Análisis de Algoritmos

Eric S. Téllez

Tabla de contenidos

Prefacio

El *Análisis de algoritmos* es una materia formativa diseñada para comprender 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. Es común que la solución adecuada sobre la solución ingenua permita optimizar de manera significativa los recursos computacionales, y que en última instancia, se pueden traducir como la reducción de costos de operación en un sistema o la posibilidad de procesar grandes cantidades de información de manera más eficiente.

En el ciclo de proyectos de análisis de datos, la construcción e implementación de algoritmos constituye la base de la programación para prueba de hipótesis y el modelado de problemas de análisis 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, en particular, determinar los recursos computacionales en dicho modelo para resolverlo de manera eficiente y escalable cuando sea posible.

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. De manera más detallada, se estudiarán estructuras de datos lineales, así como algoritmos de ordenamiento por

comparación, búsqueda por comparación en arreglos ordenados, así como operaciones eficientes de conjuntos representados como arreglos ordenados.

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.

Aprender a programar no es el objetivo de este curso, por lo que el alumno deberá fortalecer sus capacidades con auto-estudio si fuera necesario.

Sobre el lenguaje de programación

En principio casi cualquier lenguaje de programación podría utilizarse para el curso, sin embargo, para efectos prácticos, nos limitaremos a dos lenguajes de programación:

- Python, se recomieda utilizar la distribución de https://www.anaconda.com/download/
- Julia, se recomienda utilizar la versión 1.10 o superior, https://julialang.org/

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. En particular, cuando se requiera evaluar la velocidad de un algoritmo, se recomienda utilizar Julia, ya que suele ser mucho más veloz para rutinas creadas directamente en el lenguaje, sin necesidad de un segundo lenguaje para operaciones a bajo nivel.

Se hará uso intensivo de Quarto y Jupyter https://jupyter.org/ para las notas y demostraciones. Los reportes y tareas se solicitaran en estos frameworks.

Recursos para aprender Python y Julia

Python

- Documentación oficial, comenzar por el tutorial https://docs.python.org/3/
- Documentación oficial https://docs.julialang.org/en/stable/

Julia

- Información sobre como instalar Julia y flujos de trabajo simples (e.g., REPL, editores, etc.) para trabajar con este lenguaje de programación: *Modern Julia Workflows* https://modernjuliaworkflows.github.io/.
- Libro sobre julia *Think Julia: How to Think Like a Computer Scientist* https://benlauwens.github.io/ThinkJulia.jl/latest/book.html.
- Curso *Introduction to computational thinking* https://computationalthinking.mit.edu/Fall20/

Licencia

Esta obra está bajo una Licencia Creative Commons Atribución-CompartirIgual 4.0 Internacional

1 Introducción al análisis de algoritmos

Objetivo

Obtener los criterios básicos para el análisis, diseño e implementación de algoritmos.

1.1 Concepto de algoritmo y estructura de datos 1.2 Notación asintótica 1.3 Costo en tiempo y espacio

1.1 Introducción

La presente unidad esta dedicada a los fundamentos de análisis de algoritmos. En particular se intentará que el concepto de modelo de cómputo se adopte, se conozca y maneje la notación asintótica. Es de vital importancia que se entienda su utilidad y el porque es importante para el análisis de algoritmos. También se mostrarán algunos de los ordenes de crecimiento más representativos, que nos permitirán comparar répidamente algoritmos que resuelvan una tarea dada, así como darnos una idea de los recursos de computo necesarios para ejecutarlos. Finalmente, como parte de esta unidad, se dará un repaso a las estructuras de datos y a los algoritmos asociados, que dado nuestro contexto, son fundamentales y deberán ser comprendidos a cabalidad.

1.2 Concepto de algoritmo y estructura de datos

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

Cada paso u operación en un algoritmo esta bien definido y puede ser aplicado o ejecutado para producir un resultado. A su vez, cada operación suele tener un costo, dependiente del módelo de computación. Conocer el número de operaciones necesarias para transformar la entrada en la salida esperada, i.e., resolver el problema, es de vital importancia para seleccionar el mejor algoritmo para dicho problema, o aun más, para instancias de dicho problema que cumplen con ciertas características.

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

1.3 Modelos de cómputo y tipos de análisis

En los siguientes videos se introduce a los modelos de cómputo y se muestran diferentes tipos de análisis sobre algoritmos.

