Análisis de Algoritmos y Estructuras de Datos

Eric S. Téllez

Tabla de contenidos

Prefacio

El *Análisis de algoritmos y estructuras de datos* 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 árboles para resolver problemas optimizando los recursos disponibles, en particular, memoria y tiempo de cómputo.

A lo largo de las unidades que se revisarán, se mostrarán diversas técnicas de desarrollo y análisis de algoritmos aplicadas a problemas como búsqueda en arreglos ordenados, ordenamiento de los mismos, codificación de enteros, operaciones eficientes de conjuntos representados como arreglos ordenados.

Se culminará el curso con un proyecto integrado: la creación de un índice invertido para búsqueda de texto completo. Los índices invertidos son estructuras que se distinguen por su versatilidad en el análisis de datos y su alto desempeño, son una parte fundamental de un motor de búsqueda, así como un posible componente para asegurar la eficiencia en algunos algoritmos de inteligencia computacional.

A lo largo de los temas se abordarán darán detalles teóricos sobre los problemas y los algoritmos, así como también se motivará al estudiante a realizar sus propias implementaciones. En la mayoría de los temas se realizarán análisis experimentales de los mismos y reportes.

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. En cualquier caso, se espera el uso de

foros de discusión foros sobre cualquier problema, ya sea de programación o relacionado a los conceptos y problemas del curso.

Licencia

1 Motivación y explicación del curso

Objetivo

Reconocer la importancia del estudio de algoritmos y estructuras de datos.

1.1 Motivación

El análisis de grandes cúmulos de datos requiere del control completo sobre los algoritmos y las estructuras de datos que se utilicen para manejarlos, dado que los recursos disponibles, como la memoria o la velocidad de procesamiento, de una computadora son finitos y muchas veces pequeños con respecto a la cantidad de datos.

Para este fin, es necesario analizar los recursos necesarios para aplicar un algoritmo sobre una instancia del problema, y así poder seleccionar la mejor opción para dicha instancia, o en su defecto, modificar o crear un algoritmo para adaptarse mejor a las necesidades y recursos disponibles.

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.

1.2 Material audio-visual

- Motivación - parte 1:
- Motivación - parte 2:
- Índices invertidos (proyecto integrador del curso):

1.3 Estructuras de datos básicas

1.4 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.

1.4.1 Recursos para aprender Python y Julia

1.4.1.1 Python

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

1.4.1.2 Julia

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

2 Análisis de la eficiencia de un algoritmo

Objetivo

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

2.1 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.

2.2 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:

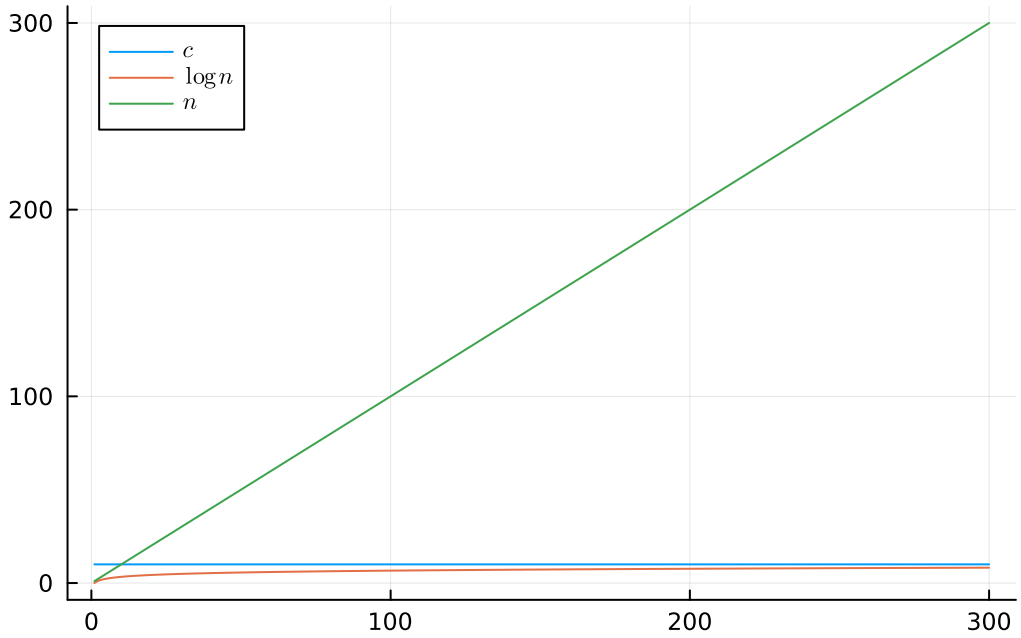
2.2.1 Ordenes de crecimiento

En este punto se puede enfatizar que las constantes suelen ignorarse, esto porque el análisis suele ser asintótico, y es que llegará un momento cuando el tamaño de la entrada sea suficientemente grande, que no importarán las constantes, y las funciones con mayor orden de magnitud o crecimiento dominarán el costo.

