. Después, 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*.

3.5 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, en este caso, cuando el tamaño de la entrada es realmente grande. Es importante

mencionar, que no se esperan entradas de tamaño descomunal, ni tampoco se espera cualquier tipo de entrada.

3.5.1 Notación Θ

Para una función dada $g(n)$ denotamos por $\Theta(g(n))$ el siguiente conjunto de funciones:

$$\Theta(g(n)) = \{f(n) \mid \text{ existen las constantes positivas } c_1, c_2 \text{ y } n_0 \text{ tal que} \tag{3.5}$$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todo } n \geq n_0\} \tag{3.6}$$

$$\tag{3.7}$$

esto es, una función $f(n)$ pertenece al conjunto $g(n)$ si $c_1 g(n)$ y $c_2 g(n)$ pueden *cubrirla* por abajo y por arriba, para esto deben existen las constantes positivas c_1 y c_2 y una n lo suficientemente larga, e.g., para eso la constante n_0. La notación propiamente de conjuntos puede usarse $f(n) \in \Theta(g(n))$ pero es común en el área usar $f(n) = \Theta(g(n))$ para expresar la pertenencia; este abuso de la notación tiene ventaja a la hora de plantear los problemas de análisis.

3.5.2 Notación O

Se utiliza para indicar una cota asintótica superior. Una función $f(n)$ se dice que esta en $O(g(n))$ si esta en el siguiente conjunto:

$$O(g(n)) = \{f(n) \mid \text{ existen las constantes positivas } c \text{ y } n_0 \text{ tal que} \tag{3.8}$$

$$0 \leq f(n) \leq cg(n) \text{ para todo } n \geq n_0\} \tag{3.9}$$

$$\tag{3.10}$$

La notación O se usa para dar una cota superior, dentro de un factor constante. Al escribir $f(n) = O(g(n))$ se indica que $f(n)$ es miembro del conjunto $O(g(n))$; hay que notar que $f(n) = \Theta(g(n))$ implica que $f(n) = O(g(n))$, i.e., $\Theta(g(n)) \subseteq O(g(n))$.

3.5.3 Notación Ω

Al contrario de O, la notación Ω da una cota asintótica inferior. Una función $f(n)$ se dice que esta en $\Omega(g(n))$ si esta en el siguiente conjunto:

$$\Omega(g(n)) = \{f(n) \mid \text{ existen las constantes positivas } c \text{ y } n_0 \text{ tal que} \tag{3.11}$$

$$0 \leq cg(n) \leq f(n) \text{ para todo } n \geq n_0\} \tag{3.12}$$

$$\tag{3.13}$$

Dado que la Ω define una cota superior, basicamente si $f(n) = \Omega(g(n))$, entonces $f(n)$ debe estar por encima de $g(n)$ con las constantes c y n_0 adecuadas. Al igual que la notación O, la notación Ω es menos estricta que Θ, esto es $f(n) = \Theta(g(n))$ implica que $f(n) = \Omega(g(n))$, por lo que $\Theta(g(n)) \subseteq \Omega(g(n))$.

Por tanto, si $f(n) = O(g(n))$ y $f(n) = \Omega(g(n))$ entonces $f(n) \in \Theta(g(n))$.

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.

3.5.4 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:

3.5.5 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.

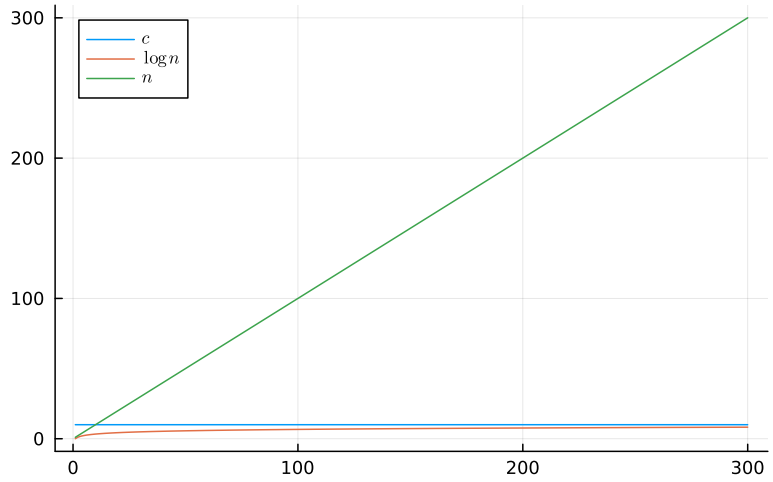
Los ordenes de crecimiento son maneras de categorizar la velocidad de crecimiento de una función, y para nuestro caso, de una función de costo. Junto con la notación asimptótica nos permite concentrarnos en razgos gruesos que se mantienen para entradas grandes, más que en los detalles, y no perder el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

3.5.5.1 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")
```

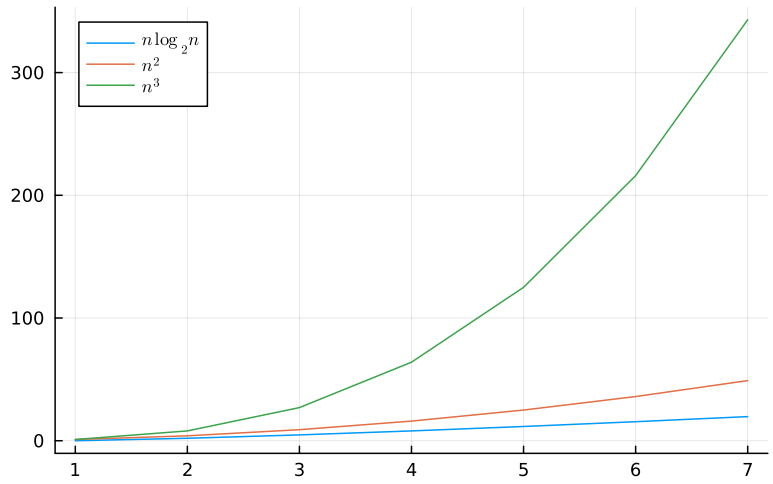


3.5.5.2 Costo $n \log n$ y polinomial

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.

```
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")
```

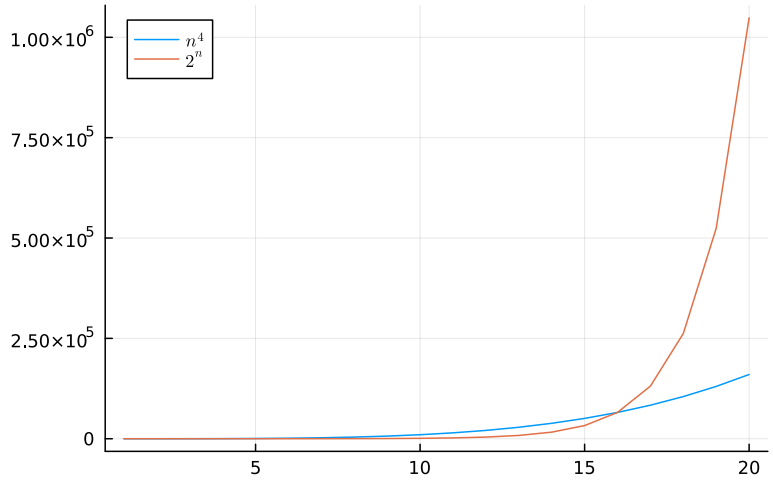


3.5.5.3 Exponencial

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.

```
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")
```

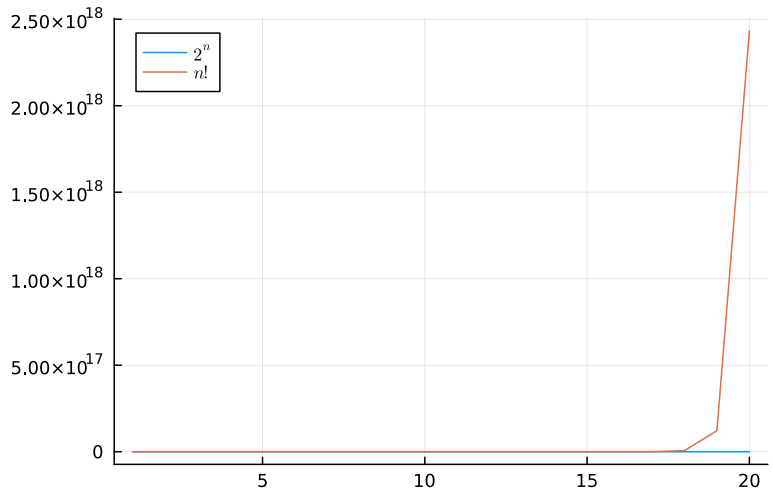


3.5.5.4 Crecimiento factorial

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.