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

1.3.1 Ordenes de crecimiento

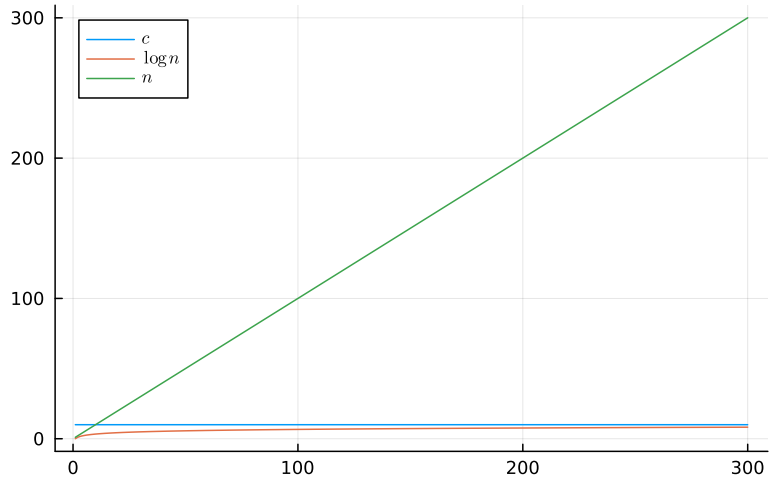
Dado que la idea es realizar un análisis asintótico, las constantes suelen ignorarse, ya que cuando el tamaño de la entrada es suficientemente grande, los términos con mayor orden de magnitud o crecimiento dominarán el costo. Esto es, es una simplificación necesaría.

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.

1.3.1.1 Costo constante, logaritmo y lineal

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

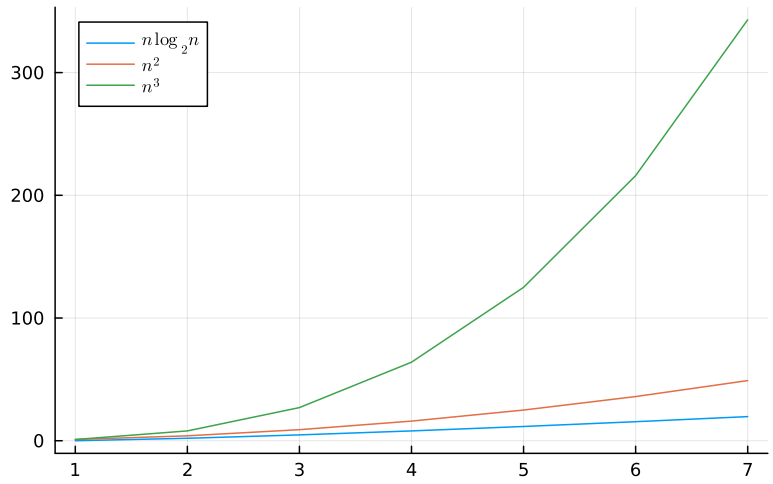
```
using Plots, LaTeXStrings
n = 300 # 300 puntos

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")
```



