TEORÍA DE ALGORITMOS 1
Introducción

Por: ING. VÍCTOR DANIEL PODBEREZSKI
vpodberezski@fi.uba.ar

1 Conceptos iniciales

> *"Es tentador pensar que, si la única herramienta que tienes es un martillo, puedes tratar cualquier cosa como si fuera un clavo"* [1]

Se conoce como "Martillo dorado" o "Martillo Maslow" al concepto de utilizar una herramienta, metodología o procedimiento para resolver cualquier problema que se nos presenta, sin importar que esa elección sea equivocada. Este accionar se puede deber a un empecinamiento o un desconocimiento de alternativas. Si estamos en presencia del primer caso, poco, tal vez, se puede hacer. Sin embargo el segundo caso tiene solución: inspeccionar, investigar, ejercitar… aprender

El propósito de este escrito es brindar una guía para construir una caja de herramientas variada llena de herramientas para múltiples usos. De manera que al enfrentarse a diversos retos sepamos manipularlas para obtener la mejor solución posible haciendo un uso inteligente de los recursos con los que contamos. No analizaremos cualquier tipo de problema. Sería necio e inabarcable tratar de hacerlo. Nos enfocaremos en un subconjunto de problemas lógicos-matemáticos que un futuro profesional informático puede enfrentar en su carrera. Nuestras herramientas serán algorítmicas.

El método de trabajo consistirá en la presentación de problemas con ciertas características en común. Para ellos podremos conseguir algoritmos de resolución que se aprovechen de estas singularidades. Podremos entonces hablar de "familias", "estrategias" o "metodologías" de algoritmos. Veremos que trabajando en diferentes estrategias, algunos

[1] Un concepto similar fue formulado por Abraham Kaplan dos años antes (1964) en "The Conduct of Inquiry: Methodology for Behavioral Science" que puede resumirse como "Dale a un niño un martillo, y descubrirá que todo lo que encuentra debe ser martillado"

problemas volverán a aparecer. Esto es porque muchas veces un mismo problema acepta múltiples formas de resolución.

Estudiaremos cada uno de estos algoritmos. Este "Análisis de los algoritmos" cubrirá diferentes campos. Algunos de los pilares que veremos serán su optimalidad y su eficiencia. Pero no descuidaremos también las consecuencias de aplicación con respecto a la solución encontrada.

Para que el aprendizaje sea exitoso - además de dedicación y paciencia - se requiere una base sólida desde donde partir. Un conjunto de conocimientos que daremos por dominados. Entre ellos:

- Nociones de algoritmia y manejo de estructuras de datos elementales.
- Conocimientos de lógica matemática y análisis matemático.
- Nociones básicas de probabilidad

2 Problemas y tipos de problemas

El concepto de "Problema" es tan natural y corriente que se hace difícil de definir. Se puede indicar que corresponde a una tarea a resolver. Partimos de una situación y queremos responder una pregunta sobre esta. Podemos ver un problema computacional entonces como la resolución de un problema que requiere del cómputo para responder esa pregunta. En nuestro cómputo la situación inicial será la entrada ("input") y la respuesta la salida ("output").

Llevándolo a un terreno matemático más abstracto podemos ver entonces a un problema como la relación entre dos conjuntos. El primero abarca todas las posibles entradas del mismo. El segundo contiene todas las salidas posibles. Llamaremos instancia a cada uno de los elementos del primer conjunto. Llamaremos solución a cada uno de los elementos del segundo. El enunciado del problema determina la relación entre cada instancia y cada solución.

Si nuestro problema es "Ordenar una lista de números naturales de mayor a menor". Una instancia posible será "4,7,3,8,2" y estará relacionada con la solución "2,3,4,7,8". Asimismo la instancia "3,8,4,2,7" estará relacionada también con la solución anterior. Por otro lado la instancia "1,2,3,4" estará relacionada con la solución "1,2,3,4"

En la naturaleza del problema está la relación entre los conjuntos de instancias y soluciones. Pero no hay una indicación de cómo encontrarlo. Requerimos de un algoritmo para hacerlo. Es este quien materializa y convierte la abstracción del problema a un ejercicio práctico.

Existen diferentes tipos de problemas según la pregunta que deseamos responder. Entre ellos:

- Problemas de evaluación: Calcular el valor óptimo de un problema numérico.
  Ejemplo: Dadas dos matrices cuadradas de n elementos calcular su multiplicación
- Problemas de búsqueda: Obtener una respuesta que satisfaga la situación planteada.
  Ejemplo: Dado un origen y un destino dentro de una red vial, encontrar un camino que los una.
- Problemas de optimización: Obtener una respuesta que maximice o minimice la situación planteada.
  Ejemplo: Dados un conjunto de elementos con un peso y valor determinado y una mochila que puede llevar como mucho n kilos y, seleccionar un subconjunto de mayor valor posible que no supere la capacidad máxima
- Problemas de decisión: Responder si se cumple o no una condición. Las respuestas posibles son “SI” y “NO”.
  Ejemplo: Dado un mapa político, ¿es posible pintar las diferentes regiones con no más de X de colores diferentes de forma que no queden regiones limítrofes con el mismo color?
- Problemas de enumeración combinatorio: Obtener el conjunto de todas las soluciones que satisfagan cierta pregunta
  Ejemplo: Obtener las diferentes formas que un grupo de diplomáticos se pueden sentar alrededor de una mesa previendo que aquellos de países con conflictos no queden a un lado del otro.

Diremos que encontramos una solución a un problema cuando encontramos un algoritmo que resuelve de forma óptima cualquier instancia del mismo. Se debe demostrar que esto ocurre y al hacerlo diremos que el algoritmo es óptimo para el problema.

Al resolver un problema es necesario abstraerse de la situación real. Convirtiendo en números, conjuntos, listas y otras estructuras de datos las entidades reales. Esto ayuda a generalizar y resolver. Veremos entonces que varios problemas diferentes pueden tener tratamientos similares. La desventaja de este enfoque es que puede esconder efectos reales de la aplicación de cierto mecanismo seleccionado. Cobra vital importancia cuando al resolver cierta instancia de un problema existen varias soluciones óptimas posibles. Y nuestro algoritmo de resolución encuentra solo una de ellas. Un análisis moral (que excede el alcance de este escrito) es necesario. Más aún cuando en base a una decisión de resolución vidas de personas se pueden ver afectadas. Vemos un simple ejemplo a continuación.

Un sistema de inscripción dentro de una facultad otorga un número de prioridad entre 1 y 50 a cada uno de sus 5 mil alumnos según su historial académico. En un primer proceso recibe un listado de alumnos ordenado por su documento nacional de identidad. Calcula y agrega un puntaje académico a cada uno de ellos. Un segundo proceso ordena a los alumnos según el puntaje calculado y separa en grupo de 100 otorgando a cada grupo su prioridad.

Se puede ver al segundo proceso como un simple ordenamiento que tiene como particularidad que la clave de ordenamiento es el puntaje académico y que puede no ser único. Varios alumnos pueden tener el mismo. Nos resta seleccionar un algoritmo de ordenamiento para encarar este problema. Por ejemplo podemos seleccionar TimSort, MergeSort o InsertionSort entre otros. Cualquiera de ellos dará la respuesta en forma óptima. Veremos que traen una consecuencia no esperada en la respuesta obtenida. Una que podría ser catalogada como “discriminatoria”.

Todos los algoritmos de ordenamiento mencionados tienen como propiedad ser “estables”. ¿Qué significa esto? Que mantienen el orden relativo de los elementos con la misma clave. Todos los alumnos con el mismo puntaje académico quedarán ordenados de la misma manera en la que llegan al proceso: ordenados por su número de identificación. Veamos un caso extremo:

Por una regulación en el inicio del uso del nuevo sistema de inscripción todos los alumnos cuentan con el mismo puntaje académico. Por ejemplo el “10”.

Podemos ver claves a ordenar en una lista “10, 10, 10, ….”. Para nuestros algoritmos de ordenamiento cada 10 no es diferente al resto y el resultado ordenado cumple con la resolución del problema. Sin embargo, si tenemos en cuenta que los alumnos más jóvenes tienen un número de identificación más grande, ellos quedarán (por el ordenamiento preexistente) con la peor prioridad posible. ¿Es esto discriminación por edad?