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

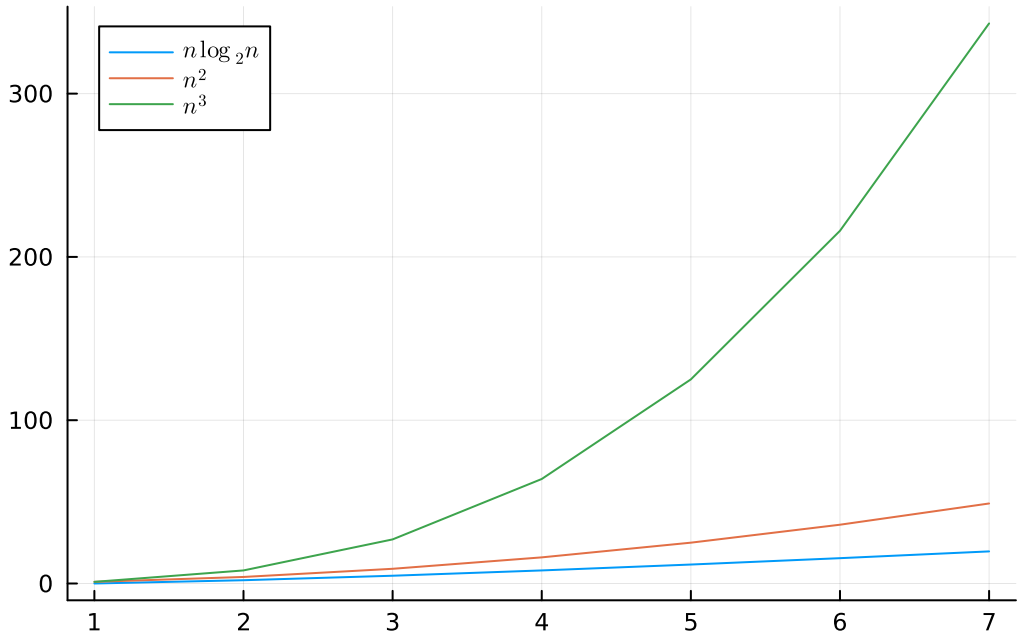
2.2.1.1 Costo constante, logaritmo y lineal

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



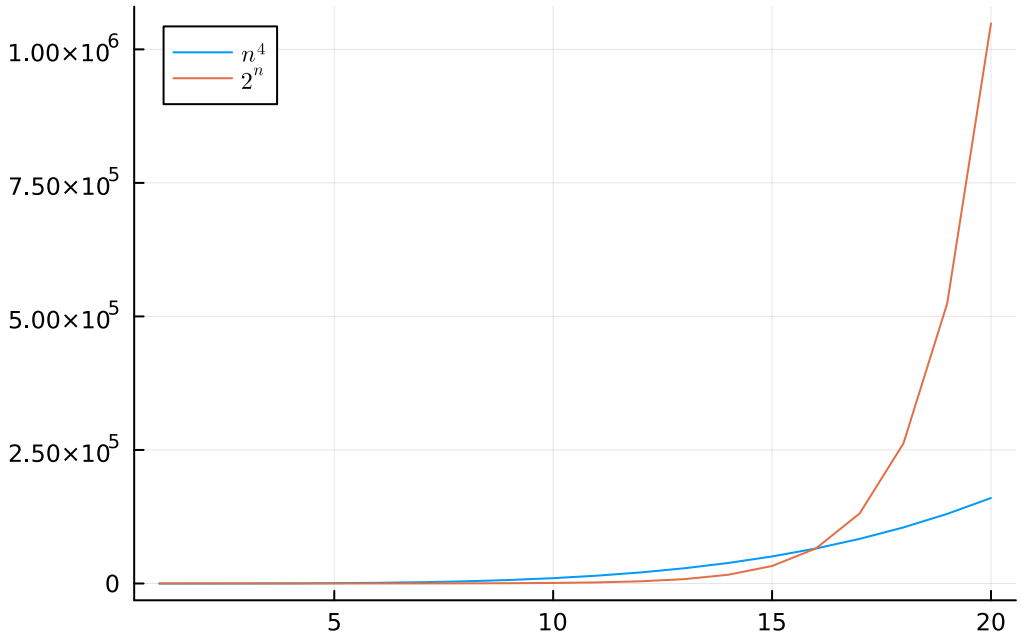
2.2.1.2 Costo $n \log n$ y polinomial

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.



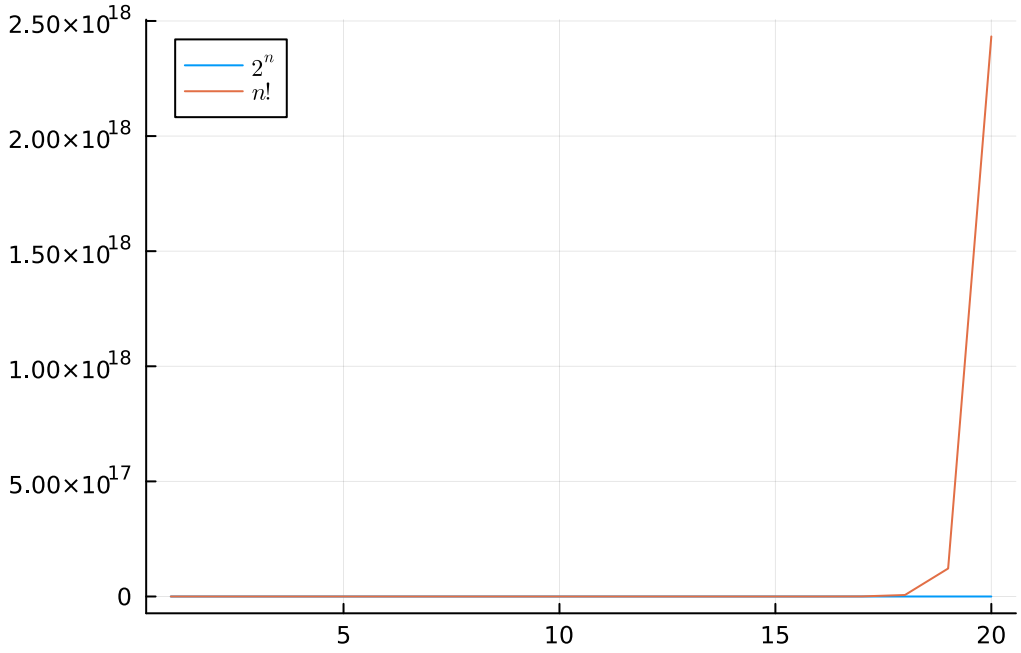
2.2.1.3 Exponencial

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.