```
n = 20

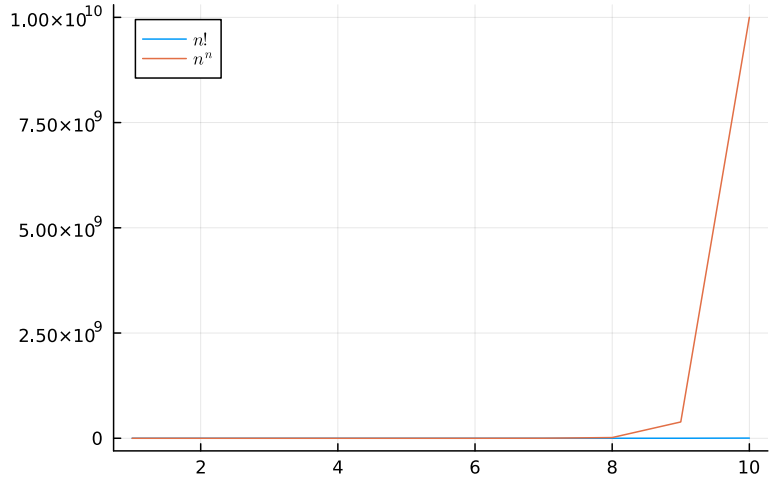
plot(1:n, [2^x for x in 1:n], label=L"2^n")
plot!(1:n, [factorial(x) for x in 1:n], label=L"n!")
```



3.5.5.5 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.

3.6 El enfoque experimental

La notación asintótica nos permite alcanzar un lenguaje común y preciso sobre los costos de problemas y algoritmos; es de especial importancia para la evaluación de las alternativas en la literatura especializada, y elegir algoritmos aún sin la necesidad de implementación. El análisis asintótico da la posibilidad de conocer el desempeño desde diferentes perspectivas como peor caso o caso promedio, utilizando un módelo de computación, y siempre pensando en entradas lo suficientemente grandes.

En la práctica, existe una múltitud de razones por los cuales los problemas que se resuelven podrian no ser tan grandes como para que un algoritmo domine a otros de manera asintótica, las instancias podrían no ser tan generales como para preocuparse en el peor caso, o el caso promedio general. En muchas situaciones, es importante sacar provecho de los casos *fáciles*, sobre todo cuando el problema a resolver podría asegurar que dichos casos simples sean abundantes. Dada la complejidad detrás de definir sub-conjuntos de instancias y llevar a cabo un análisis formal, se vuelve imperativo realizar pruebas experimentales.

Por otra parte, dada la complejidad de una computadora moderna, es necesario realizar evaluaciones experimentales de los algoritmos que tengan una complejidad similar. Las computadoras reales tienen una jerarquia de memoria con tamaños y velocidades de acceso divergentes entre sí, con

optimizaciones integradas sobre la predicción de acceso y cierto nivel de paralelismo. Incluso, cada cierto tiempo se obtienen optimizaciones en los dispositivos que podrían mejorar los rendimientos, por lo que es posible que con una generación a otra, lo que sabemos de los algoritmos y su desempeño en computadoras y cargas de trabajo reales cambie.

3.6.1 Metodología experimental

Algunos de los algoritmos que se verán en este libro son sumamente rapidos en la práctica para resolver una instancia práctica por lo que medir el desempeño de instancias solas podría no tener sentido. La acumulación de operaciones es fundamental, así como la diversidad de las instancias también lo es. Caracterizar las entradas es de vital importancia ya que la adaptabilidad a las instancias es parte de los objetivos.

Entonces, estaremos probando conjuntos de instancias, caracterizadas y estaremos utilizando tiempos promedios. También estaremos usando conteo de operaciones, por lo que los algoritmos en cuestión muchas veces serán adaptados para poder realizar este conteo.

En Julia etaremos utilizando las siguientes instrucciones:

- `@time expr` macro que mide el tiempo en segundo utilizado por `expr`, también reporta el número de alojaciones de memoria. Note que reducir la cantidad de memoria alojada puede significar reducir el tiempo de una implementación, ya que el manejo de memoria dinámica es costoso.
- `@benchmark expr params` macro del paquete `BenchmarkTools` que automatiza la repetición de `expr` para obtener diferentes mediciones y hacer un reporte, `params` permite manipular la forma en que se reliza la evaluación.
- `@btime expr params` macro del paquete `BenchmarkTools` que mimetiza la salida de `@time`.

```
a = rand(Float32, 3, 3)
@time a * a                                    ①
@time a * a                                    ②
```

① Todas las fuciones se deben compilar, la primera llamada uncluye los costos de compilación.
② El costo sin compilación, hay una alojación que es la matriz donde se guarda el resultado.

```
  0.283588 seconds (810.30 k allocations: 36.842 MiB, 5.23% gc time, 99.92% compilation time)
  0.000009 seconds (2 allocations: 112 bytes)
```

```
3×3 Matrix{Float32}:
 0.929955  0.857912  0.48801
 1.03075   1.2294    0.852178
 0.527486  0.790568  1.02193
```

Tanto `@benchmark` como `@btime` aceptan interpolación de variables con el prefijo `$` para controlar la evaluación de una expresión se debe contar como parte de lo que se quiere medir o no. Se puede combinar con el parametro `setup` para controlar de manera precisa las entradas para evaluar cada una de las repeticiones de `expr`.

```
using BenchmarkTools

@benchmark a * a setup=(a=rand(Float32, 3, 3))
```

```
BenchmarkTools.Trial: 10000 samples with 985 evaluations per sample.
 Range (min … max):  52.649 ns …  20.790  s    GC (min … max):  0.00% … 99.46%
 Time  (median):     61.157 ns                 GC (median):     0.00%
 Time  (mean ± ):   78.077 ns ± 330.801 ns    GC (mean ± ):  16.48% ±  4.29%
```

```
  52.6 ns         Histogram: frequency by time          103 ns <

 Memory estimate: 112 bytes, allocs estimate: 2.
```

```
a = rand(Float32, 3, 3)
@btime a * a setup=(a=$a)
```

```
  50.558 ns (2 allocations: 112 bytes)
```

```
3×3 Matrix{Float32}:
 0.799493  1.47276  1.23069
 0.62748   1.60687  1.42717
 0.508446  1.57522  1.689
```

El parametro `sample` controla el número máximo de muestras que se tomarán para el análisis, y `seconds` limita el tiempo sobre el cual se tomarán muestras; se asegura que al menos se tomará una muestra, se debe tener en cuenta que puede costar más que `seconds`.[1]

```
a = rand(Float32, 3, 3)
b = rand(Float32, 3, 3)
@benchmark a * b setup=(a=$a, b=$b) samples=1000 seconds=0.33
```

```
BenchmarkTools.Trial: 1000 samples with 979 evaluations per sample.
 Range (min … max):  66.323 ns … 123.906 ns    GC (min … max): 0.00% … 0.00%
 Time  (median):     76.668 ns                 GC (median):    0.00%
 Time  (mean ± ):   81.461 ns ±  13.397 ns    GC (mean ± ):  0.00% ± 0.00%


  66.3 ns         Histogram: frequency by time          104 ns <

 Memory estimate: 112 bytes, allocs estimate: 2.
```

3.6.2 Ejemplo del cálculo de máximo de un arreglo y diferentes tipos de costo.

—

```
function maximo(col)                                    ①
    maxpos = 1
    actualizaciones = 1
    i = 2
```

[1] Se recomienda visitar el sitio https://juliaci.github.io/BenchmarkTools.jl/stable/ para más información sobre el paquete `BenchmarkTools`, y en particular para sus parametros, como guardar información de corridas.

```
    while i < length(col)
        if col[maxpos] < col[i]
            maxpos = i
            actualizaciones += 1
        end
        i += 1
    end

    maxpos, actualizaciones
end
```

① Función que encuentra el máximo en una secuencia y devuelve su posición, y además devuelve el número de veces que se actualizó el máximo en el recorrido.

```
maximo (generic function with 1 method)
```

```
a = rand(UInt32, 128)
@benchmark maximo($a) samples=100 seconds=3        ①
```

① Un análisis de desempeño usando `@benchmark`; probando con máximo 100 samples en 3 segundos.

```
BenchmarkTools.Trial: 100 samples with 858 evaluations per sample.
 Range (min … max):  154.934 ns … 181.124 ns    GC (min … max): 0.00% … 0.00%
 Time  (median):     160.637 ns                 GC (median):    0.00%
 Time  (mean ± ):   161.142 ns ±   4.767 ns    GC (mean ± ):  0.00% ± 0.00%



  155 ns        Histogram: log(frequency) by time        175 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

Note que aunque se tiene un análisis muy detallado del desempeño, otras medidas de costo caen fuera del diseño del paquete, por lo que es necesario hacerlas por otros medios. Por ejemplo, suponga que el número de `actualizaciones` es nuesta medida de desempeño, un código donde se capturen las actualizaciones

```julia
using StatsBase                                                    ①
a = [maximo(rand(UInt32, 128))[2] for i in 1:100]                  ②
quantile(a, [0.0, 0.25, 0.5, 0.75, 1.0])                           ③
```

① Inclusión de un paquete para cálculo de estadísticas básicas.
② Definición de 100 experimentos que calculan `maximo` sobre arreglos aleatorios.
③ Cálculo del mínimo, cuantiles 0.25, 0.5, 0.75, y el máximo, para determinar el desempeño.

```
5-element Vector{Float64}:
  2.0
  4.0
  5.0
  7.0
 12.0
```

3.7 Actividades

1. Códifique de manera matemática (e.g., definición de estados y δ) la siguiente máquina de Turing que cálcula la expresión $3n + 1$.

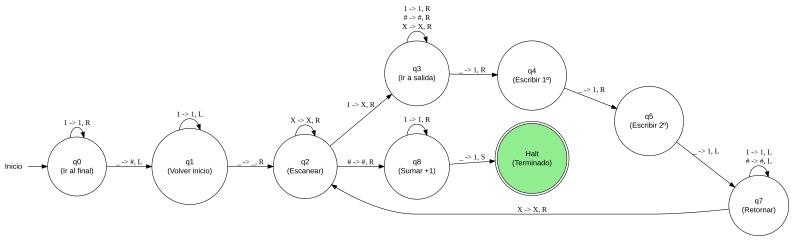


Figura 3.4: Máquina de Turing que cálcula $3n + 1$.