La propiedad de estabilidad no es mala para un algoritmo de ordenamiento per se. Existen situaciones donde esta propiedad es fuertemente buscada. Sin embargo, en el ámbito del problema genera un resultado criticable. Es necesario entender cómo funciona el algoritmo y sus propiedades para poder seleccionar el adecuado, evitar resultados no esperados y determinar los requerimientos para su uso.

3 Origen del término algoritmo

La palabra "algoritmo" se la suele identificar rápidamente con la raíz griega "arithmos" que significa "números" (de ahí aritmética). Pero, esto es un error común, de hecho, uno bastante antiguo (y que ha dejado huellas). La real academia española mediante una aclaración de la palabra algoritmo dice:

*Quizá del lat. tardío *algobarismus, y este abrev. del ár. clás. ḥisābu lGubār 'cálculo mediante cifras arábigas'.*

Hisab (حساب), significa en árabe cálculo o cuenta. Por otro lado, como nos cuenta Oystein Ore[2], se conoció como "numeración gobar" a una manera decimal basada en el sistema hindú que apareció en España. Su más antigua representación conservada data del año 976 d.e.c. El nombre gobar proviene del árabe ghabar (غبار) que significa polvo. Esto se debe a la costumbre hindú de realizar los cálculos en el suelo o en una tabla polvorienta. Se podría inferir - entonces - que según esta teoría, el nombre provendría del cálculo usando el sistema de numeración antes descrito.

Si bien es una respuesta plausible, existe otra explicación hoy más aceptada.

Tenemos que retroceder un siglo más atrás, IX d.e.c. En esos años el Islam se había convertido en un una potencia que conquistó grandes áreas del antiguo imperio romano. Dentro de sus fronteras quedaron las obras de los sabios griegos y romanos. Comienza

[2]"Number theory and its history", Oystein Ore (Mc Graw Hill 1948)

una tarea de traducción, resignificación y nuevos aportes de origen persas, hindues y chinas. En ese contexto se crean academias promovidas por diferentes califas. En una de ellas fue contratado un matemático llamado Mohamed ben Musa.

Mohamed ben Musa era originario de Khwārizm. Actualmente parte del territorio de República de Uzbekistán, en la provincia de Jiva, está situada en el curso inferior del rio Amū Daryā. En honor a su origen, pasó a autodenominarse como Mohamed ben Musa al-Khwārizmī. Es decir "el de Khwārizm".

La obra de al-Khwārizmī fue muy influyente. Para ejemplificar, basta decir que la palabra "Álgebra" debe su existencia a este matemático. Kitāb al-mukhtasar fī hisāb al-ğabr wa-l-muqābalah (كتاب المختصر في حساب الجبر والمقابلة) es el nombre original de su obra cuya traducción se conoce como "Libro Conciso sobre el Cálculo por Completación y Balance".

Mohamed ben Musa vivió entre el 780 y el 840 aproximadamente. En sus escritos explicaba cómo resolver tipos de problemas matemáticos en forma genérica (y no en casos particulares como era la norma). Además proponía la utilización del sistema de numeración hindú. El mismo era un sistema posicional decimal (y el sistema que utilizamos hoy en día). Se enfrentaba a la norma: utilizar los números romanos. En este sistema las operaciones matemáticas eran más complejas de realizar.

Su libro Kitāb al-ğam' wa-l-tafrīq bi-hisāb al-hind (كتاب الجمع والتفريق بحساب الهند) (Libro de la suma y la división según el cálculo de los hindúes) detalla las ventajas del sistema de cálculo con números hindúes. Tiene un profundo impacto. Es traducido al latín en el siglo XII por Adelardo de Bath, entre otros autores. En varias de las traducciones no le dieron título a la obra. Sin embargo, comenzaban con la frase "Dixit algorizmi" ("como dijo algorizmi"). La traducción más conocida fue titulada "Algoritmi de numero Indorum" ("Algoritmi sobre los números de los hindúes") en 1857 por Baldassarre Boncompagni.

Es en manos de sus traductores que la latinización del gentilicio del autor, pasa a convertirse en la expresión del sistema numérico y los métodos para resolver los problemas algebraicos. Un ejemplo temprano de esto se puede ver en la obra titulada "Carmen de Algorismo" (Poema de algorismo) de 1240 compuesto por Alexandre de Villedieu que comienza de la siguiente forma:

Haec algorismus ars praesens dicitur, in qua / Talibus Indorum fruimur bis quinque figuris.

(La presente técnica llamada Algorismus, en la que se utiliza dos veces cinco las figuras hindúes)

El término algoritmo, que manejamos hoy día para cualquier procedimiento (no solamente matemático), tuvo más caminos por recorrer. En el siglo XVII en Francia sufre un cambio en su forma de escribirse al moderno "algorithm". Seguramente y equivocadamente al relacionarlo al griego "arithmos" (números). De allí es tomado por el idioma Inglés, y luego regresará al español.

4 Algoritmos

El término “algoritmo” en los últimos años está en boca de todos. Vivimos en una sociedad altamente tecnológica. Donde los sistemas informáticos cada vez influyen más en la cotidianidad.

Un “algoritmo” nos puede sugerir el camino a realizar desde nuestro hogar hasta un negocio utilizando un auto, bicicleta o caminando. Puede evitar congestionamientos redirigiendo a los viajeros por rutas alternativas. Un “algoritmo” puede hacer que miremos un medio de comunicación haciendo que su base de lectores pase de pocas centenas a millones de un día para otro. Similarmente puede hacer que un medio exitoso pase al ostracismo. Un algoritmo te puede sugerir qué y dónde comer, qué vestir, qué música escuchar, qué video mirar… incluso con quien salir.

Pero ¿qué es exactamente un algoritmo?

La respuesta corta es “un conjunto de instrucciones que se deben ejecutar en orden para resolver un determinado problema”.

Esa definición no está errada. Pero es insuficiente. La podemos aplicar sin dificultad al concepto de una receta gastronómica. Uno podría circunscribir el tipo de problema a uno de índole matemático. O invocar en la definición el uso de computadoras. La primera restricción excluye varias de las aplicaciones modernas del uso de algoritmos. La segunda, excluye miles de años de historia donde los algoritmos fueron aplicados sin existir remotamente un artilugio tecnológico para ejecutarlo.

Lo que haremos, para robustecer la definición, es agregar 5 características que deben cumplir los algoritmos según Donald Knuth[3]:

1) Finitud: Debe finalizar luego de un número finito de pasos. La ausencia de ésta única característica - siempre según Knuth - hace que hablemos de un método computacional
2) Definitud: Cada paso debe ser definido con precisión. Las acciones a realizar deben ser especificadas sin ambigüedades y en forma rigurosa.
3) Un algoritmo tiene cero o más entradas. Estas corresponden a las cantidades que son introducidas al algoritmo antes de iniciar su ejecución o a medida que se va ejecutando.
4) Un algoritmo tiene una o más salidas. Corresponden a cantidades que se relacionan con las cantidades de la entrada y que se construyen a medida que se ejecutan los pasos del algoritmo.
5) Eficacia: Todas sus operaciones deben ser lo suficientemente básicas que puedan ser realizadas en un tiempo finito por alguien utilizando un lápiz y un papel. Este último ítem puede resultar extraño. El concepto de papel y lápiz tiene raíces anteriores a Knuth y el lector tendrá que tener paciencia para llegar a conocer su origen. Corresponde a una definición bastante informal. Aunque la idea es clara: lo que se debe realizar en un punto debe ser realizable en la práctica en un lapso temporal acotado.

5 Origen del análisis de algoritmos

Charles Babbage es considerado por muchos como el padre de la computación. De nacionalidad británica, vivió entre el 26 de diciembre de 1791 y el 18 de Octubre de 1871. Como matemático y científico de su época dependía de tablas de cálculo para sus trabajos. Tablas elaboradas por personas que por la naturaleza del cálculo solían tener errores a causa del cansancio o distracciones del calculista.

