TEORÍA DE ALGORITMOS 1
División y conquista

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

1 Introducción

La expresión “Dīvide et īmpera” (cuya traducción del latín corresponde a “divide y domina”) hace referencia a una máxima política militar utilizada por los romanos para consolidar y propagar su poder entre los pueblos vecinos. Corresponde a fomentar las divisiones existentes entre los grupos de poder para poder dominarlos y manipularlos sin riesgo. La frase (o alguna de sus variantes) fueron utilizadas por el emperador romano Julio Cesar y siglos después por Napoleón Bonaparte. La puesta en práctica de esta estrategia fue, es y posiblemente será de amplia utilización práctica entre los que ostentan algún tipo de poder.

En el campo de la informática corresponde a un paradigma de resolución de problemas que hace de la **división** de la instancia del problema en subproblemas de menor tamaño su estrategia para facilitar la resolución del mismo. Los subproblemas resultantes se resuelven de forma independiente de los otros generados. Se cumple que cada subproblema conserva todas las características del problema original. Esto permite que este mismo pueda a su vez ser subdividido en nuevos subproblemas de un menor tamaño. Se produce un **proceso recursivo** donde se continúa esta división hasta un **caso base** donde el problema tiene un tamaño lo suficientemente pequeño para su resolución de forma trivial. Es conocida como **conquista** la resolución de todos los subproblemas correspondientes a un problema. Posterior a esta se procede a un proceso de **combinación** de estos para generar la solución del problema general.

Al igual que los algoritmos de tipo Greedy no cualquier problema se puede resolver mediante división y conquista. Comparten con el recién nombrado la propiedad de **subestructura óptima** (optimal substructure). La capacidad de separar el problema en

pequeños subproblemas (en este caso recursivamente) y su resolución óptima nos lleva a la solución óptima global. La naturaleza de la división es sin embargo diferente. La podemos pensar como una partición “horizontal” o en anchura y en greedy una partición “vertical” o en profundidad.

La metodología de división y conquista (o “divide y vencerás”) en el campo de resolución de problemas lógico matemáticos antecede a la existencia de las computadoras modernas. Se han encontrado aplicaciones de ellas tan lejos como en la civilización babilónica[1]. En alfabeto cuneiforme grabado en tabletas de arcilla se halló listado de elementos ordenados para facilitar su búsqueda. En esos casos la búsqueda binaria, el primer algoritmo que exploramos, es requerido para encontrar eficientemente lo que deseamos. Más cercano en la historia se puede rastrear en el trabajo del matemático Carl Friedrich Gauss el uso de la metodología aquí analizada[2]. Más concretamente en su trabajo “Theoria Interpolationis Methodo Nova Tractata” publicada póstumamente dentro de su colección de obras en 1866. Allí expone un proceso similar al siglos después redescubierto para resolver la transformada de fourier rápida (Fast Fourier transform). En el primer ejemplo solo se dividía el problema en uno menor. En este segundo se puede observar una división en varios subproblemas.

Matemáticamente, para analizar la complejidad surgida por la subdivisión de problemas en otros de menor tamaño de forma recursiva, utilizaremos una **relación de recurrencia**. Corresponde a una ecuación que define una secuencia recursiva. Donde cada término de la secuencia es definido como una función de términos anteriores. Esto genera una recursión que finaliza cuando se llega a un caso base donde la función se encuentra previamente definida.

Las relaciones de recurrencia en rigor se pueden utilizar para representar no solo los problemas de división y conquista. Están implícitamente dentro de los problemas Greedy. Y los utilizaremos nuevamente dándole un papel central en los problemas que utilicen programación dinámica.

[1] D. E. Knuth, Selected Papers on Computer Science, 1996, Cambridge Univ. Press
[2] Heideman, M. T., D. H. Johnson, C. S. Burrus, "Gauss and the history of the fast Fourier transform", 1985, IEEE ASSP Magazine

Antes de pasar a diferentes problemas, que podemos resolver utilizando un algoritmo de división y conquista, aclaramos que el desarrollo de sus soluciones las realizaremos de forma recursiva. Muchos de estos algoritmos se pueden transformar en iterativos. Esta última aproximación suele ser más eficiente en el uso de los recursos al implementarlos en los lenguajes modernos de programación. Sin embargo, suelen ser menos claros para su comprensión al momento de analizarlos. Una ventaja de los algoritmos recursivos es que permiten resolverlos de forma descentralizada. Cada subproblema dependiente directamente de un problema determinado para su conquista se puede paralelizar, resolver y luego combinar.