2. 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 $O(n^n)$

a. 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.
b. Use diferentes tamaños de entrada $n = 100$, $n = 1000$, $n = 10000$ y $n = 100000$.
c. Nota: los números pueden ser muy grandes para algunas expresiones, tome decisiones en estos casos y argumente en el reporte.
d. Discuta las implicaciones de costos de cómputo necesarios para manipular grandes volúmenes de información.

3.7.1 Entregable

Su trabajo se entregará en PDF y con el notebook fuente, se entregará en el sistema en archivos separados. El código se debe documentar, así como mantener una estructura del documento 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.
- Actividad 1. Máquina de turing
- Actividad 2. Comparación de los órdenes de crecimiento.
  - Nota: poner el código cercano a la presentación de resultados.
- Conclusiones
- Lista de referencias. Nota, una lista de referencias que no fueron utilizadas en el cuerpo del texto será interpretada como una lista vacía.

Nota: no olvides revisar la rúbrica del curso.

3.8 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

4 Estructuras de datos elementales

4.1 Introducción

En esta unidad se discutirán las propiedades y operaciones básicas de estructuras como conjuntos, listas, pilas, colas, arreglos, vectores, matrices y matrices dispersas. Los ejemplos de código se muestran en el lenguaje de programación Julia, pero que puede ser traducido fácilmente en otros lenguajes de programación.

4.2 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. Un conjunto puede estar vacio ($\emptyset$) o contener elementos, e.g., $\{a, b, c\}$. La operación *unión* $\cup$ construye un nuevo conjunto a partir de otros $\{a, b\} \cup \{c\} = \{a, b, c\}$; la intersección se indica con el operador $\cap$, 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 su negación, i.e., $d \notin \{a, b, c\}$. Es común usar conjuntos mutables en diferentes algoritmos, esto es, que permitan inserciones y borrados sobre la misma estructura; desde el punto de vista de eficiencia, esto puede reducir las operaciones de manipulación de datos así como de la gestión de 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; es una convención, no un operador en sí 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.

Hay múltiples formas de representar conjuntos ya que los requerimientos de los algoritmos son diversos y tener la representación correcta puede ser una diferencia importante en el rendimiento. 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.

4.3 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 amerita, 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:

En algunos lenguajes de programación como Julia, 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

```
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.

```
"""
  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)))
```

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.

```
(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.

4.4 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 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.

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.

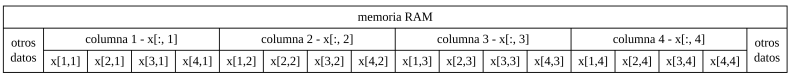


Figura 4.1: Esquema de una matriz en memoria.

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))
```

```
(74, 1.6994965f0)
```

Figura 4.2: Algoritmos para encotrar el vector columna con mayor producto en X punto contra u.

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}:
 1.60465   1.60465   1.60465   0.0       0.0
 0.650574  0.650574  0.650574  0.0       3.6f-44
 1.33931   1.33931   1.33931   0.0       0.0
 1.05112   1.05112   1.05112   3.5f-44   0.0
 0.920721  0.920721  0.920721  0.0       4.0f-45
```

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. Existen diferentes algoritmos para hacer esta operación más eficiente para diferentes casos o características de las matrices, siendo un área de investigación activa.

4.5 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 *head/cabeza* y *tail/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 aleatorio a los elementos; los accesos típicos son a los extremos de la lista (cabeza y cola), sucesor y predecesor.

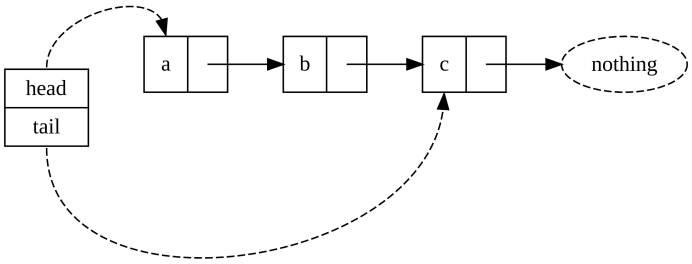


Figura 4.3: Una lista ligada simple

La Figura 4.3 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.

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

```
(10, 20, 30)
```

En el Listado 4.1 se ignora la referencia a *tail* (*head* se guarda en *node*), por lo que las operaciones sobre *tail* requieren recorrer

Listado 4.1 Código para una lista ligada simple

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

node = Node(10, Node(20, Node(30, nothing)))

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

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 primero 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.

Entre las operaciones comunes tenemos las siguientes:

- *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.

4.5.0.1 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*?

4.6 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 $A = (\{a, b, c, d\}, \{(a, b), (b, c), (c, d), (d, a)\})$

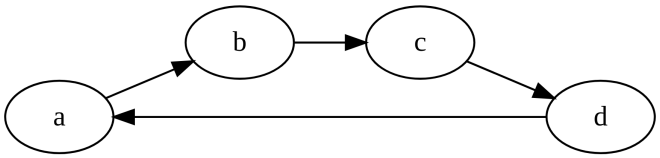


Figura 4.4: Un grafo dirigido simple

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 $peso(a) = 1.5$.
- $costo : V \times V \to \mathbb{R}$, 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.

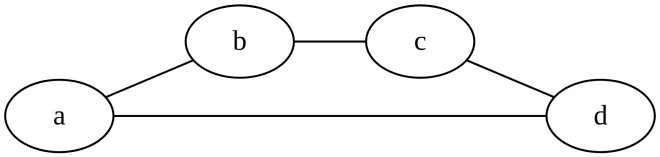


Figura 4.5: Un grafo cuyas aristas no estan dirigidas

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 *regresarse* aristas, es imposible regresar a un vértice antes visto.

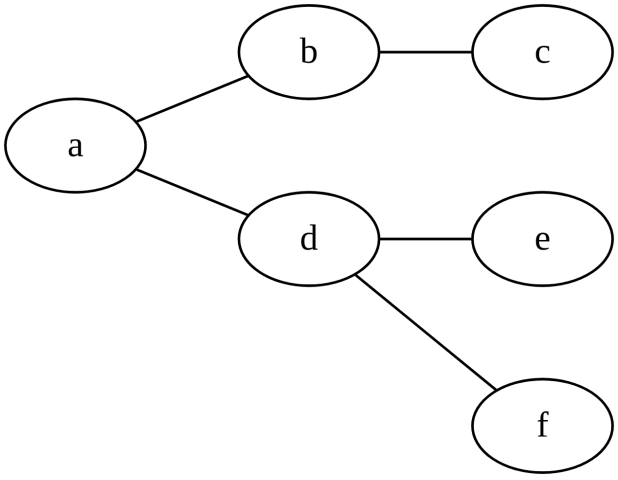


Figura 4.6: Árbol con aristas no dirigidas

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.

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

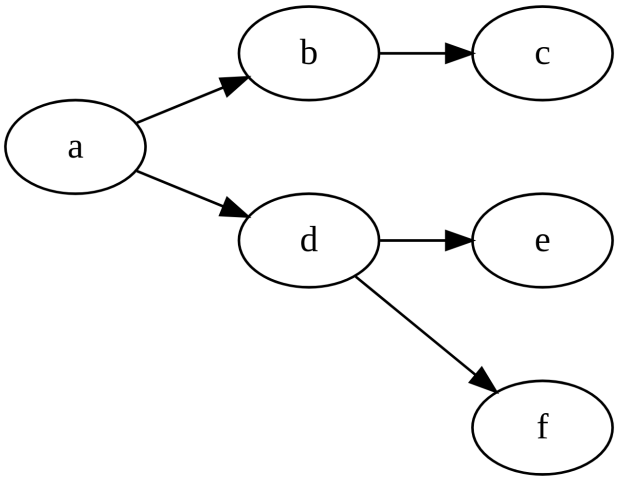


Figura 4.7: Árbol con aristas dirigidas, note que es fácil saber si hay un vértice o nodo que se distinga como raíz, o nodos que sean hojas.

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 *recordarlo* cuando se sale de una visita anidada. Entonces, 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.

4.6.0.1 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.

4.7 Actividades

Dada la matriz Q de tamaño $d \times m$ y la matriz X de tamaño $d \times n$, para cada vector columna $q \in Q$, encontrar el vector columna de $x \in X$ que maximiza su producto punto. Dicho de otra forma, el problema consiste en asociar cada q con $\arg\max\{q \cdot x \mid x \in X\}$. Este será el problema de *encontrar el producto punto máximo* o **top-1**. Obtenga un arreglo de identificadores de columna como resultado (*nns*) y otro con el valor del producto punto máximo (*dots*) para cada columna en Q.

$$\mathbf{Q} = \begin{pmatrix} q_{1,1} & q_{1,2} & \cdots & q_{1,m} \\ q_{2,1} & q_{2,2} & \cdots & q_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ q_{d,1} & q_{d,2} & \cdots & q_{d,m} \end{pmatrix}$$

$$\mathbf{X} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d,1} & x_{d,2} & \cdots & x_{d,n} \end{pmatrix}$$

a. Cree el algoritmo **A1** con operaciones matriciales explicitas.
b. Adapte el algoritmo *getmaxdot* de las notas (Cap. 2) para el mismo lenguaje de tu implementación para resolver el mismo problema, llamemoslo **A2**.