Babbage molesto con los inconvenientes y retrasos causados por los errores de cálculo visualizó la creación de un ingenio mecánico que automatice su construcción. La máquina

[3] "The Art of computer programming. Vol 1", Donald Knuth, 1968

diferencial fue su respuesta a la necesidad de aproximar funciones logarítmicas y trigonométricas mediante polinomios. El 14 de Junio de 1822 la propuso a la Royal Astronomical Society con el título "Nota sobre el uso de maquinaria para el cómputo de tablas matemáticas muy grandes".

A poco de comenzar, su genio lo empujó a encarar un proyecto más ambicioso: La máquina analítica. Dicho artilugio sería programable y podría resolver cualquier operación matemática. Utilizaría base decimal, tarjetas perforadas y se operaría mediante motor a vapor. Entre 1837 y el final de su vida Babbage trabajó en ella. Lamentablemente por carencias tecnológicas de la época y problemas de presupuesto nunca la pudo construir (más de 100 años después se tuvo que esperar para la construcción de una máquina funcional siguiendo las indicaciones dejadas por Charles).

Babbage en su libro "Passages from the Life of a Philosopher"[4] dedica el capítulo 8 "Of the analytical machine" a narrar diferentes situaciones alrededor de su intento de construcción. Entre ellos una charla en un desayuno con el Profesor MacCullagh de Dublin. Allí afirma:

> *"As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise—By what course of calculation can these results be arrived at by the machine in the shortest time?"*

Existió una persona contemporánea que quedó atrapada por la propuesta de Babbage. Hablamos de quien la historia considera el primer programador. Nos referimos a Ada Lovelace

Augusta Ada King, Condesa de Lovelace (tal su nombre completo), hija del poeta Lord Byron, fue una mujer apasionada por las matemáticas en una época donde por su condición de tal las ciencias le eran vedadas.

Fascinada por este invento realiza la traducción de un informe de Luigi Federico Menabrea, que es publicado en la revista Scientific Memoirs con el nombre de "Sketch of the analytical engine invented by Charles Babbage"[5]. Dentro de esta publicación agrega extensas notas,

[4] Passages from the Life of a Philosopher, Charles Babbage, 1864
[5] "Sketch of the Analytical Engine invented by Charles Babbage... with notes by the translator", Menabrea, Luigi Federico; Lovelace, Ada, 1843

entre ellas un algoritmo escrito en el lenguaje de la máquina analítica que generaba los números de Bernoulli (El primer programa de la historia!).

Ada en su análisis y notas de 1843 sobre la máquina analítica llegó a una conclusión aún más detallada que Babbage:

> *"In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selection amongst them for the purposes of a Calculating Engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation"*

La construcción de computadoras multipropósitos transformaría sustancialmente el curso de la ciencia. La eficiencia en el cálculo sería crucial. El estudio de cómo resolver un problema debía contener la minimización de los pasos necesarios para lograrlo. Diferentes algoritmos para un mismo problema se debían comparar para determinar cual es más eficiente. Incluso la comparación entre problemas y su complejidad (medido en mínimos pasos para realizarse) es posible. Sin saberlo y como al pasar profetizaron una rama hoy de gran importancia y vitalidad: El análisis de los algoritmos y su complejidad.

7 Análisis de Algoritmos

Cuando analizamos un algoritmo podemos enfocarnos en diferentes puntos. Ya mencionamos anteriormente el concepto de optimalidad. Otro punto a tener en cuenta es su complejidad.

El análisis de la complejidad algorítmica, tal su definición moderna se basa en el estudio de los recursos requeridos para resolver cierto problema utilizando un determinado algoritmo. Como recursos se tienen en cuenta 2 dimensiones: el tiempo de ejecución y el espacio de almacenamiento requerido.

Trabajaremos con un ejemplo sencillo conocido para que se pueda entender el funcionamiento. El problema de ordenamiento de elementos:

Ordenamiento de elementos

Contamos con un listado de números enteros. Deseamos retornar el mismo ordenado de menor a mayor.

Ejemplo: Partimos de la siguiente secuencia de elementos: 8,5,3

Utilizaremos el algoritmo conocido como “ordenamiento por selección” (Selection Sort) para resolverlo. El pseudocódigo de este algoritmos es el siguiente (en rigor se podría hacer un poco más eficiente, pero a efectos de análisis resulta más sencillo así):

Selection Sort

```
Desde el elemento en la primera posición hasta el último
        Sea elemActual el elemento en el que estoy posicionado
        Sea menor = elemActual
        Desde la posición actual hasta el último elemento
                Sea elemComparacion el elemento en el que estoy posicionado
                Si elemComparacion < menor
                        menor = elemComparacion
       Si elemActual > menor
                Intercambiar elemActual por menor
```

Podemos descomponer cada algoritmo en una serie de operaciones elementales. Cada una de ellas tendrá una duración en tiempo determinado. El tiempo de ejecución de cada una de estas puede variar de acuerdo a la máquina seleccionada para realizar el cómputo.

Llamemos Cij al tiempo de ejecución de la operación “i” utilizando la máquina “j”.

Podemos calcular el tiempo total de ejecución de un determinado algoritmo sumando el tiempo de ejecución de cada operación multiplicado por la cantidad de veces se ejecuta. Esto no va a permitir calcular de forma detallada el tiempo total de ejecución.

En nuestro ejemplo podemos encontrar las operaciones de comparación, intercambio, asignación, entre otras. Cada uno tendrá su tiempo de ejecución correspondiente. Vemos que con 3 elementos realizaremos: 3 asignaciones de “elemActual”. 6 asignaciones de “elemComparacion”. 6 comparaciones entre “elemComparacion” y “menor”. 3 comparaciones entre “elemActual” y “menor”. 3 intercambios de “elem Actual” por “menor”. Sumamos todos esos tiempos y obtenemos un tiempo total.

$Tiempo = 3C_{asig} + 6C_{asig} + 6C_{comp} + 3C_{comp} + 3C_{inter}$

Sin embargo, tenemos varios problemas siguiendo este modelo. En primer lugar, para algoritmos medianamente complejos realizar este cálculo a detalle puede ser complejo. En segundo lugar, se complica poder comparar algoritmos diferentes en diferentes máquinas. En tercer lugar, no se está teniendo en cuenta que el algoritmo tiene una entrada y que los tiempos pueden depender de ella (por ejemplo no es lo mismo ordenar quinientos elementos de mayor a menor que ordenar un millón de ellos). Queremos un método genérico que nos permita abstraernos de una máquina elegida y de un tipo de entrada particular del algoritmo

Existen diferentes aproximaciones sobre cómo proceder. Utilizaremos tal vez el más extendido y que sirve a propósitos generales para tener una noción del comportamiento de un algoritmo y su comparación con otros

Primera medida: intentaremos medir el tiempo del algoritmo en función a uno o más parámetros de la entrada del mismo. Si trabajamos con una lista llamaremos "n" a la cantidad de elementos. Si trabajamos con un grafo lo podemos caracterizar en función de sus "n" vértices y "m" aristas. El objetivo es encontrar cómo los tiempos de ejecución crecen en función de estos parámetros de entrada. De esa forma podremos comparar tiempos de algoritmos ante casos similares.

Dentro de nuestro ejemplo podemos ver que tenemos "n=3" elementos a ordenar. Realizaremos "n" asignaciones de "elemActual". n^2 asignaciones de "elemComparacion". n^2 comparaciones entre "elemComparacion" y "menor". n comparaciones entre "elemActual" y "menor". n intercambios de "elemActual" por "menor". Ahora en nuestra fórmula diremos que los tiempos son en función del tamaño "n" de la entrada del algoritmo

$Tiempo(n) = n.C_{asig} + n^2.C_{asig} + n^2.C_{comp} + n.C_{comp} + n.C_{inter}$

Segunda medida: simplificamos los cálculos tomando en cuenta que todas las operaciones tienen un tiempo de ejecución constante igual al tiempo de la operación elemental más costosa. De esa forma nos dará lo mismo si ejecuto un intercambio o una comparación. Obviamente se debe tener cuidado de no considerar operaciones no elementales como tales. Ejemplo: La instrucción "Determinar si el elemento es primo" no podría ser

considerada elemental. Su tiempo de ejecución será en muchos márgenes mayor a una comparación. Y aún más: depende del elemento en sí cual puede ser el tiempo de ejecución de esta operación (no es lo mismo determinar si el número 6 es primo que el si lo es el 999953.