1.3.1.2 Costo $n \log n$ y polinomial

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

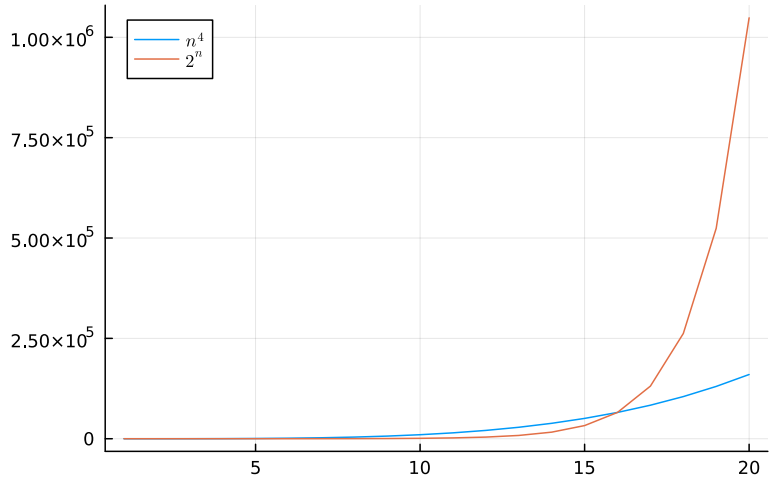
```
using Plots, LaTeXStrings
n = 7 # note que se usan menos puntos porque 300 serían demasiados para el rango

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



1.3.1.3 Exponencial

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

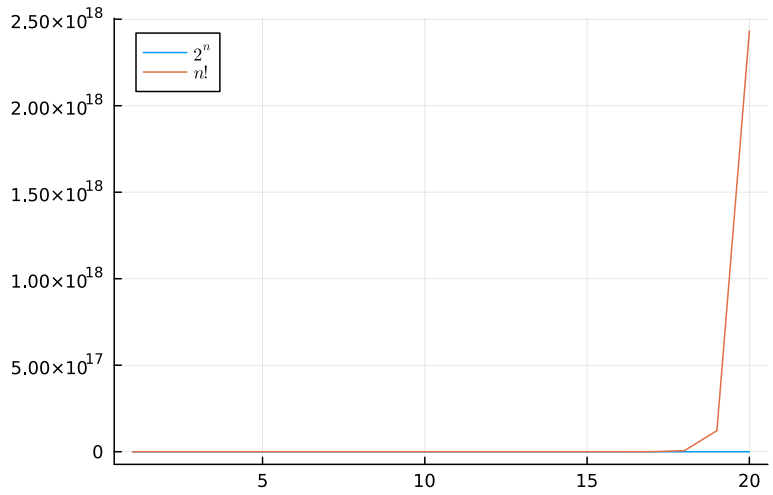
```
using Plots, LaTeXStrings
n = 20

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



1.3.1.4 Crecimiento factorial

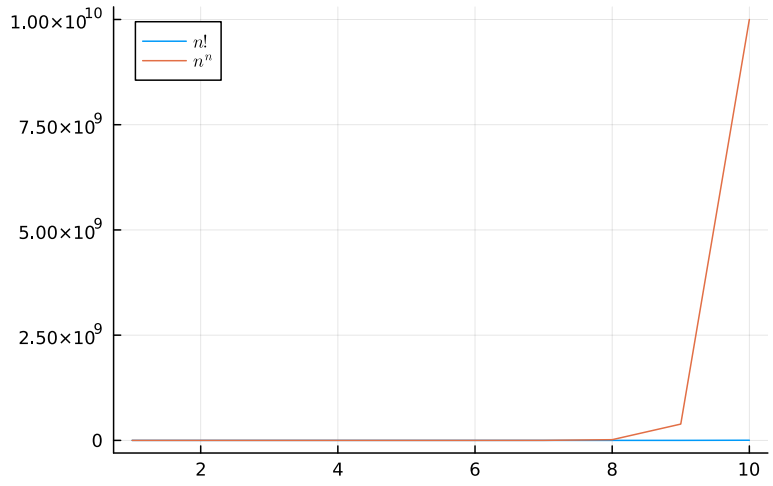
Vease como la función factorial crece mucho más rápido que la función exponencial para una n relativamente pequeña. Vea las magnitudes que se alcanzan en el *eje* y, y comparelas con aquellas con los anteriores crecimientos.

```
using Plots, LaTeXStrings
n = 20

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



1.3.1.5 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.

1.4 Conclusiones

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

1.5 Actividades

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

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

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

1.5.1 Entregable

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

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

1.6 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

2 Estructuras de datos elementales

Objetivo

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

2.1 Introducción

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

2.2 Estructuras de datos abstractas

2.2.1 Conjuntos

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

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

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

2.2.2 Tuplas y estructuras

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

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

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

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

```
280
```

Definición y acceso a los campos de una tupla en Julia

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

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

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

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

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

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

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

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

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

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

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

Funciones sobre diferentes tipos de datos

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

2.2.3 Arreglos

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

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

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

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

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

Figura 2.1: 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))
```

```
(868, 1.9890243f0)
```

