Introducción al análisis de algoritmos con Julia

Capítulo 2 — Análisis de Algoritmos con Julia

Eric S. Téllez

Tabla de contenidos

1. Introducción al análisis de algoritmos con Julia

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

1.1. 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 que 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 instancias. En este curso nos enfocaremos en problemas que pueden ser simplificados a una cantidad finita de instancias.

Cada paso u operación en un algoritmo está bien definido y puede ser aplicado o ejecutado para producir un resultado. A su vez, cada operación suele tener un costo, dependiente del modelo 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 pueda ser manipulada de manera eficiente en un contexto específico, además de simplificar la aplicación de operaciones para la aplicación de algoritmos.

1.2. 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).

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

1.2.1. Máquina de Turing

Es un modelo creado por Alan Turing a principios del siglo XX; la idea es un dispositivo que podría ser implementado de manera mecánica si se tuvieran recursos infinitos; esta máquina puede leer y escribir mediante un cabezal en una cinta *infinita* (ver Figura 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 cinta, forman una máquina de estados y memoria, que pueden realizar cualquier cálculo.

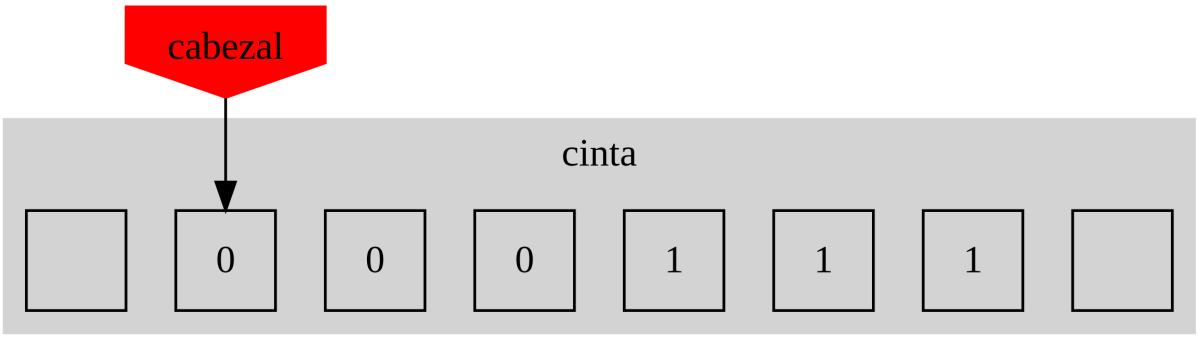


Figura 1: Cabezal y cinta.

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 indica que se debe hacer al leer un símbolo en la posición actual de lectura, esto es, qué se escribe, hacia qué 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 común plantear los problemas en forma de lenguajes; esto es, en términos de *dada la cadena S ¿la máquina M la acepta?*, donde *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 esté definida para un estado y símbolo particular y el no alcanzar un estado de aceptación.

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

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

donde la función de transición se define como:

$$\delta_{01}(q_0, \texttt{0}) = (q_0, \texttt{X}, \texttt{R}) \tag{1}$$
$$\delta_{01}(q_0, \texttt{1}) = (q_1, \texttt{X}, \texttt{R}) \tag{2}$$
$$\delta_{01}(q_1,) = (q_2, , \texttt{R}) \tag{3}$$
$$\tag{4}$$

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

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

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

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

donde la función de transición es como sigue:

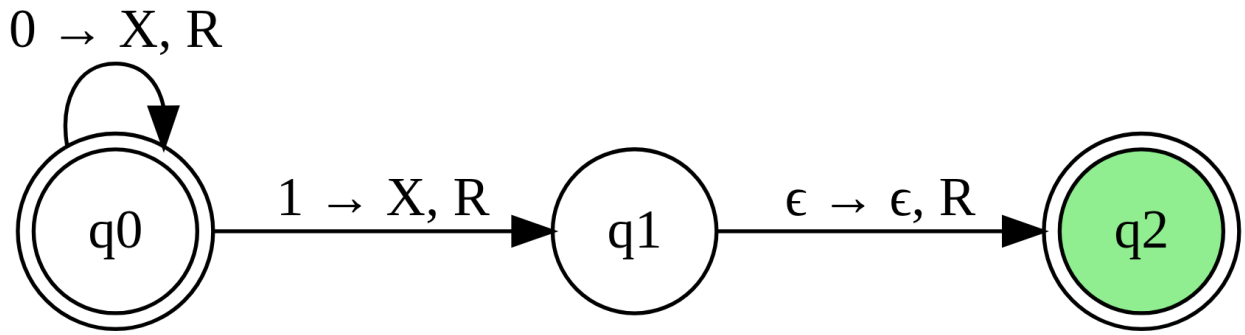


Figura 2: Ejemplo de máquina de Turing que reconoce cadenas 0^n1

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

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 válido 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 moviéndose a través de la cinta para encontrar los correspondientes.

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

1.2.2. Ejercicios

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

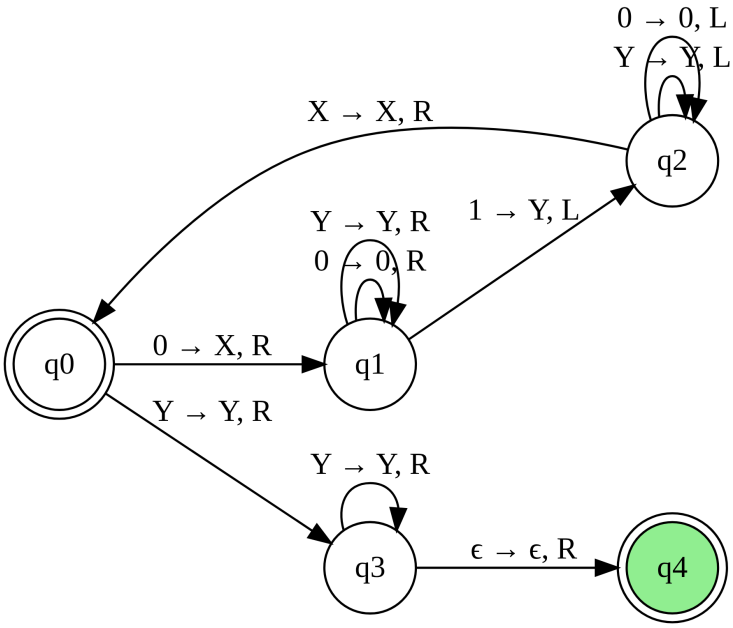


Figura 3: Ejemplo de máquina de Turing que reconoce cadenas 0^n1^n

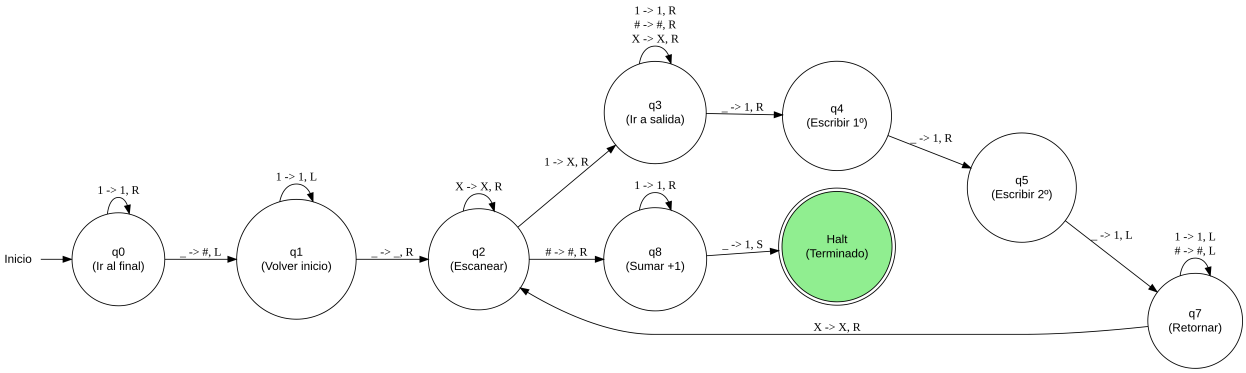


Figura 4: Máquina de Turing que calcula $3n + 1$.

1.2.3. 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 modelo 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 propósitos 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.

1.2.4. Máquina de acceso aleatorio (RAM)

Es un modelo 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 identificado 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 cómo funciona una computadora moderna entre los que hemos revisado.