2 Búsqueda Binaria y Conteo de ocurrencias

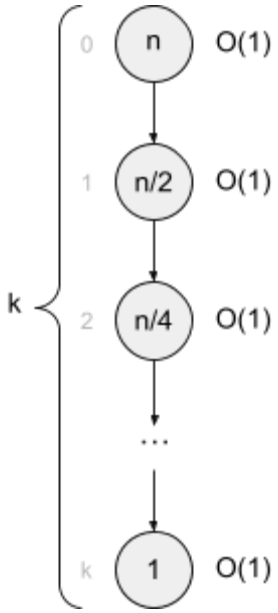
El caso más simple y conocido entre todos los problemas que utilizan división y conquista corresponde a la búsqueda binaria. Partimos de un conjunto de elementos previamente ordenados y queremos hallar si un elemento determinado pertenece al mismo y en qué posición se encuentra. La estrategia es simple: verificar si el elemento a la mitad del conjunto es mayor, menor o igual al buscado. En el último caso terminamos el proceso. Si es menor sabemos que cada uno de los elementos anteriores al comparado no pueden ser el buscado. Por lo tanto podemos descartarlos y seguir buscando recursivamente en la mitad restante. Similarmente podemos descartar la segunda mitad en caso de ser mayor al buscado. De esa forma cada nuevo subproblema contendrá como mucho la mitad de los elementos (contemplando la posibilidad que la cantidad de elementos sea impar). El proceso continúa hasta encontrar el elemento buscado o - en el peor caso - hasta llegar a un solo elemento y que no corresponda al buscado. El pseudocódigo se podría indicar de la siguiente manera:

Búsqueda Binaria

```
Sea L una lista de “n” elementos ordenados de menor a mayor
Sea e el elemento a buscar

BusquedaBinaria(L,e,inicio,final):
        Si final<=inicio
                Si L[inicio]=e
                        Retornar inicio
```

```
            Sino
                  Retornar vacio

      Sea mitad el valor intermedio entre inicio y fin

      Si L[mitad]=e
            Retornar mitad
      Si L[mitad]>e
            retornar BusquedaBinaria(L,e,inicio,mitad-1)
      Si L[mitad]<e
            retornar BusquedaBinaria(L,e,mitad+1,final)

BusquedaBinaria(L,e,0,n-1)
```

Para expresar este problema e intentar calcular su complejidad utilizamos una relación de recurrencia T(n) cuyo parámetro de entrada es el tamaño del conjunto. El problema se divide en un único subproblema de la mitad del tamaño actual. Además en cada subproblema se realizan 3 comparaciones como mucho y un cálculo del valor medio. Estas operaciones no dependen del tamaño del conjunto y por lo tanto son **O**(1) en su conjunto. Por lo tanto podemos escribir la recurrencia como $T(n) = 1T(\frac{n}{2}) + O(1)$ con el caso base de $T(1) = O(1)$



En el análisis vemos que tendremos "k" subproblemas por las llamadas recursivas. Llegaremos al caso base donde tendremos solo un elemento. En cada subproblema la complejidad asociada es **O**(1). Por lo que podemos afirmar que la complejidad global será $\sum_{0}^{k} O(1) = (k+1)O(1) = kO(1)$. El valor de k lo podemos obtener descubriendo cual es la cantidad divisiones de n posibles por la mitad. Si vemos cada subproblema como un nivel de recursión, la expresión $\frac{n}{2^l}$ nos indica la cantidad de elementos en el nivel l. En el último nivel "k" tendremos solo un elemento. Por lo que $n = 2^k$ y podemos despejar k como como $log_2 n = k$. Reemplazamos k en la expresión de la complejidad global, llegando al conocido **O**(log n).

Existen diversos problemas similares o variantes de la búsqueda binaria. Veamos unos de ellos a continuación al que llamaremos “Conteo de ocurrencias”. Lo enunciamos de la siguiente manera:

Conteo de ocurrencias
Dado “L” un listado ordenado de “n” elementos y un elemento “e” determinado. Deseamos conocer la cantidad total de veces que “e” se encuentra en “L”.