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

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

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

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

  C
end

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

```
5×5 Matrix{Float32}:
 0.350852  0.350852  0.350852  0.0      3.0f-45
 1.20811   1.20811   1.20811   0.0      0.0
 1.32639   1.32639   1.32639   0.0      0.0
 0.32793   0.32793   0.32793   4.0f-45  0.0
 0.719805  0.719805  0.719805  0.0      1.0f-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. Dado su implacto, existen diferentes algoritmos para hacer esta operación más eficiente, incluso hay áreas completas dedicadas a mejorar los costos para diferentes casos o características de las matrices.

2.2.4 Listas

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

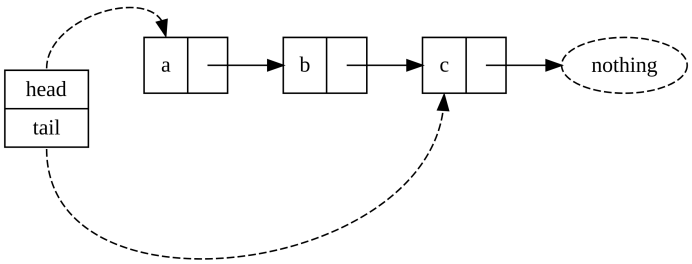


Figura 2.2: Una lista ligada simple

La Figura 2.2 muestra una lista ligada, que es una implementación de lista que puede crecer fácilmente, funciona en el *heap* de memoria por lo que cada bloque requiere memoria dinámica. Cada bloque es una estructura; se pueden distinguir dos tipos, la lista que contiene referencias al primer nodo y al último nodo. Los *nodos de de datos* contienen los elementos de la colección y referencias al siguiente nodo, también llamado *sucesor*. El nodo

nothing es especial y significa que no hay más elementos.

El siguiente código muestra como la definición de lista ligada.

Listado 2.1 Código para una lista ligada simple

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

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

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

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

(10, 20, 30)
```

En el Listado 2.1 se ignora la referencia a *tail* (*head* se guarda en *nodo*), por lo que las operaciones sobre *tail* requieren recorrer la lista completa, costando $O(n)$ en el peor caso para una lista de n elementos.

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

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

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

2.2.4.1 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*?

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

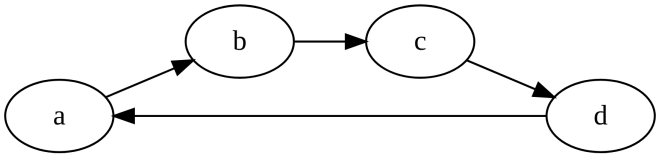


Figura 2.3: Un grafo dirigido simple

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

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

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

La estructura del grafo puede accederse mediante las funciones:

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

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

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

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

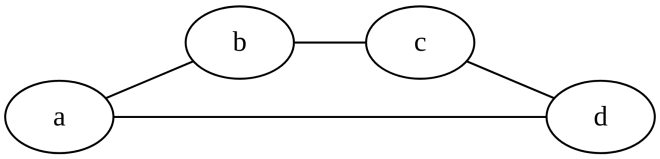


Figura 2.4: Un grafo cuyas aristas no estan dirigidas

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

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

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

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

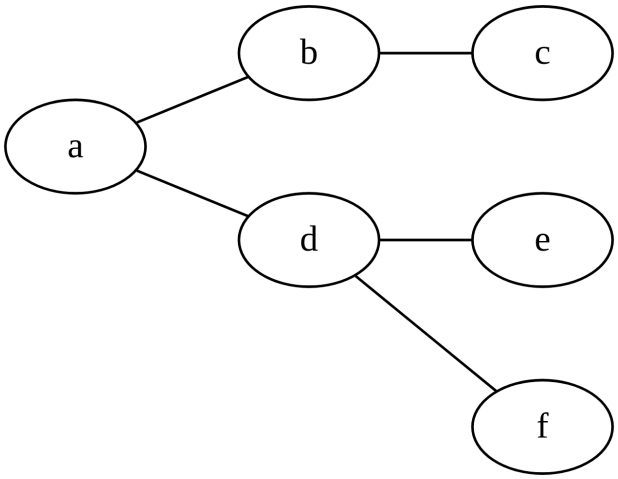


Figura 2.5: Árbol con aristas no dirigidas

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

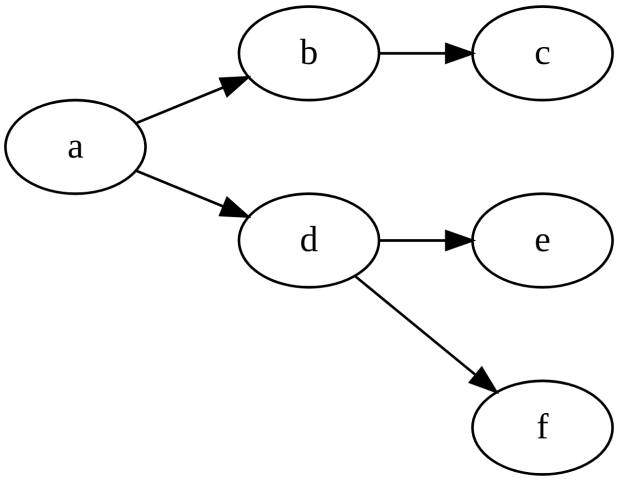


Figura 2.6: Árbol con aristas dirigidas, note que es fácil saber si hay un vértice o nodo que se distinga como raíz, o nodos que sean hojas.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.5.1 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.

2.3 Actividades

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

Entregable

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

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

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

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

2.4 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.

3 Algoritmos de ordenamiento

Objetivo

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

3.1 Introducción

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

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

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

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

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

3.1.1 Lecturas

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

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

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

3.2 Material audio-visual sobre algoritmos de ordenamiento

3.3 Actividades

3.3.1 Actividad 0 [sin entrega]

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

3.3.2 Actividad 1 [con reporte]

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

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

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

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

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

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

3.3.3 Entregable

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

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

4 Algoritmos para codificación de enteros

Objetivos

- Implementar algoritmos de codificación de enteros y su relación con algoritmos de búsqueda
- Implementar algoritmos de compresión de permutaciones y su relación con algoritmos de ordenamiento
- Optimización del memoria utilizado de índices invertidos

4.1 Introducción

La codificación de enteros es una área intimamente relacionada con los algoritmos de búsqueda. En este bloque, se estudiara la conexión entre algoritmos de búsqueda basados en comparaciones y codificaciones de enteros. Así mismo, se verá como la compresión de permutaciones esta relacionada con los algoritmos de ordenamiento.

4.2 Material audio-visual

4.2.1 Codificación - parte 1

4.2.2 Codificación - parte 2

4.2.3 Codificación - parte 3

4.3 Actividades

Considere las listas de posteo de un índice invertido. Como podrían ser aquellas del archivo `listas-posteo-100.json`.

- Representar cada lista de posteo con las diferencias entre entradas contiguas.
- Comprimir las diferencias mediante Elias-γ, Elias-δ, y las codificaciones inducidas por los algoritmos de búsqueda B_1 y B_2 (búsqueda exponencial en 2^i y 2^{2^i})

Nota: Consideré utilizar una biblioteca para manejo de arreglos de bits, por ejemplo - Python: bitarray - Julia: BitArray

4.3.1 Reporte:

- Tiempos de compresión y decompresión
- Razón entre el tamaño comprimido y sin comprimir (*compression ratio*).

Para los experimentos utilizará los siguientes datos:

- REAL: Datos reales, puede usar `listas-posteo-100.json` o puede generarla (vea Unidad 2).
- SIN8: Datos sintéticos con diferencias aleatorias entre 1 y 8, $n = 10^7$.
- SIN64: Datos sintéticos con diferencias aleatorias entre 1 y 64, $n = 10^7$.
- SIN1024: Datos sintéticos con diferencias aleatorias entre 1 y 1024, $n = 10^7$
- Las comparaciones deberán realizarse mediante figuras y tablas que resuman la información.

4.3.1.1 Sobre el reporte

El reporte deberá contener:

- Resumen
- Introducción (debe incluir una clara motivación)
- Planteamiento del problema
- Algoritmos y análisis
- Conclusiones y perspectivas
- Referencias

5 Algoritmos de intersección de conjuntos con representación de listas ordenadas

Objetivo

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

5.1 Introducción

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2 Recursos audio-visuales de la unidad

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

Parte 2: Algoritmos de intersección y algunas aplicaciones

5.3 Actividades

Implementación y comparación de diferentes algoritmos de intersección de conjuntos.

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

5.3.1 Actividad 0 [Sin entrega]

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

5.3.2 Actividad 1 [Con reporte]

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

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

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

3. Realice y reporte los siguientes experimentos:

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

5.3.3 Entregable

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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