Una computadora moderna difiere 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 necesaria, 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 está presente en una máquina RAM, tanto a nivel de procesamiento y lectura de datos.

1.3. 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 abordará

el costo realizando mediciones experimentales. Se contrastará con el análisis basado en conteo de operaciones siempre que sea posible.

1.3.1. Ejercicios

1. Considere un fragmento de código con un ciclo doble anidado que recorre un arreglo de n elementos (e.g., para calcular todas las sumas de pares $a_i + a_j$ con $i, j \in \{1, \dots, n\}$). Identifique qué operaciones se consideran de costo $O(1)$ bajo el modelo RAM presentado en esta sección, y calcule el número exacto de dichas operaciones en función de n.

1.4. 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 sí 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 analizar 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*.

1.4.1. Ejercicios

1. Implemente búsqueda lineal en Julia. Identifique explícitamente una instancia de mejor caso y una de peor caso para un arreglo de tamaño n, y proponga una forma de generar entradas aleatorias para estimar el caso promedio de manera empírica usando `@benchmark`. Reporte sus resultados con una tabla o gráfica.

1.5. 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.

1.5.1. Notación Θ

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

$$\Theta(g(n)) = \{f(n) \mid \text{ existen las constantes positivas } c_1, c_2 \text{ y } n_0 \text{ tal que} \tag{5}$$
$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todo } n \geq n_0\} \tag{6}$$
$$\tag{7}$$

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 existir 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.

1.5.2. Notación O

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

$$O(g(n)) = \{f(n) \mid \text{ existen las constantes positivas } c \text{ y } n_0 \text{ tal que} \tag{8}$$
$$0 \leq f(n) \leq cg(n) \text{ para todo } n \geq n_0\} \tag{9}$$
$$\tag{10}$$

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., $\Theta(g(n)) \subseteq O(g(n))$.

1.5.3. Notación Ω

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

$$\Omega(g(n)) = \{f(n) \mid \text{ existen las constantes positivas } c \text{ y } n_0 \text{ tal que} \tag{11}$$
$$0 \leq cg(n) \leq f(n) \text{ para todo } n \geq n_0\} \tag{12}$$
$$\tag{13}$$

Dado que la Ω define una cota inferior, básicamente 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 órdenes 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 órdenes de crecimiento para ignorar detalles y simplificar el análisis de algoritmos.

1.5.4. Ejercicios

1. Para cada uno de los siguientes pares de funciones, determine si $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, o $f(n) = \Theta(g(n))$ (puede aplicar más de una), exhibiendo las constantes c (o c_1, c_2) y n_0 que satisfacen la definición correspondiente:
   - $f(n) = n^2$, $g(n) = n^2 + 3n$
   - $f(n) = n \log n$, $g(n) = n^2$
   - $f(n) = 2^n$, $g(n) = n^{10}$

1.5.5. 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:

1.5.6. Órdenes de crecimiento

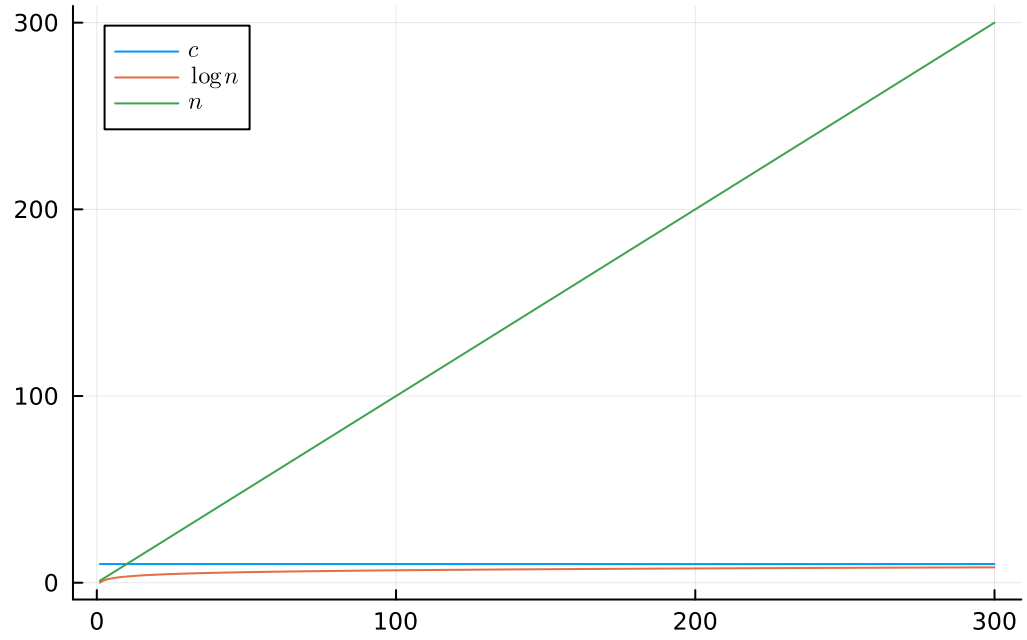
Dado que la idea es realizar un análisis asintótico, las constantes suelen ignorarse, ya que cuando el tamaño de la entrada es suficientemente grande, los términos con mayor orden de magnitud o crecimiento dominarán el costo. Esto es, es una simplificación necesaria.