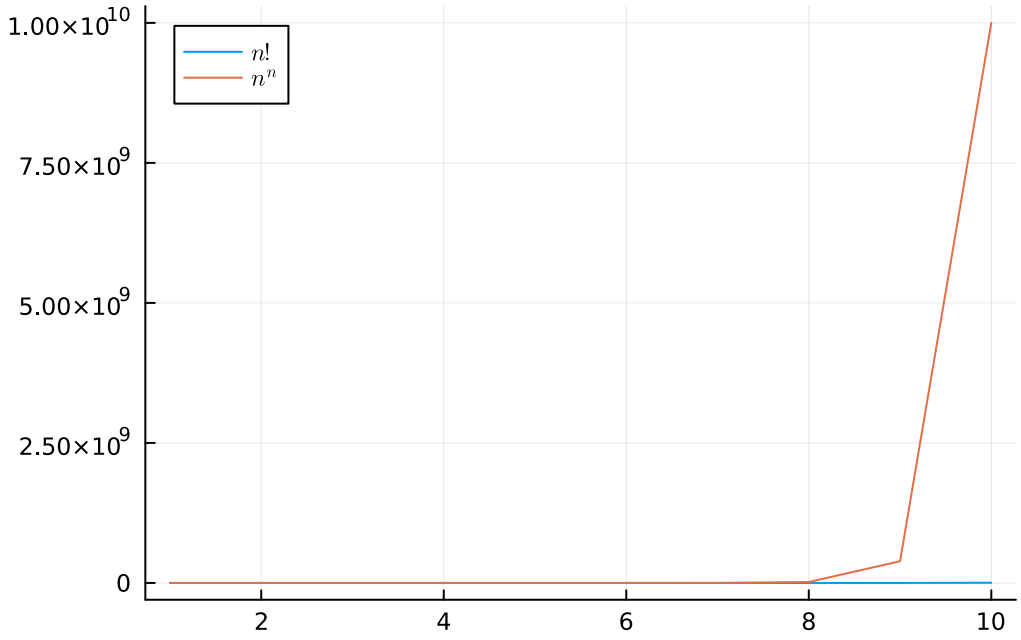


2.2.1.4 Crecimiento factorial

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.



2.2.1.5 Un poco más sobre funciones de muy alto costo



Vea la figura anterior, donde se compara $n!$ con n^n, observe como es que cualquier constante se vuelve irrelevante rapidamente; aun para n^n piense en n^{n^n}.

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

2.3 Conclusiones

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

2.4 Actividades

2.4.1 Actividad 0 [sin entrega]

Instrucciones:

Del libro "Introduction to algorithms" (ver bibliografía). Lee y resuelve los ejercicios de los capítulos 1, 2, 3, 4, 10 y 11. Resuelva ejercicios de dichos capítulos para ganar intuición Nota: En la Actividad 0 no hay entrega; use el Foro de dudas para discutir sus resultados y dudas.

2.4.2 Actividad 1 [con reporte]

– rehacer –

2. Haga una tabla donde simule tiempos de ejecución suponiendo que cada operación tiene un costo de 1 micro-segundo:

- Suponga que cada uno de los ordenes de crecimiento anteriores es una expresión que describe el costo de un algoritmo teniendo en cuenta el tamaño de la entrada del algoritmo n.
- Use como los diferentes tamaños de entrada $n = 100$; $n = 1000$; $n = 10000$ y $n = 100000$.

- Note que para algunas fórmulas, los números pueden ser muy grandes (use el foro de dudas si llega a tener problemas)

3. Dentro del notebook añada un breve ensayo o reflexión sobre los costos de computo necesarios sobre algoritmos que pudieran tener ordenes de crecimiento como los mostrados.

2.4.3 Entregable

Reporte en formato Jupyter que debe contener el código creado y los resultados obtenidos: las figuras generadas, la tabla de simulación. No olvide concluir su reporte con la reflexión solicitada. Tenga en cuenta que los notebooks de Jupyter pueden alternar celdas del notebook con texto, código y resultados; además que el texto puede ser escrito en markdown.

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 ordenes de crecimiento (instrucción 1). Explique los resultados obtenidos.
- Simulación de costo en formato de tabla (instrucción 2), explique los resultados obtenidos.
- Concluya con la reflexión (instrucción 3). La reflexión debe aboradar las comparaciones hechas y la simulación; también toque el tema de casos extremos y una n variable y asintóticamente muy grande.
- Bibliografía

Archivos a entregar: Notebook de Quarto o Jupyter y el PDF del notebook (en el caso de Jupyter, favor de generarlo mediante el mismo Jupyter, no en un procesador de texto, para esto se puede exportar a html con el navegador, y luego el html a PDF).

3 Análisis de la eficiencia de un algoritmo

Objetivo

Implementar, aplicar y caracterizar el desempeño de algoritmos en peor caso y adaptativos para búsqueda en arreglos ordenados.