1. Considere un conjunto de datos como una tupla Q, X.
2. Considere las combinaciones de $m \in \{10^3, 10^6, 10^9\}$, $n \in \{10^3, 10^6, 10^9\}$, $d \in \{8, 16, 32\}$.
3. Las matrices Q y X deberán ser aleatorias cuyos vectores columna esten en la esfera unitaria.
4. Analice la memoria necesaria por cada algoritmo (A1 y A2) para cada combinación en argumentos; grafique y discuta.
5. Reporte los costos en tiempo real para $m = 10^3, n = 10^6$ y $d = 16$ para cada A1 y A2.

6. Grafique la distribución del arreglo *dots* para la corrida previa, i.e., $m = 10^3, n = 10^6$ y $d = 16$.

4.7.1 Entregable

Su trabajo se entregará en PDF y con el notebook fuente, se entregará en el sistema en archivos separados. El código se debe documentar, así como mantener una estructura del documento 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.
- Desarrollo de las actividades.
- Conclusiones
- Lista de referencias. Nota, una lista de referencias que no fueron utilizadas en el cuerpo del texto será interpretada como una lista vacía.

Nota: no olvides revisar la rúbrica del curso.

4.8 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.

5 Algoritmos de ordenamiento

Objetivo

Implementar y analizar algoritmos de ordenamiento de arreglos con costo óptimo en el peor caso, así como algoritmos adaptativos a la entrada para caracterizar su desempeño bajo un enfoque experimental para la solución efectiva de problemas informáticos.

5.1 Introducción

En este tema se aborda el ordenamiento basado en comparación, esto es, existe un operador $<$ que es capaz de distinguir si un elemento a es menor que un elemento b.

El operador cumple con las siguientes propiedades:

- si $a < b$ y $b < c$ entonces $a < c$ (transitividad); e.g., $1 < 10$ y $10 < 100$ entonces $1 < 100$.
- tricotomía:
  - si $a < b$ es falso y $b < a$ es falso, entonces $a = b$ (antisímetria); dicho de otras formas:
    - si a no es menor que b ni b menor que a entonces a es igual a b,
    - desvelando variables, $1 < 1$ es falso, el intercambio es obvio, entonces $1 = 1$.
  - en otro caso, $a < b$ o $a < b$.

Sin perdida de generalidad, podemos planter el problema de ordenamiento sin permitir repeticiones como sigue: dado un arreglo $A[1, n] = a_1, a_2, \cdots, a_n$; un algoritmo de ordenamiento obtiene la permutación π tal que $a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}$.

Usar un operador como $<$ es suficiente para crear algoritmos correctos y eficientes, sin embargo, en la práctica y en una computadora real, también es válido utilizar operadores como $=$ o $\leq$, o intercambiar por $>$ y $\geq$ según convenga. No hay impacto en la eficiencia.

En términos prácticos, la idea es reorganizar A, mediante el cálculo implicito de la permutación 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 muchas veces se busca modificar directamente A.

Cuando se permiten elementos repetidos, se le llama *ordenamiento estable* i se asegura que en el arreglo ordenado se preserven el orden original posicional cuando $a = b$. Esta propiedad es importante cuando hay datos satélitales asociados a la llave de comparación.
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. Un ejemplo de esto es ordenar un *dataframe*, pero también estructuras de datos donde existe un campo especial y el resto de los datos asociados es de importancia para una aplicación.

5.1.1 Costo del problema

Para una entrada de tamaño n existen $n!$ permutaciones posibles; cada una de estas permutaciones es una instancia del problema de ordenamiento de tamaño n.

Existe una permutación objetivo π^*, i.e., que cumple con la definición de que esta ordenada; ahora pensemos en un grafo donde cada π_i esta conectada con todas las permutaciones en las que se puede transformar haciendo una única operación, e.g., intercambiando un elemento. El algoritmo forma ese grafo con sus posibles decisiones, por lo que el camino más largo *i.e., ruta sin ciclos*, entre cualquier π_i y la permutación π^* es el costo de peor caso del algoritmo.

Ahora, cada operación que realicemos en un algoritmo nos acercará más a π^*, descartando una cierta cantidad de instancias posibles pero no viables; si nuestra función de transición en el grafo viene dada con respecto a colocar cada par de elementos en su orden relativo, entonces, la mitad de las permutaciones se han descartado, ya que ese par no puede estar en el orden contrario. Por tanto, el costo de cualquier algoritmo que realice comparaciones y descarte la mitad del espacio de búsqueda, es $\log_2(n!)$, que usando la aproximación de Stirling,[1] lo podemos reescribir como sigue:

$$\log_2(n!) = n \log_2 n - n \log_2 e + O(\log_2 n)$$

Esto se puede simplemente escribir como $O(n \log n)$.

[1] Aproximación de Stirling https://en.wikipedia.org/wiki/Stirling%27s_approximation.

5.2 Algoritmos de ordenamiento

Existen muchos algoritmos que pueden resolver el problema de ordenamiento, es común contar el número de comparaciones ya que produce la información necesaria para la navegación en el grafo de instancias; también es común contar las operación de intercambiar elementos. Las pruebas y la navegación en el grafo determina el costo del algoritmo. Es necesario mencionar que mover datos entre diferentes zonas de memoria puede llegar a ser más costoso que solo acceder a esas zonas por lo que hay una asimetría en el costo de estas dos operaciones.

Note que algunos de los algoritmos más simples pueden tener un comportamiento oportunistas y que son capaces de obtener ventaja en instancias sencillas, por lo que no debería saltarse esas secciones si solo conoce su comportamiento en peor caso.

5.2.1 Bubble sort

El algoritmo de ordenamiento de burbuja o *bubble sort* realiza una gran cantidad de comparaciones, como puede verse en Listado 5.1, el algoritmo usa dos ciclos anidados para realizar una comparación y una posible transposición, formando un *triángulo*, i.e.,

$$\sum_{i=1}^{n-1}\sum_{j=1}^{n-i} O(1);$$

por lo tanto su costo esta dominado por el triangulo formado, i.e., $\sim n^2/2$ lo que puede escribirse simplemente como $O(n^2)$.

① Ciclo que recorre $n-1$ veces todo el arreglo; y pone el elemento máximo en su posición final.
② Ciclo que recorre $n-i$ veces el arreglo; ya que en cada corrida se pone el máximo en su posición.
③ Intercambio cuando hay pares en desorden.

```
8-element Vector{Int64}:
 1
 2
 3
 4
```

Listado 5.1 Bubble sort de peor caso

```
function bubble_sort!(A)
  n = length(A)
  for i in 1:n-1                                    ①
    for j in 1:n-i                                  ②
      if A[j] > A[j+1]
        A[j], A[j+1] = A[j+1], A[j]                 ③
      end
    end
  end

  A
end

bubble_sort!([8, 4, 3, 1, 6, 5, 2, 7])
```

```
 5
 6
 7
 8
```

El algoritmo mostrado en Listado 5.1 es un algoritmo de peor caso, ya que sin importar la complejidad de la instancia (i.e., que tal alejada esta π_i de π^*), se comporta igual.

Es relativamente fácil hacer un bubble sort que tenga en cuenta la complejidad de la instancia, medida como el número de intercambios necesarios.

① La idea es que si no hay intercambios en una iteración, entonces el arreglo ya esta ordenado.
② Contador de intercambios.
③ Condición de paro, i.e., no hubo intercambios.

```
8-element Vector{Int64}:
 1
 2
 3
 4
```

Listado 5.2 Bubble sort adaptable

```
function adaptive_bubble_sort!(A)
  n = length(A)

  for i in 1:n-1
    s = 0
    for j in 1:n-i                                    ①
      if A[j] > A[j+1]
        s += 1                                        ②
        A[j], A[j+1] = A[j+1], A[j]
      end
    end
    s == 0 && break                                   ③
  end

  A
end

adaptive_bubble_sort!([7, 8, 4, 3, 1, 6, 5, 2])
```

```
5
6
7
8
```

En la forma Listado 5.2, bubble sort es capaz de términar en $n-1$ comparaciones si el arreglo esta ordenado; sacando provecho de casos simples en términos de instancias casi ordenadas.

5.2.2 Insertion sort

El algoritmo de ordenamiento por inserción o *insertion sort* es un algoritmo simple que al igual que bubble sort tiene un mal peor caso y puede aprovechar casos simples

① El algoritmo comienza en la segunda posición del arreglo y revisará todos los elementos.

Listado 5.3 Algoritmo *insertion sort*

```
function insertion_sort!(A)
  n = length(A)
  for i in 2:n                                          ①
    key = A[i]                                          ②
    j = i - 1
    while j >= 1 && A[j] > key                          ③
      A[j + 1] = A[j]                                   ④
      j -= 1
    end

    A[j + 1] = key                                      ⑤
  end

  A
end

insertion_sort!([5, 1, 4, 8, 2, 6, 3, 7])
```

② Es importante hacer una copia de *key* para simplificar la implementación.

③ La idea general es ordenar las posiciones de $1..i$, para esto se debe recorrer hacia atrás el arreglo completo, para determinar la posición de inserción de *key*.

④ Intercambio de elementos para colocar *key* en su lugar ordenado.

⑤ *key* se pone en su lugar final.

```
8-element Vector{Int64}:
 1
 2
 3
 4
 5
 6
 7
 8
```

Para analizar Listado 5.3, es importante notar que el ciclo más externo termina con el subarreglo $A[1..i]$ ordenado; por lo que cuando se comienza el ciclo, si *key* se prueba estar en su posición correcta, entonces ya no es necesario revisar el resto del subarreglo, esto determina que un arreglo ordenado tendrá un costo de $O(n)$ comparaciones; si esta *casi ordenado* en términos del número de intercambios necesarios, entonces, el algoritmo se adaptará sacando provecho de la instancia.