Los órdenes 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 asintótica nos permite concentrarnos en rasgos 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.

1.5.6.1. Costo constante, logaritmo y lineal

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

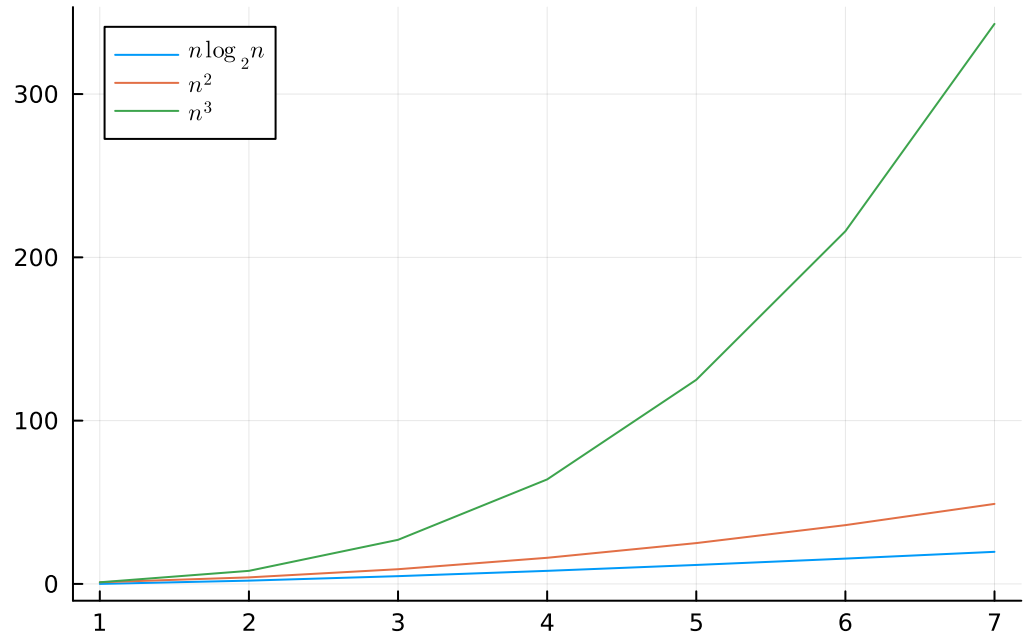
```
using Plots, LaTeXStrings
n = 300 # 300 puntos

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")
```



1.5.6.2. 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 cómo para valores pequeños de n las diferencias no son tan apreciables 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.