3.1 Introducción

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

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

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

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

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

3.1.1 Listas ordenadas

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

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

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

3.2 Material audio-visual

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

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

3.2.1 Búsqueda

3.3 Actividades

3.3.1 Actividad 0 [sin entrega]

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

- El artículo sobre búsqueda no acotada, como representativo sobre búsqueda adaptativa (Jon Louis Bentley y Yao 1976).
- Cap. 12 de (Sedgewick 1998), en partícular Sec. 12.3 y 12.4.
- Cap. 6 de (Knuth 1998), en partícular Sec. 6.1 y 6.2.
- El artículo sobre búsqueda adaptativa secuencial (Jon L. Bentley y McGeoch 1985).
- Recuerde la referencia básica para la notación y conceptos es (Cormen et al. 2022).

3.3.2 Actividad 1 [con reporte]

Realice y reporte el siguiente experimento:

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

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.

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

3.3.4 Actividad 2 [sin entrega]

Revisar el notebook `crear-indice-invertido.ipynb` para los detalles de como se generó la lista de posteo. Usted puede crear nuevas listas de posteo si lo desea usando los conjuntos de datos disponibles (listados en dicho notebook), y a su vez utilizarlas en las actividades

de este Unidad. Solo deberá indicarlo; recuerde que los números de documentos y tamaño de vocabulario cambiarán.

3.3.5 Leyendo las listas de posteo

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

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

4 Algoritmos de ordenamiento

Objetivo

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

4.1 Introducción

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

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

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

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

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

4.1.1 Lecturas

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

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

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

4.2 Material audio-visual sobre algoritmos de ordenamiento

4.3 Actividades

4.3.1 Actividad 0 [sin entrega]

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

4.3.2 Actividad 1 [con reporte]

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

- En el notebook `perturbar-listas.ipynb` se encuentran el procedimiento que se utilizó para la perturbación.

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

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

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

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

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

4.3.3 Entregable

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

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

5 Algoritmos 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

5.1 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.

5.2 Material audio-visual

5.2.1 Codificación - parte 1

5.2.2 Codificación - parte 2

5.2.3 Codificación - parte 3

5.3 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

5.3.1 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.

5.3.1.1 Sobre el reporte

El reporte deberá contener:

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

6 Algoritmos de intersección de conjuntos con representación de listas ordenadas

Objetivo

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

6.1 Introducción

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

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

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

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

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

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

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

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

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

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

Existe otra familia de algoritmos, basados en búsquedas adaptativas que pueden llegar a mejorar el desempeño bajo cierto tipo de entradas. En (Demaine, López-Ortiz, y Ian Munro 2001), (Barbay, López-Ortiz, y Lu 2006), (Barbay et al. 2010), y (Baeza-Yates y Salinger 2005) se muestran comparaciones experimentales de diversos algoritmos de intersección, entre ellos adaptables, que utilizan de manera creativa algoritmos de búsqueda adaptables para aprovechar instancias simples. Estos estudios se basan en contribuciones teoricas de los mismos autores (Demaine, López-Ortiz, y Munro 2000), (Demaine, López-Ortiz, y Ian Munro 2001), (Barbay y Kenyon 2002), (Baeza-Yates 2004).

6.2 Recursos audio-visuales de la unidad

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

Parte 2: Algoritmos de intersección y algunas aplicaciones

6.3 Actividades

Implementación y comparación de diferentes algoritmos de intersección de conjuntos.

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

6.3.1 Actividad 0 [Sin entrega]

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

6.3.2 Actividad 1 [Con reporte]

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

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

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

3. Realice y reporte los siguientes experimentos:

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

6.3.3 Entregable

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

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

7 Búsqueda en cadenas de texto

Objetivo

Introducir y comparar algoritmos de búsqueda de patrones en cadenas de símbolos.

7.1 Introducción

La presente unidad esta dedicada a los algoritmos de búsqueda de patrones en cadenas de símbolos. Antes de comenzar vamos a definir el problema:

Dado un alfabeto de simbolos $\Sigma = \{a_1, a_2, \cdots, a_\sigma\}$ una cadena de tamaño $T[1, n]$ esta definida como la concatenación de n símbolos, i.e., $T \in \Sigma^n$. Esto es, si el alfabeto es binario, $\Sigma = \{0, 1\}$, para $n = 3$, T podría ser cualquiera de las siguientes cadenas de bits: $\{000, 001, 010, 011, 100, 101, 110, 111\}$.

Por ejemplo, la cadena $T =$ `ABRACADABRA` esta descrita por el alfabeto $\Sigma = \{A, B, C, D, R\}$, y podemos acceder a cada uno de los simbolos mediante un subíndice, e.g., $T_1 =$ `A`, $T_2 =$ `B`, $T_3 =$ `R`, etc. También se puede acceder a subcadenas haciendo uso de la notación $T_{i:j}$, la cual obtendría una subcadena de tamaño $j - i + 1$ (concatenando los simbolos de que van de i a j en T). Si $j < i$ entonces se accede a la cadena vacia ϵ.

Es posible concatenar cadenas para obtener nuevas cadenas, por ejemplo, $A[1, 11] =$ `ABRACADABRA` y $B[1, 11] =$ `ARBADACARBA`, ambas vienen del mismo alfabeto, y pueden ser concatenadas $C[1, 22] = AB =$ `ABRACADABRAARBADACARBA`. De la misma forma es posible crearlas a partir de concatener cadenas y símbolos.