En el *peor caso* de insertion sort, el algoritmo no puede parar de manera prematura, e.g., un arreglo en orden reverso, el ciclo `for` se ejecutara $n-1$ veces, mientras que el ciclo `while` deberá revisar el subarreglo completo en cada iteración, sumando un costo de i operaciones en cada iteración, i.e., $\sum_{i=1}^{n} i$, esta forma produce un *triángulo*, resultando en un costo $O(n^2)$.

5.2.3 Quick sort

Quick sort (ver Cormen et al. 2022, cap. 7) es un algoritmo tipo *dividir para vencer*; esto es, un algoritmo que divide un problema grande en instancias pequeñas más sencillas. Es uno de los algoritmos más veloces en la práctica por su buen manejo de memoria, aun cuando tiene un peor caso cuadrático, en promedio el costo es $O(n \log n)$.

① El arreglo se divide en 3 partes, ordenadas entre sí, un subarreglo izquierdo, un pivote, y un subarreglo derecho; los subarreglos no estan ordenados localmente, pero el pivote esta en su posición final.
② Se resuelve el problema izquierdo y el problema derecho por separado.
③ La función *part!* particiona el arreglo $A[low : end]$ en 3 partes como se específico en el punto 1; para eso selecciona de manera aleatoria un pivote. Lo ponemos al final del arreglo para simplificar el código siguiente.
④ Este ciclo itera por todo el subarreglo, su objetivo es asegurar que $A[i] < piv$ para todo $i \in low : piv - 1$ y $piv < A[i]$ para todo $i \in piv + 1 : high$.
⑤ Intercambia elementos si $A[j] < piv$, hacemos seguimiento de i ya que esta posición determinará al pivote.

⑥ Como *piv* se encontraba en *high*, entonces hay que intercambiarlos para que *qsort!* sepa como manejarlos; recordando que los subarreglos no estan ordenados dentro de sí.

```
8-element Vector{Int64}:
 1
 2
 3
 4
 5
 6
 7
 8
```

El código Listado 5.4 es relativamente simple, usa recurrencias sobre *qsort!* sobre dos partes extremas divididas por un pivote; estos tres elementos son encontrados en *part!*. La función *part!* es muy eficiente en términos de memoria, lo que puede hacer la diferencia en la práctica. La correcta selección del pivote es muy importante para evitar casos malos, i.e., costo cuadrático; en esta implementación se realiza una selección aleatoría de pivote que funcionará en la mayoría de los casos.

El peor de los casos en *qsort!* es debido a una mala selección del pivote, de tal forma que

$$|A[low : piv - 1]| \ll |A[piv + 1 : high]|,$$

o lo contrario en toda selección, en el extremo una de los subarreglos puede verse como de tamaño constante o cero, i.e., selección de pivote como el *minimo* o el *máximo*. Esta estrategía reduce a *qsort!* a un costo $O(n^2)$.

Si se realiza un particionado donde

$$|A[low : piv - 1]| \approx |A[piv + 1 : high]|,$$

entonces tenemos un algoritmo $O(n \log n)$; ya que hace una división en dos partes casi iguales en cada recurrencia a *qsort!*, y esto solo puede profundizar a $\log n$ veces, y en cada nivel *part!* tiene un costo lineal.

5.3 Skip list

Una *skip list* (Pugh 1990) es una lista ligada con capacidades de búsqueda eficiente con garantías probabilísticas, esto es que se cumplen con alta probabilidad. Para esto, la idea es que cada dato tiene asociado un arreglo de punteros o referencias hacia nodos sucesores, i.e., los nodos a nivel i se conectan con el siguiente nodo a nivel i. En el nivel más bajo, la *skip list* es una simple lista ligada, mientras que sube, se vuelve más dispersa dando saltos más largos.

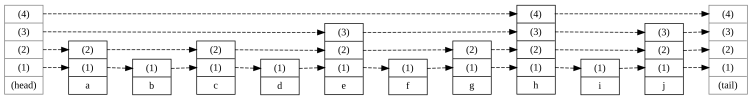


Figura 5.1: Ejemplo de una skip list

A diferencia de los algoritmos vistos anteriormente, en este caso, ya se tiene una estructura de datos, que conlleva un costo en memoría explícito por nodo. Figura 5.1 ilustra la estructura.

La altura de cada nodo es calculada de manera probabilística, dada la probabilidad p. Un valor común de $p = 0.5$. La altura de cada nodo se calcula como sigue:

```
function levels(p)
  i = 1
  while rand() < p
    i += 1
  end

  i
end
```

```
levels (generic function with 1 method)
```

Si tenemos n evaluaciones de *levels*, Los niveles pequeños son relativamente probables, mientras que niveles grandes son relativamente poco probables. De hecho, los niveles $\log_{1/p} n$

son cercanos a una constante, $\log_{1/p} n - 1$ son $1/p$ veces la constante, $\log_{1/p} n - 2$ son $1/p^2$ veces la constante, etc.

A diferencia de los algoritmos anteriores, una *skip list* comienza vacia, y se va poblando insertando elementos a la lista. Se va colocando en la posición que no viola el orden; generando el nodo correspondiente con nivel calculado. Los nodos especiales *head* y *tail* siempre tienen el nivel máximo posible. La inserción de un valor encapsulado en el nodo u comienza por visitar el máximo nivel en *head* e ir bajando hasta determinar $u.dato > head[level].dato$; en ese momento se debe avanzar al nodo apuntado por $head[level]$ y repetir el algoritmo hasta que $level = 1$, en cuyo caso encontramos el lugar de inserción del nuevo dato. Se procede a reasignar los punteros de los sucesores y ajustar los punteros hacia los nodos sucesores a los niveles que tiene u.

Cada inserción tiene un costo $O(\log_{1/p} n)$, garantía probabilística; por lo que insertar n elementos tiene un costo:

$$\sum_{i=1}^{n} O(\log_{1/p} i) = O(\log_{1/p} \prod_{i=1}^{n} i) = O(\log_{1/p} n!) = O(n \log n);$$

usando la aproximación de Stirling.

A diferencia de la versión basada en arreglos, una *skip list* es capaz de aceptar nuevos elementos y mantener el orden de manera eficiente.

5.3.1 Ejercicios:

1. Investigue, implemente y pruebe *merge sort*. 1.1 ¿Cuales son las ventajas y desventajas de *merge sort*? 1.2 ¿Por qué *merge sort* se puede utilizar en algoritmos paralelos y otros pueden tener muchas dificultades? 1.3 ¿Cómo se puede reducir la memoria extra necesaria de *merge sort*?
2. Investigue, implemente y pruebe *heap sort*. 2.1 ¿Cuales son las ventajas y desventajas de *heap sort*?
3. ¿Cuál es el costo en memoria de una *skip list*?. 3.1 Investigue, implemente y pruebe un *skip list*.
4. Investigue, implemente y pruebe un árbol binario de búsqueda.

5.4 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.

5.5 Material audio-visual sobre algoritmos de ordenamiento

5.6 Actividades

1. Implementa y compara los siguientes algoritmos de ordenamiento:

- heapsort

- mergesort
- quicksort
- bubblesort

2. Utiliza los diferentes archivos proporcionados, los cuales tienen diferentes niveles de desorden y mide tanto el número de comparaciones como el tiempo necesario para ordenarlos.

3. Por cada archivo de datos, compara todos los métodos implementados mediante figuras o tablas de datos (número de comparaciones y tiempo por separado).

4. Discute tus resultados.

Entregable:

1. Tu 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.

2. No olvides estructurar tu reporte, en particular, 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.

Listado 5.4 Algoritmo *quick sort*.

```
using Random

function qsort!(A, low=1, high=length(A))
  if low < high
      piv = part!(A, low, high)                    ①
      qsort!(A, low, piv - 1)                      ②
      qsort!(A, piv + 1, high)
  end

  A
end

function part!(A, low, high)
  ipiv = rand(low:high)                            ③
  A[ipiv], A[high] = A[high], A[ipiv]
  piv = A[high]

  i = low - 1  # uno antes porque se accede después de un i+1
  for j in low:high - 1                            ④
      if A[j] < piv
          i += 1
          A[i], A[j] = A[j], A[i]                  ⑤
      end
  end

  ipiv = i + 1
  A[ipiv], A[high] = A[high], A[ipiv]              ⑥
  ipiv
end

qsort!([6, 8, 3, 7, 4, 1, 2, 5])
```

6 Algoritmos de búsqueda en el modelo de comparación

```
Plots.GRBackend()
```

Objetivo

Analizar algoritmos de búsqueda en arreglos ordenados basados en funciones de comparación, con el objetivo de localizar elementos y posiciones específicas, usando técnicas de peor caso y adaptables a la distribución de los datos para una solución eficiente de problemas informáticos.

6.1 Problema

Sea $A[1..n] = a_1, \cdots, a_n$ un arreglo ordenado con $n \geq 1$ y un operador $<$ (menor que); por simplicidad, también usaremos $\leq$ (menor o igual que). Supondremos que no hay elementos duplicados en A, note que esto no implica una perdida de generalidad.

La tarea será: dado el valor x a ser localizado en A, el problema consiste en determinar la posición de inserción p tal que suceda alguna de las siguientes condiciones:

- si $p = 1$ entonces $x \leq A[p]$.
- si $2 \leq p \leq n$ entonces $A[p-1] < x \leq A[p]$.
- si $p = n + 1$ entonces $A[n] < x$.