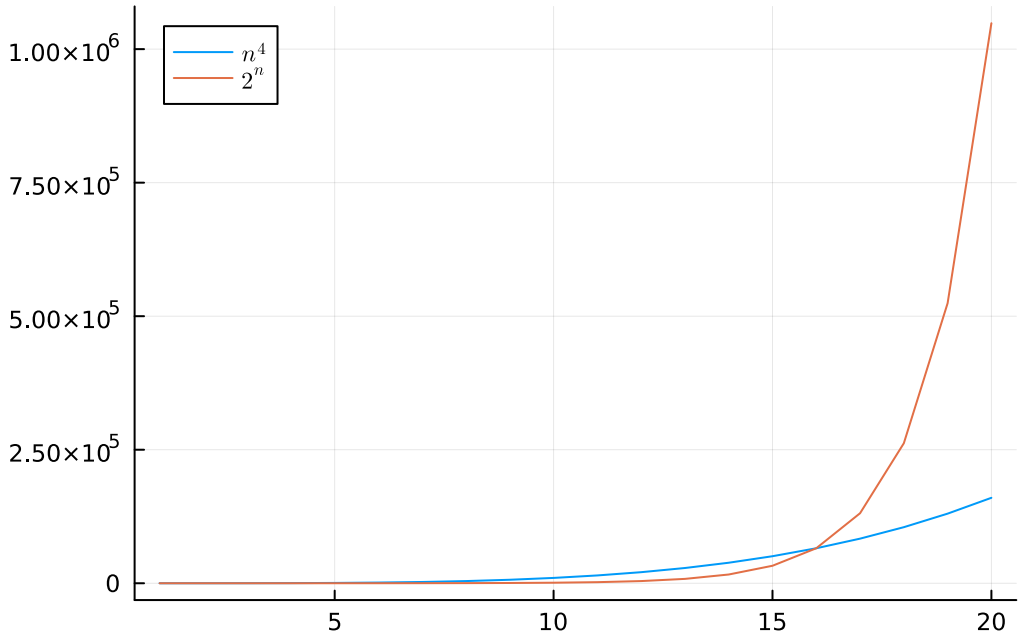
```
n = 7 # note que se usan menos puntos porque 300 serían demasiados para el rango

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")
```



1.5.6.3. 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 rápidamente a la polinomial.

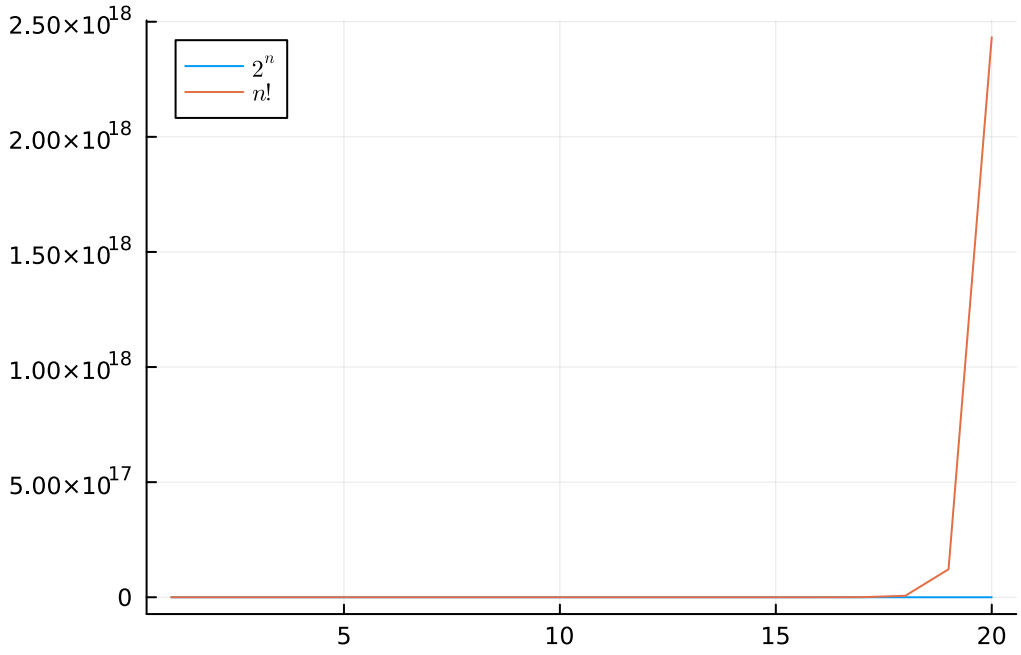
```
n = 20

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")
```