Ahora, hay tres tipos de subcadenas, según su aparición dentro de otra cadena. Sean A, B, C tres cadenas de símbolos; $D = ABC$ sería la concatenación de las tres cadenas, entonces a A se le llama *prefijo* de D, C es llamado *sufijo* de D y B sería un *factor* de D.

7.1.1 El problema de búsqueda en cadenas

Dada una cadena larga $T[1, n]$ y una cadena corta $P[1, m]$, llamada *patrón*, el problema consiste en encontrar todas las ocurrencias de P en T, esto es, todos los puntos donde P sea una subcadena en T. Esta operación es llamada *búsqueda*. Para esto se pueden tener dos variantes típicas:

1. T es estático o cambia muy poco, por lo que se puede crear una estructura sobre T para resolver las búsquedas.
2. T varia frecuentemente, o no esta acotado, por lo que la estrategía es apoyarse en una estructura sobre P para resolver consultas de manera eficiente.

7.1.2 Esquemas de búsqueda

Para el primer problema, se puede crear algún índice tipo índice invertido si el texto ser divido en pequeñas piezas o tokens, e.g., palabras y símbolos de puntuación en texto escrito en lenguaje natural (y con pocas propiedades aglutinantes). De otra forma requerirá índices similares a árboles de sufijos o arreglos de sufijos. Estas últimas estructuras se basan en crear un árbol que contenga todos los sufijos (mediante punteros e índices al texto original). (Ver video)

Para el segundo problema, la idea general es revisar la cadena T en ventanas de tamaño m y calcular una estructura sobre P tal que usando la información de la ventana siendo analizada sea posible avanzar la ventana de manera segura (sin perder información) y rápida (moverla lo más posible). Para esto se usan los prefijos, sufijos y factores de P y la ventana en cuestión. Esta unidad se enfoca en este segundo problema.

7.1.3 Autómatas finitos y búsqueda de cadenas

Muchos de los algoritmos de búsqueda de patrones en cadenas estan basados en algoritmos sobre autómatas finitos, por lo que nos remitiremos a dicha estructura, el *autómata finito*.

7.1.3.1 Autómatas Finitos

Para nuestros propósitos, un autómata finito es una estructura discreta formada por una serie de estados Q, entre los cuales hay dos tipos de estados especiales. El estado inicial $I \in Q$ y los estados terminales $F \subseteq Q$. Entre cada par de estados puede existir una transición, etiquetada por elementos de un alfabeto $\Sigma \cup \{\epsilon\}$; estas transiciones son descritas de manera precisa mediante una función especial llamada función de transición $\mathcal{D}(q, \alpha) = \{q_1, \cdots, q_k\}$ esto es, asocia estados Q por medio de símbolos $\alpha \in \Sigma \cup \{\epsilon\}$. Un autómata es descrito por estas partes como $A = (Q, \Sigma, I, F, \mathcal{D})$.

Dependiendo de la función de transición, podemos distinguir dos tipos de autómatas. El autómata determinista (DFA) es aquel donde $\mathcal{D}$ asocia pares de estados; y $\mathcal{D}$ puede definirse en terminos de una función parcial $\delta : Q \times \Sigma \to Q$. El no determinista (NFA) puede asociar diferentes estados usando el mismo cáracter de transición, $\mathcal{D} : Q \times \Sigma \to \{q_1, \cdots, q_k\}$ para $k > 1$, así como también cuando hay alguna transición definida por la cadena vacia, i.e., $\mathcal{D}(q, \epsilon)$. El NFA suele ser mucho más sucinto en cuanto a su descripción formal, lo que hace preferible para trabajar en la práctica; sin embargo, es necesario hacer notar que NFA y DFA son equivalentes en cuanto a su capacidad de expresión de cadenas.

Las siguientes figuras muestran un par de autómatas finitos. El primero es un DFA que es capaz de reconocer las palabras `ABRACADABRA` y `CABRA`. El segundo autómata es no determininista (NFA) y reconoce `ABRACADABRA` y todos sus sufijos, i.e., `BRACADABRA`, `RACADABRA`, `ACADABRA`, `CADABRA`, `ADABRA`, `DABRA`, `ABRA`, `BRA`, `RA`, y `A`.