Al estar ordenado el listado sabemos que todos los elementos a buscar estarán contiguos. Una solución muy rudimentaria consiste en recorrer linealmente todo el listado. Esa solución sería en el peor de los casos $\mathbf{O}(n)$ por ejemplo si el elemento no se encuentra. Una alternativa que mejora ciertos casos corresponde a aprovechar la búsqueda binaria. En $\mathbf{O}(\log n)$ nos podemos enterar si el elemento al menos existe en el listado y si la respuesta es afirmativa en qué posición se encuentra. Desde ese lugar podemos movernos linealmente para un extremo y luego el otro contando las ocurrencias. Esta aproximación sigue siendo $\mathbf{O}(n)$. Dado que podríamos estar frente a un elemento que es muy representativo en el conjunto total.

Nuestro objetivo será disminuir la complejidad lineal. Lo lograremos modificando la búsqueda binaria. En primer lugar no queremos encontrar el elemento en sí, sino la primera (y respectivamente última) posición en la que ocurre. Por eso quitamos de nuestro pseudocódigo la verificación de igualdad “Si L[mitad]=e ”. De esa forma el algoritmo no termina al encontrar por primera vez el elemento. Seguirá reduciendo el problema recursivamente. Modificamos, también, que aun al terminar con un solo elemento y no sea el buscado retorna la posición. Terminará encontrando la última repetición del elemento buscado o el primero que lo supere. Invirtiendo el orden de la comparación entre el elemento y el valor del medio podemos encontrar la primera aparición del elemento o el elemento inmediatamente menor. El siguiente pseudocódigo muestra los cambios.

Contando ocurrencias
Sea L una lista de “n” elementos ordenados de menor a mayor Sea e el elemento a contar

```
Ocurrencias(L,e,inicio,final,extremo):
        Si final<=inicio
                Retornar inicio

        Sea mitad el valor intermedio entre inicio y fin

        Si extremo = 'final'
                Si L[mitad]>e
                        retornar Ocurrencias(L,e,inicio,mitad-1)
                retornar Ocurrencias(L,e,mitad+1,final)
        Si extremo = 'inicial'
                Si L[mitad]<e
                        retornar Ocurrencias(L,e,mitad+1,final)
                retornar Ocurrencias(L,e,inicio,mitad-1)

inicio = Ocurrencias(L,e,0,n-1,'inicial')
fin = Ocurrencias(L,e,0,n-1,'final')
```

Resta verificar si en la posición de "inicio" o su posterior se encuentra el elemento buscado. Similarmente en la posición "fin" o su anterior. Si no los encuentra, éste no está en la lista. Si los encuentra, la diferencia entre las posiciones es la cantidad total de veces que aparece en ella.

El proceso corresponde a dos búsquedas similares a las binarias. La complejidad de cada una es $\mathbf{O}(\log n)$. Luego se realizan un número constante de operaciones de comparación y una resta. Por lo tanto todo el proceso se podría resumir como $\mathbf{O}(2\log n + c) = \mathbf{O}(\log n)$.

3. Merge Sort y conteo de inversiones

Merge Sort es un clásico algoritmo de ordenamiento que utiliza división y conquista. La autoría y fecha de creación fue adjudicado por Knuth[3] a John von Neumann en el año 1945. La propuesta es sencilla: dividir el problema en mitades, resolver cada una de ellas recursivamente. Como resultado se espera tener 2 listas de elementos ordenados. La combinación de estas listas mediante el proceso que se conoce como "merge" permite

[3] The Art of Computer Programming. Vol. 3: Sorting and Searching. 2da edición, Knuth, Donald, 1998, Addison-Wesley. pp. 158–168.