6.1.1 Costo de peor caso

Para $A[1..n]$ y el valor x a localizar su posición de inserción, el resultado puede ser cualquiera de las $n+1$ posiciones posibles, i.e., instancias del problema. Un algoritmo naïve utilizaría n comparaciones para resolverlo.

```
"""
    seqsearch(A, x, sp=1)

Búsqueda exhaustiva con inicio
"""
function seqsearch(A, x, sp=1)
    n = length(A)
    while sp <= n && x > A[sp]
        sp += 1
    end

    sp
end

let S=[10, 20, 30, 40, 50, 60, 70]
    (seqsearch(S, 0), seqsearch(S, 69), seqsearch(S, 70), seqsearch(S, 71))
end
```

```
(1, 7, 7, 8)
```

Sin embargo, dado que el arreglo esta ordenado y no hay duplicados, se puede mejorar mucho el tiempo de búsqueda.

El costo de búsqueda para cualquier instancia es $O(\log n)$, y viene de la búsqueda binaria:

Si se permiten duplicados se pueden mejorar muchos los tiempos; sobre todo si podemos preprocesar el arreglo, i.e., para determinar las zonas con duplicados.

```
"""
    binarysearch(A, x, sp=1, ep=length(A))

Encuentra la posición de inserción de `x` en `A` en el rango `sp:ep`
"""
function binarysearch(A, x, sp=1, ep=length(A))
    while sp < ep                                   ③
        mid = div(sp + ep, 2)                       ①
```

```
        if x <= A[mid]
            ep = mid                                    ②
        else
            sp = mid + 1
        end
    end

    x <= A[sp] ? sp : sp + 1                            ④
end

let S=[10, 20, 30, 40, 50, 60, 70]
    (binarysearch(S, 0), binarysearch(S, 69), binarysearch(S, 70), binarysearch(S, 71))
end
```

① Para el rango de búsqueda $sp : ep$ se determina su punto central mid y se compara con x,
② Si el elemento x esta a la izquierda, se ajusta el limite superior ep, o de lo contrario se ajusta sp. Ambos ajustes se hacen tomando en cuenta la posición comparada.
③ Se itera mientras no se junten los dos extremos del rango.
④ Finalmente, se ajusta para valores fuera del rango.

```
(1, 7, 7, 8)
```

Este algoritmo es simple y efectivo, y es capaz de resolver cualquier instancia en tiempo logarítmico, y esto lo hace al dividir el rango siempre a la mitad por cada iteración. El costo de búsqueda binaria es de $C_{\text{bin}}(n) = \lfloor \log n \rfloor + O(1)$ comparaciones antes de colapsar el rango donde puede estar la posición de inserción.

Es importante hacer notar que la búsqueda binaria es muy eficiente en memoría y tiene un peor caso óptimo, ya que es idéntico al costo del problema, i.e., así lo determinamos. Si fuera posible tener probar varios puntos, i.e., m segmentos en una sola operación, el costo estaría acotado en $\lceil \log_m n \rceil$. Esto tiene sentido para estructuras de datos que trabajan en diferentes niveles de memoría, donde aunque las comparaciones en hardware moderno sean binarias, la diferencia entre velocidades de los diferentes niveles de memoria se puede pensar que el costo dominante es, por ejemplo, acceder a una zona de disco y

obtener una decisión entre $m-1$ posibles, que particionan los rangos en m divisiones.

6.2 Búsqueda *no* acotada

Cuando el tamaño del arreglo es demasiado grande, o la relación entre p/n es significativamente pequeña, la búsqueda acotada no es la mejor opción. Aun cuando en la práctica el límite superior n podría estar determinado, y por lo tanto, se pueden resolver búsquedas en $O(\log n)$, es posible obtener una cota relativa a p, independiente de n, por lo que los casos de interés se verán beneficiados.

Una estrategia simple y poderosa es la siguiente:

1. Determinar un *buen* rango que contenga la respuesta.
2. Aplicar búsqueda binaria en ese rango para obtener la respuesta.

Bentley y Yao (1976) describen a detalle una familia de algoritmos casí óptimos para la búsqueda no acotada siguiendo la estrategía anteriormente mencionada. En particular, poniendo un enfásis importante en la determinación del rango. Lo consigue mediante la definición de algoritmos definidos de manera interesante como sigue en el resto de la sección.

6.2.1 Algoritmo B_0 (búsqueda unaría)

Es el algoritmo más simple, y ya lo vimos con anterioridad, realiza una búsqueda exhaustiva de la posición de inserción, hacendo pruebas para toda posición $x \leq A[1], x \leq A[2], \cdots, x \leq A[p+1]$, por lo que su costo será de $p+1$.

Sea $F_0(n)$ una secuencia de puntos para un arreglo de longitud n, donde se harán comparaciones para determinar el rango que contenga la respuesta para el algoritmo B_0 y $C_0(p)$ el costo de búsqueda. Entonces:

- $F_0(n) = 1, 2, \cdots, n, n+1$.
- $C_0(p) = p+1$; no requiere búsqueda binaria.

6.2.2 Algoritmo B_1 (búsqueda doblada: *doubling search/galloping*)

Consiste en comparar las posiciones 2^i, i.e., $2^1, 2^2, 2^3, \cdots, 2^{\lfloor \log_2 p+1 \rfloor+1}$, tal que $A[2^{\lfloor \log_2 p \rfloor+1}] \leq x \leq A[2^{\lfloor \log_2 p+1 \rfloor+1}]$. De manera similar que para B_0 definimos $F_1(n)$ y C_1:

- $F_1(n) = 2^1, 2^2, \cdots, 2^{\log \lfloor n \rfloor+1}$;
- $C_1(p) = C_{\text{bin}}(2^{\log_2 p+1}) + \log_2 (p+1) + 1 < 2 \log_2 p + O(1)$.

La explicación viene a continuación. El número de comparaciones para determinar el rango esta determinado por $\lfloor \log_2 p + 1 \rfloor + 1$. Una vez determinado el rango la búsqueda binaria sobre

$$A[2^{\lfloor \log_2 p \rfloor+1} : 2^{\lfloor \log_2 (p+1) \rfloor+1}],$$

lo cual corresponde a $\log_2 2^{\log (p+1)+1}/2 = \log_2 (p+1)$. El costo $C_1(p)$ puede ser escrito como $2 \log_2 p + O(1)$, con un poco de manipulación algebraica.

Es importante saber cuando usar un algoritmo u otro, por tanto determinar cuando $2 \log_2 p + O(1) < \log_2 n + O(1)$. Para simplificar este análisis ignoraremos algunos detalles de la expresión:

$$2 \log_2 p < \log_2 n, \tag{6.1}$$
$$2^{\log_2 p^2} < 2^{\log_2 n}, \tag{6.2}$$
$$p^2 < n, \tag{6.3}$$
$$p < \sqrt{n}; \tag{6.4}$$
$$\tag{6.5}$$

esto indica que si p es menor a $\sqrt{n}$ entonces hay una ventaja al usar B_1; lo cual nos dice que para posiciones cercanas al inicio el uso de B_1 puede llevar a búsquedas más veloces. Note que en la práctica es necesario tener en cuenta la memoria, interesantemente, para p pequeñas es posible que esto beneficie al algoritmo ya que podría mantener las listas en cache.

El siguiente código implementa B_1

```
function doublingsearch(A, x, sp=1)
    n = length(A)
    p = 0
```

```
    i = 1

    while sp+i <= n && A[sp+i] < x                ①
        p = i
        i += i
    end

    binarysearch(A, x, sp + p, min(n, sp+i))      ②
end

let S=[10, 20, 30, 40, 50, 60, 70]
    (doublingsearch(S, 0), doublingsearch(S, 69), doublingsearch(S, 70), doublingsearch(S, 71))
end
```

① Determinación del rango.
② Aplicar un algoritmo de búsqueda eficiente en el rango que contiene la respuesta.

```
(1, 7, 7, 8)
```

Es cierto que estos algoritmos son oportunistas, pero hay aplicaciones donde esto realmente sucede. En el peor caso, el costo será apenas dos veces el óptimo.

6.2.3 Algoritmo B_2 (búsqueda doblemente doblada, *doubling-doubling search*)

Aquí será más clara la dinámica. B_2 consiste en comparar las posiciones 2^{2^i}, i.e., $2^4, 2^{16}, 2^{256}, \cdots, 2^{2^{\lfloor \log_2 \lfloor \log_2 p+1 \rfloor +1 \rfloor +1}}$, tal que

$$A[2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor +1 \rfloor +1}}] \leq x \leq A[2^{2^{\lfloor \log_2 \lfloor \log_2 p+1 \rfloor +1 \rfloor +1}}];$$

La determinación de este rango requiere $\lfloor \log_2 \lfloor \log_2 p + 1 \rfloor + 1 \rfloor + 1$ comparaciones; sin embargo, este rango seguramente será muy grande, por el tamaño de los saltos que se estan dando entre puntos de comparación, por lo que no conviene usar busqueda binaria y podemos aplicar B_1 para resolver en ese rango acotado.

$$F_2(n) = 2^{2^1}, 2^{2^2}, \cdots, 2^{2^{\lfloor \log_2 \lfloor \log_2 n \rfloor + 1 \rfloor + 1}}; \tag{6.6}$$
$$C_2(p) = \lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor + 1 + C_1(p') \tag{6.7}$$
$$< \log_2 p + 2 \log_2 \log_2 p + O(1); \tag{6.8}$$
$$\tag{6.9}$$

donde $p' = p - 2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor}}$, es decir, la posición de inserción en el rango ya acotado.