1.5.6.4. Crecimiento factorial

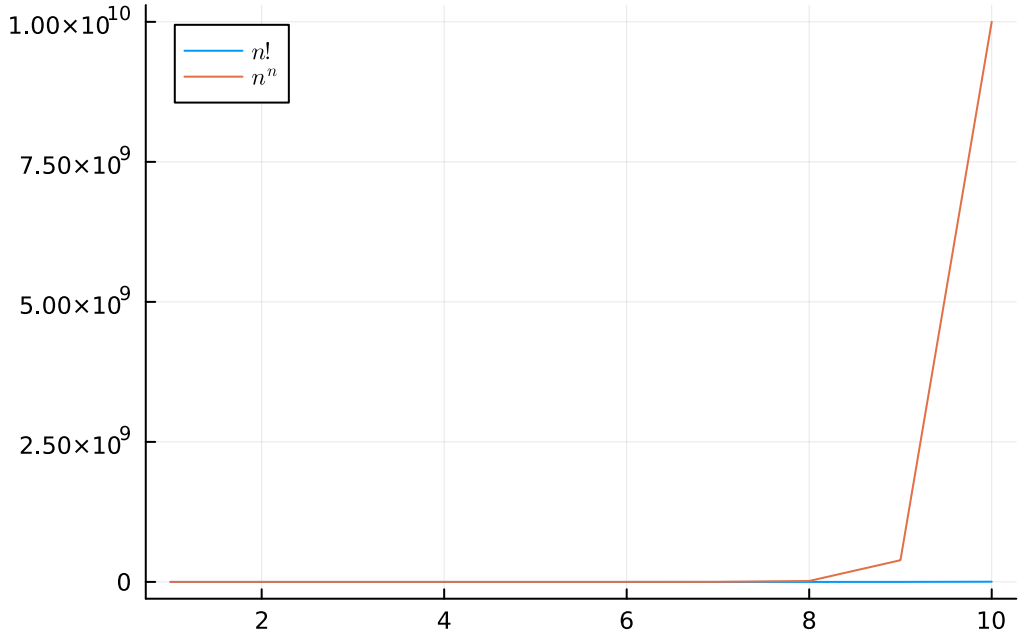
Véase cómo la función factorial crece mucho más rápido que la función exponencial para una n relativamente pequeña. Vea las magnitudes que se alcanzan en el *eje y*, y compárelas con aquellas con los anteriores crecimientos.

```
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!")
```



1.5.6.5. 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 cómo es que cualquier constante se vuelve irrelevante rápidamente; 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, o 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 cuándo es posible solucionarlos, o cuándo 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 abordarán problemas con un costo menor, pero que por la cantidad de datos, i.e., n, se vuelven muy costosos y veremos cómo aprovechar supuestos como las distribuciones naturales de los datos para mejorar los costos.

1.5.7. Ejercicios

1. 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.

1.6. 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 modelo de computación, y siempre pensando en entradas lo suficientemente grandes.

En la práctica, existe una multitud de razones por los cuales los problemas que se resuelven podrían 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 jerarquía 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.

1.6.1. Metodología experimental

Algunos de los algoritmos que se verán en este libro son sumamente rápidos 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 estaremos utilizando las siguientes instrucciones:

- `@time expr` macro que mide el tiempo en segundos utilizado por `expr`, también reporta el número de asignaciones 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 realiza 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 funciones se deben compilar, la primera llamada incluye los costos de compilación.
② El costo sin compilación, hay una asignación que es la matriz donde se guarda el resultado.

```
  0.937496 seconds (2.02 M allocations: 137.010 MiB, 99.97% compilation time)
  0.000013 seconds (1 allocation: 96 bytes)
```

```
3×3 Matrix{Float32}:
 0.737322  1.06583  0.767428
 0.652599  1.40222  0.840803
 1.04762   1.79897  1.30408
```

Tanto `@benchmark` como `@btime` aceptan interpolación de variables con el prefijo `$` para controlar si la evaluación de una expresión se debe contar como parte de lo que se quiere medir o no. Se puede combinar con el parámetro `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 987 evaluations per sample.
 Range (min … max):  48.315 ns …   8.262  s    GC (min … max): 0.00% … 98.65%
 Time  (median):     68.060 ns                 GC (median):    0.00%
 Time  (mean ± ):   75.132 ns ± 202.178 ns    GC (mean ± ):  8.32% ±  3.11%



  48.3 ns       Histogram: log(frequency) by time       151 ns <

 Memory estimate: 96 bytes, allocs estimate: 1.
```

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

```
  48.613 ns (1 allocation: 96 bytes)
```

```
3×3 Matrix{Float32}:
 0.337788  0.622647  0.386497
 0.423687  0.849213  0.510101
 0.636291  0.962026  0.656119
```

El parámetro `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
```