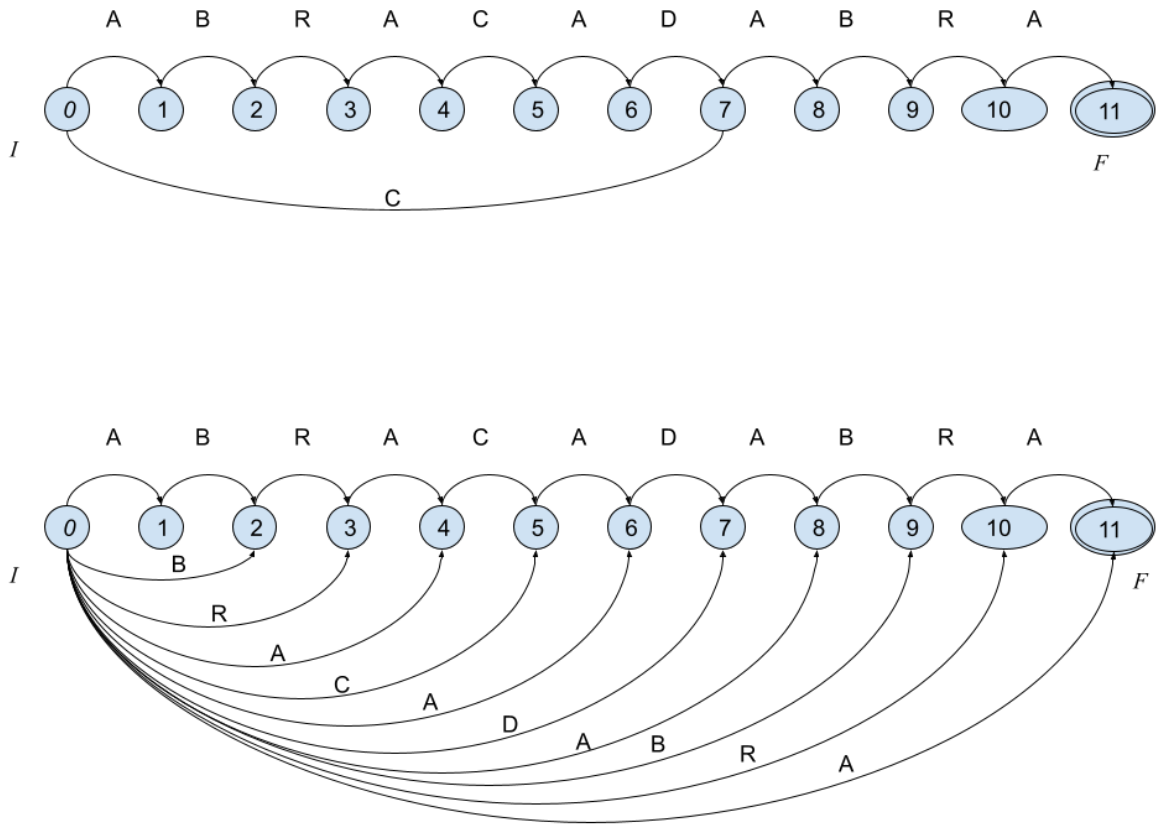


Figura 7.1: Autómatas DFA (arriba) y NFA (abajo)

7.1.4 Algoritmos

Muchos de los algoritmos que trabajan sobre textos que varian mucho, no estan acotados, o simplemente son demasiado grandes para poder ser preprocesado, utilizan alguna estructura basada en un autómata finito para acelerar la resolución de búsquedas.

Como ya se había comentado, dichos algoritmos intentarán revisar el texto $T[1, n]$ usando ventanas w del tamaño del patrón $P[1, m]$. Dichas ventanas deberan ser probadas en la estructura con el fin de observar si es posible o no un emparejamiento con P. El objetivo de la estructura y el algoritmo será avanzar tan adelante como sea posible la ventana sin perder posibles ocurrencias.

Considere el siguiente ejemplo:

```
    1  2  3  4  5  6  7  8  9  10 11
T = A  B  R  A  C  A  D  A  B  R  A
    -------
       -------  w
          -------
             -------

P = A B R
```

Se creará un autómata $(\{0, 1, 2, 3\}, \{A, B, C, D, R\}, 0, \{3\}, \mathcal{D})$. A continuación se ven dos posibles autómatas que pueden usarse para resolver las búsquedas:

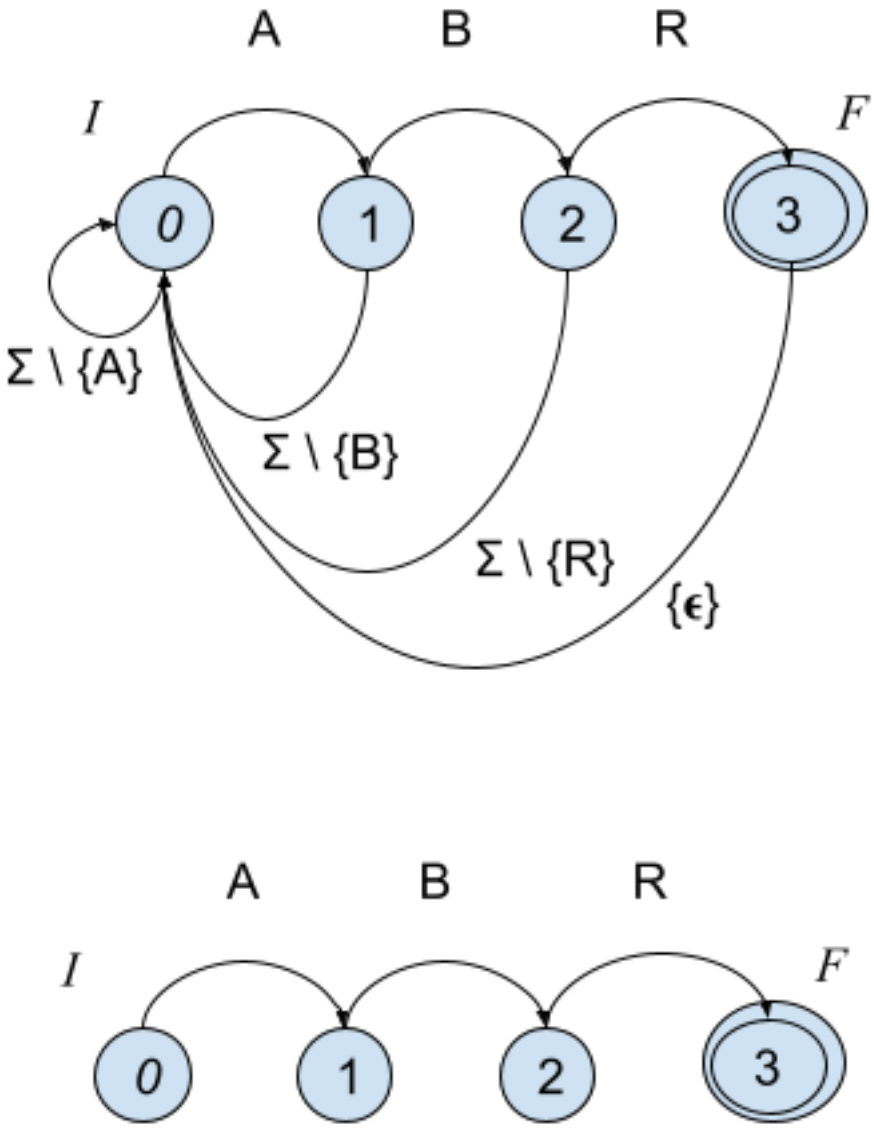


Figura 7.2: Autómatas ABR

La diferencia viene en la definición de $\mathcal{D}$. El primero puede consumir cáracter por cáracter

de T y reportar una ocurrencia cada vez que se toque el estado F (reportar la posición en T dónde ocurre). El segundo hace uso del concepto de ventana, para cada ventana solo existirá un emparejamiento si iniciando en 0 se termina en el estado 3. En parte, los algoritmos verán como avanzar la ventana de manera más eficaz; es posible leer las prefijos o sufijos de las ventanas, así mismo, es posible utilizar información de factores para mejorar el deslizamiento de la ventana. Para más información referirse a [NR02].

Con la representación basada en autómatas es posible consider clases de caracteres, e.g., dígitos numéricos, caracteres alfabeticos, puntuaciones, o en general conjuntos de símbolos que se deseen agrupar. Así como soportar cualquier tipo de expresión regular [NR02].

7.1.5 El algoritmo *Shift-And*

Una de los algoritmos más sencillos y eficientes es el algoritmos de Shift-And, el cual consiste en simular el NFA usando operaciones a nivel de bits. En particular, este algoritmo es muy veloz en patrones que quepan en la palabra de la computadora donde se aplica (e.g., 32 o 64 bits); cuando el patrón sea más largo que el tamaño de la palabra, las operaciones pueden ser implementadas teniendo en cuenta los corrimientos a nivel de bits que pudieran surgir en las operaciones. Dado que las operaciones a nivel de bits se realizan de manera paralela, estas pueden realizarse de manera muy eficiente.

```
    1  2  3  4  5  6  7  8  9  10 11
T = A  B  R  A  C  A  D  A  B  R  A

P = A  B  R
```

Como se había observado, es suficiente tener 4 estados para este patrón. Es necesario crear la tabla D que codifica $\mathcal{D}$. Para construirla, es necesario codificar el alfabeto en una matriz binaria de $|\Sigma| \times m$ elementos (i.e., longitud del alfabeto $\times$ longitud del patrón). Donde cada fila corresponde a los caracteres del alfabeto Σ y las columnas a los estados (que a su vez corresponden con el patrón P); cada fila en D codifica con 1 si para cada estado, el carácter se encuentra en el patrón en la columna correspondiente, y 0 si no lo hace. Adicionalmente, se debe considerar que el estado inicial tiene un transición a sí mismo con la cadena vacia ϵ. Para nuestro ejemplo, la matriz quedaría como sigue:

```
    R  B  A        <- P reverso para su codificación
    3  2  1  0     <- estados
    F        I     <- estados de fin e inicio
A   0  0  1  0   \
B   0  1  0  0    |
C   0  0  0  0    |  codificación de la función
```

```
D   0  0  0  0    |  de transición D
R   1  0  0  0    |
eps 0  0  0  1   /
```

El patrón y el contador estan revertidos para denotar su posición en la codificación binaria. Note que se ha añadido una transición de cadena vacia en el estado 0.

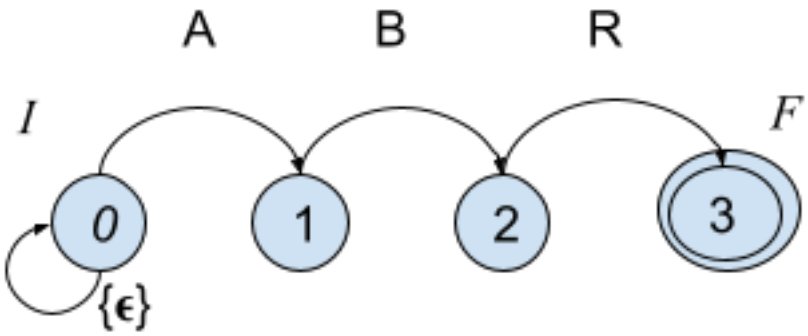


Figura 7.3: Autómatas ABR

Shift-And es un algoritmo bastante simple y eficiente, que recorre el texto por ventanas, haciendo uso del autómata del patrón.

A continuación se muestra una implementación en lenguaje Julia.

```julia
function pattern(pat::T) where T
    D = Dict{eltype(pat),UInt64}()
    for i in eachindex(pat)
        c = pat[i]
        d = get!(D, c, zero(UInt64))
        d |= 1 << (i-1)
        D[c] = d
    end
    D
end

function search(text, pat, L=Int[])
    D = pattern(pat)
    S = 0
```

```
    plen = length(pat)
    m = 1 << (plen - 1)
    for i in eachindex(text)
        d = get(D, text[i], 0)
        S = ((S << 1) | 1) & d
        if S & m > 0
            push!(L, i-plen+1)
        end
    end
    L
end
```

La función `pattern` construye de manera parcial la tabla D, mientras `search` implenta el algoritmo Shift-And. La operación más importante para entender del algoritmo esta en la línea `S = ((S << 1) | 1) & d`; donde la transición por ϵ en el estado cero se realiza mediante la operación a nivel de bits $|1$, se simula las transiciones en el autómata mediante `S<<1` y `& d` hace el emparejamiento con el cáracter que esta siendo leído. Note también, que `d` se pone a cero cuando el cáracter no esta en D (i.e., esta en T pero no en P). Las ocurrencias se ponen en `L` y estas ocurren cuando `S` tiene un 1 en la última posición del patrón, i.e., el estado final F esta activo.