Note como el término de mayor peso es muy similar a B_1 pero destaca la inclusión del término log log que permite adaptarse a p muy grandes con un pequeño costo adicional, que en términos prácticos se puede ver como una constante.

La idea principal es como sigue: una vez determinado el rango, en lugar de usar búsqueda binaria y tener un costo $\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor + 1 + C_{\text{bin}}(2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor + 1}} - 2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor}})$ es preferible usar B_1 y conseguir un algoritmo que se adapte a la entrada. De manera más precisa, tomar ventaja de

$$C_1(2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor + 1}} - 2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor}}) < C_{\text{bin}}(2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor + 1}} - 2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor}})$$

cuando

$$p' < \sqrt{2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor + 1}} - 2^{2^{\lfloor \log_2 \lfloor \log_2 p \rfloor + 1 \rfloor}}}$$

.

Simplificando las expresiones, la relación que nos describe cuando es mejor usar B_2 que la búsqueda binaria es como sigue:

$$\log_2 p + 2 \log_2 \log_2 p < \log_2 n \tag{6.10}$$
$$2^{\log_2 p + \log_2 \log_2^2 p} < 2^{\log_2 n} \tag{6.11}$$
$$2^{\log_2 (p \log_2^2 p)} < 2^{\log_2 n} \tag{6.12}$$
$$2p \log_2 p < n \tag{6.13}$$
$$p \log_2 p^2 < n \tag{6.14}$$
$$\tag{6.15}$$

Si $p = \sqrt{n}$ entonces $\sqrt{n} \log_2 n$ claramente es menor que n incluso para valores relativamente pequeños de n, por lo que B_2 funciona mejor para p relativamente grandes en comparación con B_1.

6.2.4 Algoritmo B_k

Bentley y Yao (1976) generalizan la estrategía para cualquier k. De manera simplificada:

- $F_k(n) = 2^{\cdot^{\cdot^{2^i}}}$ (exponenciando k veces) para i desde 1 a $\log_2^{(k)} n$;
- $C_k(p) = \log_2^{(k)}(p) + C_{k-1}(2^{\cdot^{\cdot^{2^{\log_2^{(k)}(p)}}}} - 2^{\cdot^{\cdot^{2^{\log_2^{(k)}(p)-1}}}})$;

donde $\log_2^{(k)}(n) = \log_2(\lfloor \log_2^{(k-1)}(n) \rfloor + 1)$, con el caso base de $\log_2^{(1)} n = \lfloor \log_2 n \rfloor + 1$.

La estrategia lleva a que el valor casi óptimo para la búsqueda por comparación se da cuando $k = \log_2^\star n$ donde $\log_2^\star$ es el logaritmo iterado, que esta definido como las veces que se debe iterar aplicando el logaritmo para obtener un valor de 1 o menor que 1, i.e., la k más pequeña tal que $\log_2^{(k)} n \leq 1$.

6.3 Ejercicios

- Implementar y probar B_2.
- Derivar el costo $C_2(p)$.
- ¿Cuando B_1 es mejor que B_2?
- Haga un pseudo-código para B_k.
- ¿Cuál es el costo C_k?
- ¿Qué es un árbol binario de búsqueda?
- ¿Cuál es el costo de búsqueda en un árbol? ¿qué se debe hacer para asegurar los costos?
- ¿Qué es un finger tree?
- ¿Cuál es el costo de búsqueda de la *skip list*?
- ¿Cómo se puede hacer la *skip list* adaptativa? ¿qué otra forma podría aplicar?

6.4 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.

7 Algoritmos de intersección y unión de conjuntos en el modelo de comparación

Objetivo

Analizar el rendimiento de algoritmos de unión e intersección de conjuntos representados como listas ordenadas parametrizando los algoritmos con los algoritmos internos de búsqueda, tamaño de los conjuntos y la distribución de los elementos, bajo un enfoque experimental midiendo los costos en términos del tiempo de ejecución y el uso de memoria.

7.1 Problema

Cómo se vió en Capítulos anteriores, un conjunto es una colección de elementos donde no hay repetición. El uso de conjuntos es fundamental para un gran número de problemas. En particular, en este capítulo representaremos conjuntos como arreglos ordenados de números enteros; esto para posicionarlo dentro de un dominio de aplicación objetivo, que es la Recuperación de Información, como parte de la representación de una la matriz dispersa muy grande, llamada *índice invertido*.

Estaremos resolviendo los problemas de intersección y unión de conjuntos. Demaine, López-Ortiz, y Munro (2000) demuestra que el costo y procedimiento de las intersecciones y uniones de conjuntos representados como arreglos ordenados, es básicamente el mismo; ya que requieren determinar la misma información. Claramente, colectar los datos para la unión y la intersección, requieren diferentes esfuerzos.

7.1.1 Costo del problema

En general, dados dos conjuntos $A[1..m] = \{a_1 < a_2 < \cdots < a_m\}$ y $B[1..n] = \{b_1 < b_2 < \cdots < b_n\}$, el costo de unión es

$$\log \binom{m+n}{m},$$

ver Hwang y Lin (1971).

De manera más detallada, supongamos que $A \cap B = \emptyset$, esto es, el conjunto de salida será de tamaño $m + n$. De manera similar al razonamiento que se utilizó para el problema de ordenamiento, el problema puede verse como todas las posibles instancias de ordenes o permutaciones de tamaño $n + m$; removiendo la necesidad de los ordenes parciales, esto es $\binom{n+m}{m}$ posibles instancias de tamaño $n + m$, generadas por dos conjuntos de tamaño n y m. Dado que estamos en un modelo basado en comparaciones, y dado el mejor algoritmo s puede dividir el espacio de posibles ordenes en 2, por tanto, dicho algoritmo necesitará

$$\log_2 \binom{n+m}{m}$$

comparaciones para resolver la unión de cualquier par de conjuntos de tamaño m y n.

Usando la aproximación de Stirling para coefficientes binomiales de MacKay (2003), el costo se convierte en:

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

Recuerde que $\log_2 x = \frac{\log_e x}{\log e}$.

7.2 Algoritmos

Se puede observar que si $m \approx n$, entonces el costo se convierte en $O(m + n)$, esto es, lo más eficiente sería tomar el siguiente algoritmo:

```
function merge2!(C, A, B)                          ①
    i = j = 1
    m, n = length(A), length(B)

    @inbounds while i <= m ||  j <= n              ②
        a, b = A[i], B[j]
        if a == b
            push!(C, a)
            i += 1
            j += 1
        elseif a < b
            push!(C, a)
            i += 1
        else
            push!(C, b)
            j += 1
        end
    end

    C
end
```

```
merge2! (generic function with 1 method)
```

7.2.1 Ejercicio

- Escriba y pruebe el algoritmo de intersección de merge para $n \approx m$.

7.3 Algoritmos para arreglos de tamaño muy diferente

Si $m \ll n$, el costo tenderá a $O(m \log n)$, por lo que se pueden realizar m búsquedas binarias *directas* para localizar la posición de inserción en B.

Se hace notar que A y B estan ordenados, y por lo tanto, localizar $A[i]$ en $B[j]$ significa que $B[j-1] < A[i]$, por lo que intentar

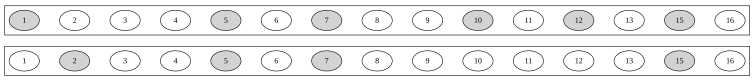


Figura 7.1: Dos listas alineadas donde los nodos sombreados son elementos de los conjuntos.

localizar $A[i+1]$ puede comenzar en $B[j+1]$. A continuación se muestra el código de un algoritmo de intersección usando algoritmos de búsqueda con memoria de la posición anterior.

```
function intsearch!(C, A, B, algosearch=doublingsearch)
    if length(B) < length(A)
        A, B = B, A
    end

    p = 1
    for (i, a) in enumerate(A)
        p = algosearch(B, a, p)
        p > length(B) && break
        if a == B[p]
            p += 1
            push!(C, a)
        end
    end

    C
end
```

Hwang y Lin (1971) propone otro algoritmo que funciona para casos similares:

1. Divide B en bloques de tamaño m, define un arreglo *virtual* $B'[1..n/m]$ donde $B'[i] = B[i \cdot m]$
2. Se búsca la posición de inserción p de cada $a \in A$ en B', costando $\log n/m$ para cada búsqueda.
3. Después se localiza dentro del B en el p-ésimo bloque, i.e., $B[(p-1)m + 1..p \cdot m]$, por la posición de inserción del bloque, con un costo de $\log m$.

Entonces, se obtiene un costo de $O(m \log n/m + m \log m)$; esto es equivalente en el peor caso a búsquedas directas, i.e., las

posiciones de inserción de $a \in A$ se encuentran distribuidas de manera uniforme en B. Sin embargo, es posible mejorar si se descartan bloques en el paso 1. Esto es, si se hay concentración de elementos de A en bloques de B. Para esto, es necesario un análisis de costo promedio, el cual se muestra en el artículo.

Incluso cuando hay concentración, podemos recordar la $(i-1)$ posición de inserción para iniciar la i-ésima búsqueda, y sacar provecho de posiciones esperadas cercanas de la posición inicial de búsqueda, i.e., podemos utilizar algoritmos de búsqueda adaptables para mejorar el desempeño.