[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 parámetros, como guardar información de corridas.

```
BenchmarkTools.Trial: 1000 samples with 974 evaluations per sample.
 Range (min … max):  62.957 ns …   6.497  s    GC (min … max): 0.00% … 98.00%
 Time  (median):     78.183 ns                GC (median):    0.00%
 Time  (mean ± ):   89.472 ns ± 204.234 ns    GC (mean ± ):  7.12% ±  3.10%


  63 ns         Histogram: log(frequency) by time       179 ns <

 Memory estimate: 96 bytes, allocs estimate: 1.
```

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

_

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

    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 867 evaluations per sample.
 Range (min … max):  136.328 ns … 142.690 ns    GC (min … max): 0.00% … 0.00%
 Time  (median):     136.558 ns                 GC (median):    0.00%
```

```
 Time  (mean ± ):   136.993 ns ±   1.247 ns    GC (mean ± ):  0.00% ± 0.00%


  136 ns        Histogram: log(frequency) by time        141 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 nuestra medida de desempeño, un código donde se capturen las actualizaciones

```
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
  6.0
  7.0
 11.0
```

1.6.3. Ejercicios

1. Considere dos implementaciones de un mismo problema con distinto costo teórico, por ejemplo, la suma de los primeros n enteros mediante un ciclo ($O(n)$) contra la fórmula cerrada $n(n+1)/2$ ($O(1)$). Mida ambas con `@benchmark` para valores crecientes de n y contraste el resultado empírico contra lo esperado por el análisis teórico.

1.7. Actividades

1. Considere dos estructuras para representar un mapa de enteros (`Int64`) a flotantes (`Float64`), sin llaves repetidas:

   - Un vector de pares sin ordenar (e.g., `Vector{Pair{Int64,Float64}}`), donde toda operación recorre linealmente el vector.
   - Un `Dict{Int64,Float64}`, la tabla hash nativa de Julia.

Implemente, para ambas estructuras, las operaciones `buscar`, `insertar!` y `borrar!` con la misma semántica de mapa (a lo más un valor por llave).

Restricción importante: `insertar!` y `borrar!` sobre el vector deben implementarse sin modificar el tamaño (longitud) del arreglo, es decir, sin usar `push!`, `pop!`, `deleteat!` ni operaciones equivalentes que provoquen realojamiento o desplazamiento de memoria. Esto aísla el efecto que se quiere medir (costo del recorrido y localidad de referencia) del efecto de las políticas de crecimiento dinámico de arreglos. Para lograrlo:

- Reserve el vector con una capacidad fija desde el inicio (la n máxima que va a usar en el experimento).
- Use un valor centinela (e.g., una llave reservada que nunca aparezca en los datos) para marcar las posiciones lógicamente "vacías", o use un contador de elementos ocupados (añadiendo la lógica necesaria para llevar a cabo las operaciones).
- `borrar!` sobrescribe la posición encontrada con el centinela; `insertar!` busca primero la llave (para actualizar) y, si no existe, busca una posición marcada como vacía para colocar el nuevo par; ambas búsquedas son recorridos lineales, nunca hay cambio de tamaño del arreglo.

2. Mida con `@benchmark` el tiempo de cada operación (`buscar`, `insertar!`, `borrar!`) para n creciente (potencias de 2 desde 2^2 hasta el máximo práctico en su equipo), con la estructura ya poblada con n pares aleatorios antes de medir. Para `buscar` y `borrar!` la llave a consultar debe elegirse aleatoriamente dentro del `setup=` de cada muestra (no fija), para capturar el caso promedio y no solo el mejor o peor caso posicional. Aplique el mismo cuidado del lado del `Dict` (considere `sizehint!` para controlar el efecto de *rehash* al crecer).
3. Para cada una de las tres operaciones, determine el o los rangos de n donde el vector es más rápido que el diccionario, y viceversa. Reporte con gráficas tiempo-vs-n (escala log-log recomendada) y/o tablas.
4. Discuta sus resultados en términos de localidad de referencia (fallos de caché por saltos de puntero en el diccionario vs. recorrido contiguo en el vector) y el costo de calcular el hash y resolver colisiones.

1.7.1. 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. Vector vs. diccionario para mapas pequeños.
    - 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.

1.8. 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