Dentro del ejemplo trabajado vemos que tenemos diferentes operaciones. Todas ellas se pueden considerar elementales. Tomamos aquella cuyo tiempo de ejecución es mayor y lo llamamos C_{op_elem}*. Reemplazamos en nuestro cálculo:*

$$Tiempo(n) \leq n.C_{op\_elem} + n^2.C_{op\_elem} + n^2.C_{op\_elem} + n.C_{op\_elem} + n.C_{op\_elem}$$
$$\leq 3.n.C_{op\_elem} + 2.n^2.C_{op\_elem}$$

Tercera medida: Un mismo algoritmo ante una misma cantidad de elementos de entrada podría comportarse de forma diferente. Parte del código podría no ejecutarse bajo ciertas circunstancias. En casos extremos un mismo algoritmo para una misma cantidad de elementos podría comportarse de forma diferente. En ese panorama podemos buscar el peor de los casos, el mejor de los casos o un valor promedio. Por regla general el peor caso es el que suele utilizarse para comparar entre algoritmos. Esta elección podría ser objetada. ¿Por qué utilizar un caso muy malo entre millones superiores? La respuesta es práctica: instalaría un programa por ejemplo para controlar un automóvil autónomo que en promedio funcionó muy bien, pero que tiene ciertos casos (o al menos 1 caso) que bajo ciertas circunstancias no llega a reaccionar a tiempo y puede ocasionar una colisión?

Dentro del ejemplo trabajado trabajamos con 3 elementos a ordenar: 8,5,3. Está ordenada en el sentido opuesto al deseado, por ese motivo en cada iteración realizada deberemos realizar el intercambio de "elemActual" por "menor". Sin embargo si los mismos elementos estuviesen ya ordenados con el criterio buscado (3,5,8), esos intercambios no se ejecutarán. Podemos dejar de contabilizar los intercambios en la fórmula y como consecuencia se reduce una constante en el término asociada a n.

$$Tiempo(n) \leq n.C_{asig} + n^2.C_{asig} + n^2.C_{comp} + n.C_{comp} + \cancel{n.C_{inter}}$$
$$\leq n.C_{op\_elem} + n^2.C_{op\_elem} + n^2.C_{op\_elem} + n.C_{op\_elem} + \cancel{n.C_{op\_elem}}$$
$$\leq 2.n.C_{op\_elem} + 2.n^2.C_{op\_elem}$$

Se pueden analizar diferentes ordenamientos y valores de los n elementos y se verá que el primer caso presentado corresponde al peor de los casos, el segundo corresponde al mejor de

los casos. Realizando un análisis estadístico se podrá obtener el caso promedio. Nosotros nos quedaremos con el peor de los casos para el resto del análisis

Cuarta medida: A medida que crece el tamaño de la entrada del algoritmo analizado podremos ver que ciertos términos en nuestra ecuación de tiempo de ejecución van perdiendo peso en el tiempo total. En general cuando analizamos los complejidad de un algoritmo se intenta ver su funcionamiento para tamaño de entradas grandes. Siguiendo ese criterio aquellos términos pequeños se simplifican. Nos quedamos con aquel mas representativo para cada variable de entrada analizada.

Dentro del ejemplo trabajado trabajamos vemos que tenemos un termino lineal y otro cuadrático. Nos quedaremos con este último

$$Tiempo(n) \leq \cancel{2.n.C_{op\_elem}} + 2.n^2.C_{op\_elem}$$
$$\leq 2.n^2.C_{op\_elem}$$

Quinta medida: Llegado a este punto vemos que los tiempos calculados son "maquina-dependiente". La operación elemental más costosa por máquina no es un parámetro fiable para comparar. Para evitar esto y permitir las comparaciones llegamos a un último punto donde a tamaño suficientemente grande de la entrada al algoritmo estas constantes se vuelven también poco influyentes en los tiempos totales calculados. Podemos reemplazarlas por una constante general "C". De esa forma llegamos a una representación compacta y falta de analizar. Implementaremos finalmente la notación asintótica también conocida como notación de Landau

En nuestro ejemplo podemos llevar a que la complejidad temporal de nuestro algoritmo para un tamaño de entrada "n" suficientemente grande estará acotada por una constante multiplicando a n^2. El símbolo se lee "O" (aunque la letra corresponde a la griega Omicron)

$Tiempo(n) \leq c.n^2$

$Tiempo(n) \in O(n^2)$El mismo tratamiento podemos realizar para analizar la complejidad espacial. Aclarando que tenemos en cuenta el espacio extra utilizado para la resolución del problema. Espacio adicional al correspondiente al de la entrada del problema.

En nuestro problema podríamos ir modificando el orden de los elementos en la misma estructura en que los recibimos (sería un ordenamiento "inline"). En ese caso solo requerimos el espacio adicional para el elemento menor, elemento actual y el elemento a comparar (incluso menos dependiendo la implementación). Estos elementos no crecen ni se modifican a medida que aumenta el tamaño "n" de la entrada. Por lo tanto requerimos un espacio "c" constante en función de "n". Lo representamos como:

$Espacio(n) \leq c.1$
$Espacio(n) \in O(1)$

Con esta metodología conseguimos una forma simplificada para comparar diferentes algoritmos fácil de calcular y sin depender de la tecnología que seleccionamos para implementarla. Para un problema determinado podemos idear diferentes formas de solucionarlo. Comparar entre esas soluciones sus complejidades temporales y espaciales nos ayuda a tomar decisiones inteligentes de cual utilizar y en qué caso. Incluso el descubrimiento de nuevas soluciones las podemos analizar a la luz de las existentes.

8 Notación Asintótica

Mediante la notación asintótica determinamos cómo caracterizar un algoritmo en dos dimensiones: temporal y espacial. Corresponde a una función que toma como parámetro uno o más indicadores del tamaño de la instancia y mide el crecimiento del tiempo de ejecución o tamaño extra de almacenamiento en función de estos. Estos parámetros toman valores enteros mayores o iguales a cero.

No corresponde a una medida exacta. Es una cota asintótica que acompaña al crecimiento. Esa cota puede ser inferior, superior o ambas. Depende de las circunstancias o posibilidades utilizaremos una u otra.

Para efectos de las siguientes explicaciones imaginemos un determinado algoritmo que se comporta en cuanto a su tiempo de ejecución como una función f(n) donde n corresponde al tamaño de la instancia.

Si logramos una que a partir de un valor n_0 una segunda función g(n) multiplicada por una constante "c" positiva sea mayor o igual a f(n) para toda $n \geq n_0$:

$$f(n) \leq c.g(n) \ , \ \forall \ n \geq n_0$$

Entonces g(n) acota superiormente a f(n) y diremos que

$$f(n) \in \mathbf{O}(g(n))$$

El símbolo corresponde a la letra griega Omicron. Se la pronuncia como la vocal latina mayúscula "O" y por ella recibe el nombre de "big-o".

Si logramos una que a partir de un valor n_0 una segunda función g(n) multiplicada por una constante "c" positiva sea menor o igual a f(n) para toda $n \geq n_0$:

$$f(n) \geq c.g(n) \ , \ \forall \ n \geq n_0$$

Entonces g(n) acota superiormente a f(n) y diremos que

$$f(n) \in \mathbf{\Omega}(g(n))$$

El símbolo corresponde a la letra griega Omega. Se la expresa como "big-omega".

Si logramos una que a partir de un valor n_0 una segunda función g(n) multiplicada por una constante "c_1" positiva sea menor o igual a f(n) para toda $n \geq n_0$ y que para una segunda constante "c_2" positiva sea mayor o igual a f(n) para toda $n \geq n_0$:

$$c_1.g(n) \geq f(n) \geq c_2.g(n) \ , \ \forall \ n \geq n_0$$

Entonces g(n) acota superior e inferiormente a f(n) y diremos que

$$f(n) \in \mathbf{\Theta}(g(n))$$

El símbolo corresponde a la letra griega Theta. Se la expresa como "big-theta".