Al correr la función, tenemos lo siguiente

```
julia> search("ABRACADABRA", "ABR")
2-element Vector{Int64}:
 1
 8

julia> search("MISSISSIPPI", "SS")
2-element Vector{Int64}:
 3
 6

julia> search("MISSISSIPPI", "I")
4-element Vector{Int64}:
  2
  5
  8
 11
```

Por las características de Julia, podemos cambiar fácilmente el tipo de los datos y seguir obteniendo una buena eficiencia.

```
julia> A = rand(1:6, 1000_000_000)
julia> @time search(A, [3,1,4,1,6]);
31.544242 seconds (22 allocations: 2.001 MiB)
```

#€ Actividades

7.1.6 Actividad 0 [Sin entrega]

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

7.1.7 Actividad 1 [Con reporte]

1. Sea T el contenido del archivo `pi-1m.txt`, éste contiene el primer millón de dígitos de π (tomado de https://newton.ex.ac.uk/research/qsystems/collabs/pi/). También puede usar los archivos de datos que hemos usado pero debería adaptar y explicar la adaptación en el reporte.

- Considere que $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Sea $A \subseteq \Sigma^4$, $|A| = 1000$; seleccione de manera aleatoria A.
- Sea $B \subseteq \Sigma^8$, $|B| = 1000$; seleccione de manera aleatoria B.
- Sea $C \subseteq \Sigma^{16}$, $|C| = 1000$; seleccione de manera aleatoria C.
- Sea $D \subseteq \Sigma^{32}$, $|D| = 1000$; seleccione de manera aleatoria D.
- Sea $E \subseteq \Sigma^{64}$, $|E| = 1000$; seleccione de manera aleatoria E.

2. Implemente el algoritmo Shift-And o use el que se proporciona antes.

3. Implemente el algoritmo naïve que consiste en verificar ventana a ventana por emparejamiento sin usar operaciones a nivel de bits y avanzando uno en uno los carácteres.

4. Realice y reporte los siguientes experimentos:

- Para cada $p \in A$ busque p en T y reporte de manera acumulada el tiempo en segundos, compare Shift-And y el algoritmo naïve usando figuras `boxplot`.
- Para cada $p \in B$ busque p en T y reporte de manera acumulada el tiempo en segundos, compare Shift-And y el algoritmo naïve usando figuras `boxplot`.
- Para cada $p \in C$ busque p en T y reporte de manera acumulada el tiempo en segundos, compare Shift-And y el algoritmo naïve usando figuras `boxplot`.
- Para cada $p \in D$ busque p en T y reporte de manera acumulada el tiempo en segundos, compare Shift-And y el algoritmo naïve usando figuras `boxplot`.
- Para cada $p \in E$ busque p en T y reporte de manera acumulada el tiempo en segundos, compare Shift-And y el algoritmo naïve usando figuras `boxplot`.

7.1.8 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.

8 Proyecto integrador

Objetivo

Integrar los temas del curso en un proyecto final.

8.1 Introducción

Durante el curso se estudiaron fundamentos para el análisis de algoritmos, y algoritmos para resolver diferentes tipos de problemas, como ordenamiento, búsqueda, codificación de conjuntos, e intersecciones de conjuntos. Todos estos temas se articulan en la construcción de índice invertido, que permite realizar búsquedas eficientes en colecciones. Puede ser utilizado para realizar análisis de grandes colecciones de datos no estructurados, y en particular, de documentos. Su uso puede ir desde análisis exploratorio, es decir, un análisis cuyo objetivo es ayudar al científico de datos a conocer y descubrir características de la colección, así como para realizar clasificación automática si se utiliza una colección etiquetada.

8.2 Actividad [con reporte]

Instrucciones:

En un notebook de Jupyter construye un índice invertido a partir de una de las colecciones provistas.

Diseña un algoritmo para resolver consultas conjuntivas (terminoA & termino B & …) utilizando el algoritmo de intersección de su elección (recuerde que debe explicar porque eligió ese algoritmo en particular).

Crea una celda en el notebook donde se construya el índice invertido (no importa que se definan las funciones necesarias en otras celdas). Use cualquiera de los conjuntos de datos que hemos usado en el curso o incluso el emo50k.json.gz que se explica en el video.

Integra otra celda en el notebook para realizar las consultas (no importa que se definan las funciones necesarias en otras celdas)

Realiza una revisión de los algoritmos vistos en el curso, repasa los experimentos y comparaciones en cada tema, discuta los resultados de cada tema.

Tu reporte de actividad debe incluir:

- Tu nombre
- Introducción a los índices invertidos
- Resume
- Explicación de cada tema y discusión de los resultados experimentales.
- Descripción de su índice invertido y de como funcionan las celdas para construcción y consulta
- Conclusiones
- Referencias

Entrega tu actividad en formato Notebook de Jupyter y el PDF del mismo Notebook.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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