7.3.1 Algoritmo de Baeza Yates

Baeza-Yates (2004) propone un algoritmo eficiente para intersecciones de dos conjuntos. El algoritmo tiene una estrategía *dividir para vencer*:

1. Se toma la mediana M de A y se busca en B obteniendo su posición de inserción p.
2. El problema entonces se divide en 3 subproblemas:

$$C_< = \{A[1..M-1] \cap B[1..p-e]\} \tag{7.1}$$
$$C_= = \{A[M]\} \cap \{B[p]\} \tag{7.2}$$
$$C_> = \{A[M+1..m] \cap B[p+e..n]\} \tag{7.3}$$
$$\tag{7.4}$$

   donde $e = 1$ si $A[M] = B[p]$ y $e = 0$ cuando $A[M] \neq B[p]$.
3. La unión de estos tres conjuntos es la solución $C_< \cup C_= \cup C_>$.
4. El problema $C_=$ es trivial, y $C_<$ y $C_>$ se implementan recurriendo, ajustando los rangos de trabajo.

A continuación se muestra el código en Julia, usando los algoritmos de búsqueda del Cap. 5.

```
# Adaptado de https://github.com/sadit/Intersections.jl

function baezayates!(output, A, B, findpos::Function=binarysearch) ①
    baezayates!(output, A, 1, length(A), B, 1, length(B), findpos)
end
```

```
function baezayates!(output, A, a_sp::Int, a_ep::Int, B, b_sp::Int, b_ep::Int, findpos::Functi
    (a_ep < a_sp || b_ep < b_sp) && return output
    imedian = ceil(Int, (a_ep + a_sp) / 2)
    median = A[imedian]
    ## our findpos returns n + 1 when median is larger than B[end]
    medpos = min(findpos(B, median, b_sp), b_ep)  ②

    matches = median == B[medpos]
    baezayates!(output, A, a_sp, imedian - 1, B, b_sp, medpos - matches, findpos) ③
    matches && push!(output, median)              ④
    baezayates!(output, A, imedian + 1, a_ep, B, medpos + matches, b_ep, findpos) ⑤
    output
end
```

① Punto de entrada.
② Búsqueda de la posición de inserción de la mediana de A en B.
③ Recurrencia para el problema $C_<$.
④ Añadir al resultado el valor de la mediana si es que se encontró en B; es importante que este paso este entre las recurrencias para que *output* sea un arreglo ordenado.
⑤ Recurencia para el problema $C_>$.

El algoritmo de Baeza Yates es óptimo en el peor caso y es capaz de aprovechar casos donde $C_<$ o $C_>$ se convierten en triviales, lo cual da muy buenos casos en algunas distribuciones.

7.3.2 Ejercicios

1. Implemente la unión con el algoritmo de Baeza Yates.

7.4 Operaciones con tres o más conjuntos

Los algoritmos y costos hasta ahora revisados se cumplen para dos conjuntos; se mencionaron diferentes algoritmos, algunos de ellos especializados por características como las proporciones de los conjuntos de entrada.

En particular, es importante hacer notar que ni el problema ni las aplicaciones estan limitadas a dos conjuntos, y por tanto, es importante algoritmos y estrategías para resolver $\bigcup_i A_i$ así como $\bigcap_i A_i$.

7.4.1 Algoritmo SvS

Dado $C = A \cap B$ es un hecho que $|C| \leq min\{|A|, |B|\}$. Recordando, que hay maneras relativamente simples y eficientes de resolver la intersección cuando $m \ll n$; por tanto, cuando tenemos más de dos conjuntos podemos aplicar la estrategía *Small vs Small (SvS)*, que consisten en intersectar los k conjuntos por pares intersectando el par de arreglos más pequeños cada vez.

```
# Adaptado de https://github.com/sadit/Intersections.jl

function svs(L::Vector{T}, in2::Function=baezayates!) where T
    prev, curr = eltype(T)[], eltype(T)[]
    sort!(L, by=length, rev=true)
    curr = pop!(L)

    while length(L) > 0
        empty!(prev)
        isize = in2(prev, curr, pop!(L))
        isize == 0 && return prev
        prev, curr = curr, prev
    end

    curr
end
```

7.4.2 Algoritmo de Barbay y Kenyon

Existe otra familia de algoritmos, basados en búsquedas adaptativas que pueden llegar a mejorar el desempeño bajo cierto tipo de entradas. Demaine, López-Ortiz, y Ian Munro (2001), Barbay, López-Ortiz, y Lu (2006), y Barbay et al. (2010) muestran algoritmos de intersección basados en búsqueda adaptables para aprovechar instancias simples. Estos estudios

se basan en contribuciones teóricas de los mismos autores: Demaine, López-Ortiz, y Munro (2000), Demaine, López-Ortiz, y Ian Munro (2001), Barbay y Kenyon (2002) y Baeza-Yates (2004).

El algoritmo de Barbay, López-Ortiz, y Lu (2006) trabaja sobre los k conjuntos de entrada, representados como arreglos ordenados de números enteros. Es un algoritmo simple pero poderoso: hace uso de búsquedas adaptivas con memoria para guardar las posiciones donde se avanza, de tal forma que no se recalculen posiciones. Las diferentes estrategias para revisar los conjuntos pueden dar diferentes desempeños, como se valida en Barbay et al. (2010), donde además de hacer una gran variedad de experimentos sobre diferentes algoritmos de búsqueda, se introducen variantes en el orden de acceso de cada conjunto.

A continuación se muestra el código del algoritmo base:

```
# Adaptado de https://github.com/sadit/Intersections.jl

function bk!(output, L::AbstractVector, findpos::Function=doublingsearch)
    P = ones(Int, length(L))
    bk!(output, L, P, findpos)
end

function bk!(output, L, P, findpos::Function=doublingsearch) ①
    n = length(L)                                          ②
    el = L[1][1]                                           ③
    c = 0                                                  ④

    @inbounds while true
        for i in eachindex(P)
            P[i] = findpos(L[i], el, P[i])                 ⑤
            P[i] > length(L[i]) && return output
            pval = L[i][P[i]]
            if pval == el
                c += 1
                if c == n                                  ⑥
                    push!(output, el)
                    c = 0                                  ⑦
                    P[i] += 1
                    P[i] > length(L[i]) && return output
```

```
                el = L[i][P[i]]
            end
        else
            c = 0
            el = pval
        end
    end
end

output
end
```

① El algoritmo de Barbay & Kenyon recibe: i) `output` el conjunto de salida. ii) `L` la lista de conjuntos (representados como arreglos ordenados). iii) `P` arreglo de posiciones *actuales* para cada arreglo. iv) findpos‘ función de búsqueda.
② Número de conjuntos en `L`.
③ `el` es el elemento siendo búscado en todos los arreglos.
④ `c` número de listas que contienen `el`.
⑤ Búscando la posición de inserción de `el` en `L[i]`, comenzando en `P[i]`.
⑥ Esta igualdad implica que hay interección.
⑦ Reiniciando `el` y `c` y actualizando `P[i]`.

De manera particular, Barbay et al. (2010) presentan un estudio experimental sobre los algoritmos presentados en el área durante la decada de 2000 a 2010, dichos algoritmos se parametrizaron de maneras que nos permiten aprender diferentes características de cada uno de ellos, dependiendo de los algoritmos de búsqueda que usan, la arquitectura computacional donde se evalúa, y el número de conjuntos siendo procesados.

7.5 Recursos audio-visuales de la unidad

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

Parte 2: Algoritmos de intersección y algunas aplicaciones

7.6 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.

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.

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 Louis, y Andrew Chi-Chih Yao. 1976. «An almost optimal algorithm for unbounded searching». *Information processing letters* 5 (SLAC-PUB-1679).

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.

MacKay, David JC. 2003. *Information theory, inference and learning algorithms*. Cambridge university press.

Pugh, William. 1990. «Skip lists: a probabilistic alternative to balanced trees». *Commun. ACM* 33 (6): 668-76. https://doi.org/10.1145/78973.78977.

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.

Galería de actividades MCDI

Este curso se impartió en este formato en MCDI-2025-1; el plan es modificar las actividades ligeramente con forme pasan los semestres y que la galería apoye a los nuevos estudiantes para tener ejemplos claros de lo que se solicita.

Semestre 2025-2

Alumno	Sitio
Dalia Isabel Mendiola Alavéz	https://isabelmend.github.io/analisis-de-algoritmos/
Edgar Santiago Gonzalez Martinez	https://edgarsanti.github.io/edgarsantiago/
Emmanuel Ruiz Guarneros	https://ruge1507.github.io/Analisis-de-Algoritmos/

Semestre 2025-1

Alumno	Sitio
Alma Diana Herrera Ortiz	https://aldihero.github.io/datascience_foundations/
Arif Narváez de la O	https://arifnvz.github.io/about.html
Brigitte Rarinka Godinez Montoya	https://github.com/Programadari/analisis-de-algoritmos-2025-1/
David Segundo Garcia	https://davidsg24.github.io/An-lisis-de-algoritmos/

Alumno	Sitio
Isaac Hernández Ramírez	https://isaachr141522.github.io/reportes_algoritmos/
José Alberto Villegas Díaz Disciplina	https://albertovillegas07.github.io/AnalisisAlgoritmos2025/
José Francisco Cázarez Marroquín	https://dsfrankcaza.github.io/Analsis-de-Algoritmos/
Juan Antonio Velasquez Martinez	https://juan21javm.github.io/proyecto-analisis-de-algoritmos/
Luis Alberto Rodríguez Catana	https://albertocat.github.io/Algoritmos/
Santiago Botero Sierra	https://sboteros.com/about.html