Para cualquier de los tres casos se debe tener en cuenta que para n menores a n_0 esta cota no necesariamente se cumple para un valor determinado de n. Sin embargo lo que nos interesa es un comportamiento general para valores del problema lo suficientemente grandes.

¿Cuándo utilizamos cada uno de ellos? Por regla general nos interesa conocer cómo se comporta un determinado algoritmo en el peor de los casos para un tamaño determinado de "n" utilizaremos $\mathbf{O}$. Si nos interesa conocer el mejor de los casos utilizaremos $\mathbf{\Omega}$. Si por el otro lado tenemos en cuenta el caso promedio utilizaremos $\mathbf{\Theta}$.

Algunas propiedades son prácticas a la hora de trabajar con esta notación. entre ellas

Reflexividad:

$f(n) \in \mathbf{O}(f(n))$

$f(n) \in \mathbf{\Omega}(f(n))$

$f(n) \in \mathbf{\Theta}(f(n))$

Transitividad:

Si $f(n) \in \mathbf{O}(g(n))$ y $g(n) \in \mathbf{O}(h(n)) \Rightarrow f(n) \in \mathbf{O}(h(n))$

Si $f(n) \in \mathbf{\Omega}(g(n))$ y $g(n) \in \mathbf{\Omega}(h(n)) \Rightarrow f(n) \in \mathbf{\Omega}(h(n))$

Si $f(n) \in \mathbf{\Theta}(g(n))$ y $g(n) \in \mathbf{\Theta}(h(n)) \Rightarrow f(n) \in \mathbf{\Theta}(h(n))$

Cuando trabajemos con algoritmos buscaremos describirlos mediante notación asintótica con funciones que lo acoten lo más posible. De esa forma la caracterización es ajustada y sirve para compararla con otros algoritmos. Generalmente se suelen utilizar un conjunto de familias de funciones. Las más corrientes: 1, log n , n, n log n, n^2, n^3, 2^n, n!

En diversa bibliografía y publicaciones se usa indistintamente el "=" en reemplazo al "$\in$". Por ejemplo "$f(n) = \mathbf{\Theta}(g(n))$". El significado es el mismo, pero esta última puede ser calificada como un abuso de notación. Dado que la igualdad no es en ambos sentidos.

En algunos algoritmos pueden existir más de un parámetro que determine el tamaño de la entrada. El ejemplo más común es si la entrada es un grafo. En ese caso tendremos habitualmente dos parámetros a tener en cuenta: la cantidad de vértices y la cantidad de aristas. La misma notación se puede utilizar sin problemas, indicando por ejemplo "$f(n,m) \in \mathbf{\Theta}(g(n,m))$". Existen publicaciones[6] que demuestran que con ciertos cuidados la notación asintótica se puede usar sin problemas. La utilizaremos cuando sea necesaria.

9 Algoritmo buenos y problemas manejables

[6] On Asymptotic Notation with Multiple Variables, Rodney R. Howell, 2008

Cuando trabajamos en un problema e investigamos la forma de resolverlo podemos hallar diferentes algoritmos para resolverlos. A la hora de utilizarlos debemos seleccionar uno de ellos. ¿Cuál elegir? La respuesta habitual suele ser "el mejor entre de ellos". ¿Cómo determinamos cual es el mejor conocido? ¿Es este último suficientemente "bueno"?

En 1965, Jack Edmonds en una digresión dentro de un hoy importante artículo[7], da una respuesta a esta última pregunta. Considera que un algoritmo es "bueno" (también nombrado como eficiente) si su comportamiento en tiempo es proporcional al tamaño de la entrada. Ahondado en la idea vemos que se refiera que su comportamiento se puede caracterizar por estar acotada por una función asintótica de tipo polinómica $\mathbf{O}(n^c)$. En el mismo año y en otro artículo[8] Alan Belmont Cobham llega a una idea similar.

Se conoce como **Tesis Cobham–Edmonds** a aquella que afirma que un problema es manejable o tratable ("Tractable" en su expresión original en inglés) si existe un algoritmo "bueno" que lo soluciona. ¿Implica esto que vamos a poder resolver en un tiempo razonable el problema cualquiera sea el tamaño del mismo? No necesariamente. Lo que se espera teniendo un algoritmo bueno es que el avance de la tecnología eventualmente permita resolver problemas de un tamaño dado. Pongamos un ejemplo que pueda visualizar este concepto.

La supercomputadora más rápida en el año 2020[9] fue "Fugaku", construida por Fujitsu en Japón. Alcanzaba los 442,01 PFLOPS (Peta Floating Point Operations Per Second, equivalente a $442{,}01 \times 10^{15}$ FLOPS). FLOPS es la unidad estándar que se utiliza para comparar supercomputadoras que deben realizar cálculos científicos utilizando números de alta precisión. Solo diez años antes la más veloz fue Tianhe-1A de origen Chino con 2,57 PFLOPS. Es decir que en 10 años el poder de procesamiento aumentó un 17.198,83%. En el cuadro se pueden ver para un caso hipotético de algoritmos que en 2010 podrían resolver como máximo un problema de tamaño n=100, en un tiempo razonable (algún valor seleccionado arbitrariamente). Luego basándose en el incremento de poder de procesamiento cual seria el tamaño máximo que puede resolver en 2020 en el mismo tiempo.

[7] "Paths, trees, and flowers", Jack Edmonds, 1965.
[8] "The intrinsic computational difficulty of functions", Alan Cobham, 1965
[9] "Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D". TOP500 Supercomputer Sites. https://www.top500.org/system/179807/

	O(log n)	**O**(n)	**O**(n log n)	**O**(n^2)	**O**(n^3)	**O**(2^n)	**O**(n!)
2010	100	100	100	100	100	100	100
2020	100^{171}	17.198	8.728	1.311	556	107	101

Se puede observar que a mayor complejidad menor el incremento del problema máximo a resolver. Lo que queda claro es que para los algoritmos no polinomiales (no "buenos") el incremento es casi marginal (7 y 1 respectivamente). Se debe aclarar que estos valores son ilustrativos y no pertenecen a ningún algoritmo en particular. Y que el escenario fue absurdamente simplificado para ejemplificar el punto deseado.

Ante la existencia de varios algoritmos para resolver un mismo problema cuál elegir? Pueden existir varios "buenos". Comparamos sus complejidades temporales y espaciales. Estaremos inclinados a seleccionar aquel que tenga menor complejidades. Tal vez un algoritmo es mejor espacialmente pero peor temporalmente o viceversa con respecto a su competidor. Consideraremos entonces qué factor prima más. También es importante conocer la naturaleza particular del problema que debemos resolver. Conocer si las instancias con las que nos enfrentaremos tienen cierta condición que inclinan la balanza frente a cierto algoritmo u otro.

Por último y no por eso para despreciar, se debe tener en cuenta que la notación utilizada puede jugarnos una mala pasada en ocasiones. Recordemos que la notación asintótica afirma que para un valor del tamaño del problema en adelante se cumple con la complejidad dada. ¿Qué pasa si el tamaño de nuestro problema es menor? Sobre este punto se detuvieron Richard Lipton y Ken Regan. Ellos acuñaron la clasificación informal "Galácticos" para ciertos tipos de algoritmos.

> *some algorithms are wonderful, so wonderful that their discovery is hailed as a major achievement. Yet these algorithms are never used and many never will be used—at least not on terrestrial data sets.*[10]

Concretamente que el tamaño del problema tiene que ser monstruosamente grande para que los beneficios de utilizar el algoritmo sea palpable. Es decir para que cumpla con la complejidad que promete. Se requiere que el problema tenga el tamaño de una galaxia para que esto se cumpla. Un ejemplo de este tipo de algoritmos corresponde al propuesto

[10] "People, Problems, and Proofs: Essays from Gödel's Lost Letter:2010", Lipton, Richard J; Kenneth W. Regan, 2013, pp. 109–112.

para multiplicar números enteros por David Harvey y Joris van der Hoeven[11]. Su propuesta tiene una complejidad temporal $\mathbf{O}(n \log n)$ comparado con la complejidad de nuestra multiplicación tradicional que es $\mathbf{O}(n^2)$. Sin embargo recién para números con una cantidad de dígitos mayor a $10^{10^{38}}$ se logra esta cota. Este número es mayor a la cantidad de átomos del universo conocido y por lo tanto el algoritmo no se usa en la práctica. Pero fue un hito importante al demostrar que esa cota de complejidad es posible.

Siendo menos dramáticos, se debe considerar que tamaño de entrada enfrentará el algoritmo utilizado. Diferentes, que resuelven el mismo problemas serán los más adecuados de acuerdo a esta información.

10 Caso práctico: El problema de los casamientos estables.

Se conoce como el problema del emparejamiento estable (stable matching problem) al ejercicio de construir grupos entre elementos de acuerdos a ciertas preferencias de emparejamiento entre cada uno de los elementos. Se espera que una vez armados estos grupos no exista posibilidad de ruptura de los mismos por miembros que prefieran estar en otro grupo y que en el otro grupo lo prefieran a algún integrante existente. Existe una amplia cantidad de variantes. Cada uno de ellos para situaciones particulares. Tal vez el más conocido y no carente de cierta controversia es el problema de los casamientos estables (stable marriage problem).

El problema de los casamientos estables fue caracterizado formalmente y analizado por Gale y Shapley[12]. El problema es un caso simplificado de generación de parejas estables. Tiene el siguiente enunciado:

Problema de los casamientos estables
Sean un conjunto de “n” hombres y “n” mujeres. Cada uno de ellos con una lista de preferencias completas y sin empates ordenadas de sus contrapartes. Se desean conformar parejas entre ellos de forma que nadie se quede sin pareja y que las parejas armadas sean estables en el tiempo.

[11] “Integer multiplication in time O(n log n)”, David Harvey y Joris van der Hoeven, 2020
[12] “College Admissions and the Stability of Marriage”, David Gale; Lloyd Shapley, 1962.

Cuando se indica que es completa significa que cada persona tiene en sus preferencias a todos las personas del otro grupo. El concepto de “sin empates” determina que no hay en una lista de preferencias dos o más personas con las que “le de lo mismo” formar pareja.

Veamos el ejemplo que utilizaremos en las siguientes secciones.

Tenemos 4 hombres y 4 mujeres. En las siguientes tablas se muestran las preferencias de cada uno de ellos. La primera corresponde a la preferencia de cada hombre de mayor a menor de las mujeres. La segunda es el equivalente para las mujeres y su preferencia de los hombres

H	*Preferencias*
1	*1,2,3,4*
2	*1,3,4,2*
3	*2,4,3,1*
4	*4,2,1,3*

M	*Preferencias*
1	*2,3,4,1*
2	*4,3,1,2*
3	*1,2,3,4*
4	*3,4,2,1*

Para continuar requerimos realizar ciertas definiciones.

Def. 1: “S” es un **matching** si está formado por un conjunto de parejas (h,m) donde los hombres y mujeres aparecen como mucho en una de ellas. Un hombre o mujer podría no estar en el matching.

El siguiente conjunto corresponde a un matching de nuestro ejemplo: S_1={(1,3), (2,4)}. Sin embargo S_2 = {(1,2), (3,2)} no es matching. La mujer 2 aparece en más de una pareja.

Def. 2: “S” es un **matching perfecto** si está formado por un conjunto de parejas (h,m) donde cada hombre y mujer aparece en una (ni más ni menos) pareja.

Se desprende que dado que tenemos igual cantidad de hombres y mujeres no hay nadie sin emparejamiento.

El siguiente conjunto corresponde a un matching perfecto de nuestro ejemplo: S={(1,3), (2,4), (3,1), (4,2)}.

Def 3: Una pareja (h_1,m_1) en un matching es inestable si existe otra pareja (h_2,m_2) donde h_1 prefiere a m_2 antes que a m_1 y además m_2 prefiere a h_1 en lugar que a h_2.

Vemos que en el matching S={(1,3), (2,4), (3,1), (4,2)} la pareja (2,4) es inestable. Vemos esta inestabilidad con la pareja (3,1). El hombre 2 tiene antes en su lista de preferencias a la mujer 1 que a su pareja actual la mujer 4. Al mismo tiempo la mujer 1 tiene antes al hombre 2 que a su actual pareja 4. Por lo tanto, corresponde a una inestabilidad.

Def 4: "S" es un **matching estable** si es perfecto y no tiene parejas inestables.

Si nos entregan un matching y deseamos comprobar que es estable tendremos que por cada pareja comprobar con el resto de ellas sí conforman una inestabilidad. Para eso se debe consultar a cada individuo que la conforma su lista de preferencias. Se debe verificar también que todos los individuos estén en una y solo una pareja.

Pueden existir más de un matching estable por instancia del problema.

Vemos que el matching S_1*={(1, 3), (2,1), (3, 2), (4, 4)} corresponde a un matching estable. Vemos que todos los individuos están en una pareja. Falta verificar si hay parejas con inestabilidad. El hombre 1 rompería su pareja de tener oportunidad por las mujeres 1 o 2 (pero no por la mujer 4 que inferior su preferencia comparada a su actual). Pero ninguna de las dos mujeres rompería con sus parejas actuales. La mujer 1 está en pareja con el hombre 2 que es a quien más prefiere. La mujer 2 está con el hombre 3 que prefiere al hombre 1. De igual manera se debería verificar con el resto de las parejas.*

Además podemos demostrar por el mismo mecanismo que el matching S_2*={(1,3), (2,1), (3,4), (4, 2)} corresponde a un matching estable*

Con las definiciones anteriores podemos reformular el problema cómo "encontrar un matching estable para los conjuntos de "n" hombres y mujeres con sus preferencias". En las siguientes secciones presentaremos un algoritmo bueno y demostraremos la existencia de una solución siempre. Por último trabajaremos sobre las características de la solución encontrada

11 Matrimonio estable: Algoritmo de aceptación diferida

Los autores del enunciado del problema del casamiento estable, también dieron el primer algoritmo para resolverlo. El mismo es conocido como “Algoritmo de aceptación diferida” (deferred acceptance algorithm) y también por el apellido de los creadores “Gale-Shapley”

El algoritmo asigna un rol diferente a cada uno de los conjuntos. Los hombres son quienes proponen armar una pareja (“solicitantes”). Por otro lado, las mujeres tendrán un papel pasivo y esperarán propuestas (“requeridos”). Ellas aceptarán formar pareja en dos casos. Si hasta el momento no tienen compañero o si prefieren más a quien le propone de su contraparte actual (teniendo en cuenta su lista). Si no se cumplen las condiciones el hombre es rechazado.

Es un algoritmo iterativo. Inicia con un matching vacío. En cada “ronda” un hombre que aún no tiene pareja le solicita a una mujer formarla. Lo hará siguiendo su orden de prioridad: desde la mujer que prefieran más hasta la que menos. Los hombres pueden estar sin pareja por dos motivos: si nunca fueron aceptados o si perdieron su pareja ante el rompimiento del vínculo por parte de la mujer. Finaliza el proceso cuando no queden más hombres sin pareja.

Veamos el pseudocódigo de la solución. En vez de hombres y mujeres hablaremos de “**solicitantes**” y “**requeridos**” (más adelante veremos el por qué este cambio de nomenclatura)

Algoritmo de aceptación diferida (Gale-Shapley)

```
Inicialmente M=∅     // Matching
Mientras exista un solicitante sin pareja que no aun no se haya postulado a todas sus posibles parejas
        Sea s un solicitante sin pareja
        Sea r el requerido de su mayor preferencia al que no le solicitó previamente
                Si r está desocupado
                        M = M ∪ (s,r)
                        s está ocupado
                Sino
                        Sea s' tal que (s',r) ∈ M
                        Si r prefiere a s sobre s'
                                M = M – {(s',r)} ∪ (s,r)
                                s está ocupado
```

s’ está libre
Retornar M

Iniciamos con ninguna pareja armada.
Ronda 1: Seleccionamos un solicitante sin pareja: hombre 1. Este le solicita a la primera en su lista: mujer 1. Como ella no está en pareja acepta. Parejas hasta el momento M={(1,1)}
Ronda 2: Seleccionamos un solicitante sin pareja: el hombre 2. Ese le solicita a la mujer 1. Ella está en pareja con el hombre 1. Pero rompe la pareja puesto que prefiere al nuevo solicitante. Parejas hasta el momento M={(2,1)}
Ronda 3: Seleccionamos: hombre 3. Este le solicita a mujer 2. Ella acepta por que no tiene pareja. Parejas M={(2,1), (3,2)}
Ronda 4: Seleccionamos: hombre 4. Le solicita a mujer 4 y acepta. Parejas M={(2,1), (3,2), (4,4)}
Ronda 5: Seleccionamos: hombre 1. Le solicita a mujer 2 y lo rechaza.
Ronda 6: Seleccionamos: hombre 1. Le solicita a mujer 3 y acepta. Parejas M={(1,3), (2,1), (3,2), (4,4)}. Termina la ejecución con todos en pareja.

Algunas cosas que se pueden notar a primera vista del algoritmo incluyen la ausencia de un criterio de selección del próximo solicitante a intervenir en la ronda. Esto nos puede hacer preguntarnos si el orden de las selecciones determina el matching final encontrado. Por otro lado, terminamos encontrando un matching y podes ver que es estable. Pero no sabemos si siempre termina con un matching estable. Pero incluso, ¿puede pasar que una instancia no tenga un matching estable posible? Trabajaremos sobre esta cuestión en la próxima sección

12 Gale-Shapley: Análisis de optimalidad

Cuando iniciamos con la ejecución de nuestro algoritmo no hay parejas armadas. Y en cada ronda la cantidad de parejas puede aumentar en uno o mantenerse. El primer caso es cuando el requerido no tenía pareja. El segundo caso es cuando se rechaza al solicitante o cuando se rompe una pareja existente para armarse una nueva. Por lo tanto, la cantidad de parejas durante la ejecución del algoritmo es monótonamente creciente.

Por la naturaleza del algoritmo, un requerido que se empareja en cierto momento no estará sin pareja hasta finalizar la ejecución. Esto es porque el solicitante una vez en pareja no la romperá por voluntad propia. Solo ante el ofrecimiento de conformación de pareja de

otro solicitante puede romper una pareja existente. Pero esto ocurre solo si el requerido lo prefiere y generando una nueva pareja junto al nuevo solicitante.

Partiendo de la característica anterior se puede afirmar que un requerido a partir que tiene una pareja únicamente podrá mejorar en sus próximas teniendo en cuenta su lista de preferencias. Similarmente un solicitante que se encuentra en pareja solo podrá empeorar en sus nuevas parejas (siempre hablando según sus preferencias). Pasando además por momentos sin emparejamiento.

¿Cuándo termina de ejecutarse el algoritmo? Hay dos condiciones de finalización. La primera corresponde a la ausencia de solicitantes sin parejas. En el mejor de los casos en el primero de los pedidos de todos los solicitantes consiguen pareja y en ninguno de esos pedidos se rompe una existente. Por lo tanto tenemos un mínimo de "n" rondas. La segunda condición corresponde al peor de los casos. Todos los solicitantes agotan sus listas de preferencias. Tenemos "n" solicitantes y cada uno de ellos le tuvo que consultar a "n" requeridos. Por lo tanto tendremos "n^2" rondas.

¿Pueden agotarse todos los pedidos de emparejamientos posibles y quedar solicitantes sin parejas? Imaginemos por un momento que esto ocurre. Nos centramos en un solicitante que no tiene pareja. Le tuvo que preguntar a todos los requeridos en el orden de su lista de preferencias. Todos los tuvieron que rechazar. ¿Cuándo lo rechaza un requerido? Cuando ya tienen una pareja mejor o cuando luego de aceptarlo lo deja por un mejor candidato. "n" requeridos lo tuvieron que rechazar y por lo tanto estar en pareja. En ese caso están en pareja con "n" solicitantes. Pero esto es absurdo por que la cantidad de solicitantes es "n" y si uno está sin pareja como mucho n-1 pueden estarlo. En definitiva, al terminar el algoritmo no pueden quedar solicitantes sin parejas.

Resta comprobar si existen inestabilidades en el matching conseguido. Tal como en el punto anterior intentaremos mostrar por el absurdo que esto no es posible. Supongamos que existe una inestabilidad en el matching "M" conseguido. Eso implica que hay un par de parejas solicitantes-requeridos (s_1,r_1) y (s_2,r_2) en donde s_1 y r_2 se prefieren entre ellos que a sus respectivas parejas. Si eso ocurre significa que en la lista de preferencias de s_1 se encuentra r_2 antes que r_1. Debido a cómo funciona el algoritmo, s_1 le propone formar pareja a r_2 antes que a r_1. Pudo ocurrir que r_1 lo rechazó en ese momento porque estaba con alguien al que prefería más. La alternativa es que conformó pareja con él y más

adelante lo rechazó por alguien al que prefiere más. Como vimos anteriormente los requerido sólo pueden mejorar en sus parejas a medida que se ejecuta. Pero sin embargo quedó emparejado con s_2 al que prefiere menos que s_1. Lo que siguiendo el algoritmo no es posible y por lo tanto es absurda la hipótesis inicial.

Sabemos que al finalizar siempre tenemos un matching perfecto y además que el matching conseguido no tiene inestabilidades. Por lo tanto podemos afirmar que el algoritmo de Gale-Shapley soluciona cualquier instancia del problema del casamiento perfecto de forma óptima. Pero aún hay más: podemos estar seguros que siempre existe un matching estable.

13 Gale-Shapley: Análisis de complejidad

Nos centraremos en la complejidad temporal y espacial del algoritmo de aceptación diferida. Para eso nos enfocaremos en el pseudocódigo presentado anteriormente. En el primer análisis pudimos ver que la iteración que conforma todas las rondas está acotada en el mejor de los casos por "n" y en el peor de los casos por "n^2" donde n es el tamaño del conjunto de los requeridos y de los solicitados. Pero aún no podemos afirmar categóricamente la complejidad final. Para eso tenemos que seleccionar las estructuras de datos a utilizar y recién en base a la complejidad de sus operaciones y al espacio de requerido podremos completar el análisis. Intentaremos seleccionar aquellas estructuras que hagan que las complejidades sean lo menor posible. En ocasiones tendremos alternativas que mejoren alguna dimensión de la complejidad en detrimento de la otra.

Debemos seleccionar una estructura para los solicitantes sin pareja. Proponemos utilizar una pila o una cola. Ambas tendrán un proceso inicial de carga $\mathbf{O}(n)$ de todos los solicitantes que inicialmente están sin parejas. Luego tiene una complejidad de $\mathbf{O}(1)$ tanto para obtener el próximo elemento (un solicitante que aún no tiene pareja) como para agregar uno nuevo (solicitante que se quedó sin pareja). La complejidad espacial es $\mathbf{O}(n)$ dado que comienza con todos los solicitantes y va reduciéndose en el tiempo.

Debemos definir una estructura para mantener las preferencias de cada solicitante. Tiene que ser una estructura que nos permite recorrer de forma eficiente las preferencias en forma descendente. Podemos utilizar una matriz donde cada fila i corresponde a un solicitante y en sus celdas tiene el identificador de los requeridos ordenados. La

complejidad espacial es $\mathbf{O}(n^2)$ y el acceso a la información es $\mathbf{O}(1)$. Por cada solicitante se debe además conocer cuál es el próximo requerido al que le deben proponer. Eso con una vector donde en la posición "i" se almacena la posición en la matriz para el solicitante "i" se puede lograr. Tendrá un tiempo de acceso $\mathbf{O}(1)$ y un espacio requerido $\mathbf{O}(n)$.

Para las preferencias de los requeridos también utilizaremos una matriz de n x n. Pero en este caso la forma de almacenar la información es diferente. Requerimos poder comparar velozmente para un requerido entre dos solicitantes a cuál prefiere. Lo que haremos es mantener para la celda i,j la posición que ocupa en la lista de preferencias del requerido i el solicitante j. La complejidad espacial es $\mathbf{O}(n^2)$. Comparar entre dos solicitantes es un proceso $\mathbf{O}(1)$.

Por último se necesita almacenar las parejas existentes y que un requerido pueda conocer de forma eficiente con quien está emparejado al momento de recibir una solicitud. Podemos resolver ambos requerimientos con un vector de n posiciones. En la posición i se almacena el identificador de la pareja actual del requerido i. Tendrá un tiempo de acceso $\mathbf{O}(1)$ y un espacio requerido $\mathbf{O}(n)$. El tiempo de preparación es $\mathbf{O}(n)$ para indicar que no hay parejas iniciales.

Teniendo en cuenta que todas las operaciones realizadas en cada ronda son $\mathbf{O}(1)$. y que tendremos como mucho "n^2" rondas. Entonces podremos afirmar que en el peor de los casos el algoritmo se ejecuta con una complejidad $\mathbf{O}(n^2)$. Si para la ejecución del algoritmo contamos previamente con las matrices de preferencias vemos que en el mejor de los casos tendremos "n" rondas. Por lo tanto podremos afirmar que es $\mathbf{\Omega}(n)$. Como se puede observar las complejidades dependen de las estructuras de datos utilizadas. Si no se definen, el análisis no es completo.

En definitiva ya sabemos que Gale-Shapley resuelve el problema del matrimonio estable de forma óptima y corresponde a un algoritmo bueno (o eficiente). Resta analizar la solución obtenida.

14 Gale-Shapley: Análisis de la solución

Probamos que al finalizar la ejecución del algoritmo de Gale-Shapley tendremos un matching estable. También sabemos que pueden existir más de un matching estable por

instancia del problema (en un ejemplo particular encontramos dos de ellos para la misma instancia). ¿Qué podemos decir de la solución encontrada por el algoritmo propuesto?

Notamos que el algoritmo no identifica un orden en el que ir seleccionando el próximo solicitante aun sin pareja para proponer. ¿Puede el uso de criterios de elección diferentes cambiar el matching resultante? ¿o siempre encontrará lo mismo? Y si es el mismo, ¿podemos encontrar alguna característica particular sobre este?

El primer impulso será probar prácticamente diferentes criterios de elección del próximo solicitante a peticionar y diferentes instancias de evaluación. En todos ellos veremos que sin importar qué criterio seleccionemos el matching estable encontrado será el mismo. También podemos tomarnos el trabajo de obtener manualmente todos los matching estables posibles y compararlos con los obtenidos. Eso nos puede dar un indicio pero no una prueba de lo que está pasando. Tenemos que demostrarlo formalmente.

Una observación que puede surgir de estas pruebas es la siguiente. Si durante la ejecución de Gale-Shapley, un solicitante es rechazado por un requerido, entonces no existe un matching estable que contenga esa pareja.

Llamemos s_1 al solicitante y r_1 al requerido. El requerido r_1 puede rechazar a s_1 en favor de otro solicitante s_2, si ocurren dos circunstancias: Si al momento que s_1 le solicita, r_1 no lo acepta por estar con una mejor pareja (s_2) o porque luego de formar pareja otro solicitante (s_2) más apetecible le ofrece emparejarse y entonces lo deja.

Recordemos que según el algoritmo cada solicitante intenta formar parejas en orden descendente de su preferencia. Por lo que s_2 prefiere a r_1 que a todas las requeridos que le quedan en su lista.

Consideremos la existencia de un matching estable que si contiene a la pareja (s_1,r_1). Entonces s_2 tiene que estar en pareja con alguien a quien prefiera a r_1. De lo contrario tendríamos una inestabilidad. Pero si en la ejecución de Gale-Shapley s_2 le solicitó a r_1 es porque todas las posibles parejas mejor rankeadas a s_2 prefieren estar con alguien diferente. Es decir que ninguna pareja posible mejor que r_1 le queda a s_2. Y por lo tanto llegamos a una contradicción a nuestra consideración.

Este punto tiene consecuencias en el tipo de matching estable que consigue el algoritmo Gale-Shapley. Como no existen matching estables entre los solicitantes y los requeridos que lo rechazaron entonces cada solicitante está emparejado con el mejor requerido que puede tener en cualquier matching posible. Por lo tanto **el matching estable es óptimo para los solicitantes**. Y más aún, no depende del orden en que los solicitantes sin pareja peticionan.

Podemos revisar todos los posibles matching y hacer un listado reducido por cada solicitante de los requeridos con los que puede estar en pareja ordenado por prioridad. Llamamos a esta lista parejas válidas. La pareja seleccionada por Gale-Shapley siempre será la primera de esa lista. Por lo que lo llamamos la **mejor pareja posible**. El mismo proceso podemos realizar con las parejas válidas de los requeridos. Si marcamos aquellas parejas que consiguen mediante el algoritmos de aceptación diferida, notaremos que siempre es el último en su lista. Podemos llamar a este su **peor pareja posible**. Parece ser un mal negocio para el requerido la pasividad. Pero antes de continuar hay que demostrar que este es el caso siempre.

Consideremos la existencia de la pareja (s_1,r_1) en el matching producido por Gale-Shapley. Y la existencia de las parejas (s_2,r_1) y (s_1,r_2) en otro matching estable donde r_1 prefiere a s_1 antes que a s_2 . Entonces s_1 debe preferir a r_2 a s_2 para que no sea una inestabilidad. Que contradice que (s_1,r_1) es la mejor solución para r_1. Comparándolo con el resto de los matching estables disponibles eso nos lleva a observar que en Gale-Shapley el requerido tendrá siempre igual o peor pareja que en el resto de ellos.

Hay diferentes formas de probar estas inequidades, las anteriores están basadas a las realizadas por Knuth[13].

15 Gale-Shapley: Conclusiones.

El tratamiento del problema del matrimonio estable fue propuesto en 1962 por David Galey y Lloyd Shapley. Ciertamente no lo hicieron para resolver un problema de divorcios

[13] "Stable Marriage and Its Relation to Other Combinatorial Problems. An Introduction to the Mathematical Analysis of Algorithms", Donald E. Knuth, 1997

ni infidelidades. Trabajaron con un problema más general (emparejamientos estables) y buscaron simplificarlo para resolverlo y tener un punto inicial de trabajo. Usar hombre y mujeres y casamientos tenía como objetivo dar un marco fácil de entender y seguir

El papel del requerido (mujer) es de quien rompe las relaciones y se asegura de mantenerse siempre en pareja mejorando en cada nueva oportunidad. Podría parecer ventajoso a primera vista, pero en un análisis detallado es quien se lleva la peor parte. El papel del solicitante (hombre) como alguien activo y que trata de encontrar la mejor pareja es quien en última instancia se ve beneficiado.

Hablamos de una época donde el emparejamiento entre personas del mismo género no era muy bien tolerado, donde el concepto de "caballerosidad" y "buenas costumbres" otorgaba a las mujeres un lugar en un segundo plano. Si bien comenzaba lentamente a resquebrajarse estos conceptos, faltarían años para que se observara un cambio significativo. En la actualidad presentar un problema similar no es políticamente correcto y tampoco aplicable a la situación planteada.

Al aplicar el algoritmo de Gale-Shapley se puede observar que entre los dos conjuntos uno de ellos se ve perjudicado frente al otro de la peor manera posible. Hay una asimetría de poder entre los mismos. Esa diferencia puede existir en otros casos prácticos y es donde este puede ser utilizado. El más conocido en la bibliografía corresponde al problema de los alumnos y los centros de educación. Donde los alumnos quieren ser aceptados y los centros quieren aceptar solo a un número máximo de ellos. Cada uno tiene sus preferencias. Ciertamente es el centro educativo es el que ofrece pues es quien tiene más poder entre estos dos grupos.

Existe una gran cantidad de variantes del problema original. Algunos de ellos incluyen: listas incompletas de preferencias, indiferencia entre candidatos, parejas de más de 2 elementos, emparejamiento igualitario (donde no hay ventaja entre grupo de pertenencia), emparejamiento entre elementos del mismo conjunto, etc. Cada una de ellas tiene potencialmente otras aplicaciones. Muchas de estas variantes se encuentran disponibles en un libro de Dan Gusfield y Robert Irving[14]

[14] Dan Gusfield. Robert Irving, The stable marriage problem: Structure and algorithms, The MIT Press, Cambridge, MA

En el caso general hemos realizado un análisis de los aspectos a los que le daremos relevancia para el estudio: optimalidad, eficiencia e implicancias de la solución encontrada. Este será el camino a transitar de aquí en adelante.