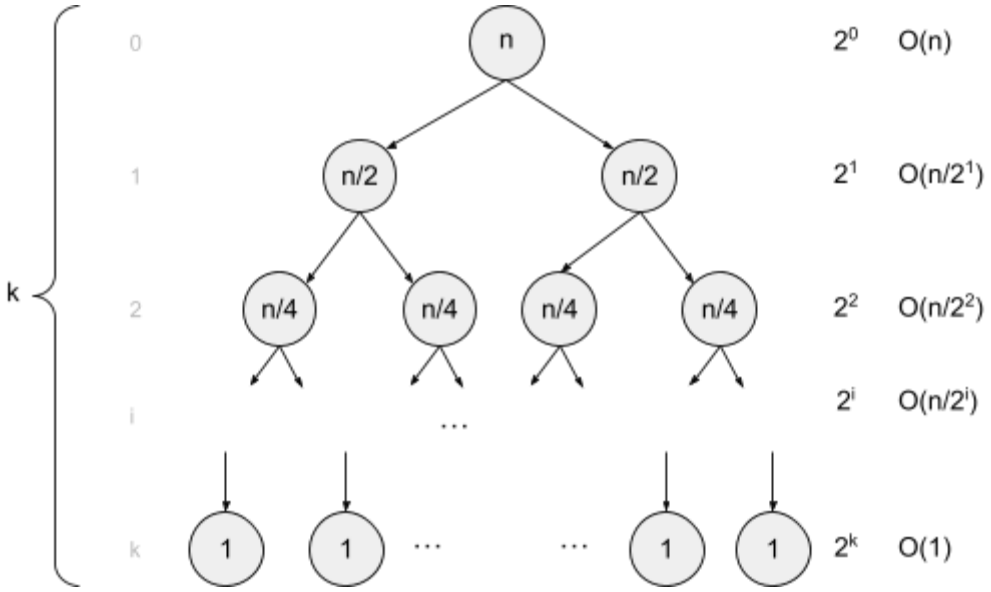
obtener una única lista completamente ordenada. El siguiente pseudocódigo resume el proceso:

Merge Sort

```
Sea L una lista de "n" elementos

MergeSort(L):
        Si |L|=1 Retornar L

        Sea A y B la primera y segunda mitad de L respectivamente
        Retornar Merge ( MergeSort(A) , MergeSort(B) )

Merge(A,B):
        Sea M una lista vacía
        Repetir
                Sea a primer elemento de A
                Sea b primer elemento de B
                Si a<b
                        insertar a en M
                        quitar a de A
                Sino
                        insertar b en M
                        quitar b de B
        Mientras queden elementos en A o en B
        Insertar los elementos restantes de A en M
        Insertar los elementos restantes de B en M
        Retornar M

MergeSort(L)
```

Para analizar la complejidad temporal del proceso podemos ver que por cada problema se desprenden 2 nuevos con la mitad de los elementos. El proceso de merge o combinación de los problemas consume un $\mathbf{O}(n)$ de tiempo. Por último el caso base cuando solo tengo un elemento corresponde a simplemente retornarlo en $\mathbf{O}(1)$. Con esto podemos expresar la relación de recurrencia como $T(n) = 2T(\frac{n}{2}) + O(n)$ con el caso base de $T(1) = O(1)$

En la siguiente imagen se puede apreciar la cantidad de problemas que se generan por nivel de recursión y con cuantos elementos cuenta cada uno de ellos.



Podemos aprovechar parte del análisis que realizamos para la búsqueda binaria: El nivel máximo de recursión está dado por el mismo patrón de división en mitades. Ocurre cuando $k = log_2 n$. La complejidad total la podemos descomponer en la sumatoria de las complejidades de cada nivel. Para el nivel i, tendremos 2^i problemas donde cada problema tiene $n/2^i$ elementos. Cada subproblema se resuelve en tiempo lineal según la cantidad de elementos que contiene. Es decir en $c\frac{n}{2^i}$. Uniendo todo lo anterior podemos expresar la complejidad final como $\sum_{i=0}^{k} 2^i * c\frac{n}{2^i}$. Podemos cancelar y extraer las constantes y expresar como $c\sum_{i=0}^{k} n$ y reemplazando k con su correspondiente valor de n llegamos a que el trabajo final es c.n.log(n). Es decir que la complejidad espacial del mergesort corresponde a **O**(nlogn)

Con respecto al análisis espacial se puede construir en **O**(n) teniendo en cuenta cómo se van resolviendo recursivamente los problemas.

Así como en la búsqueda binaria, el mergesort tiene una gran variedad de problemas similares. El que veremos a continuación lo llamaremos “conteo de inversiones”.

Contamos con una determinada lista de elementos que esperamos que tenga un determinado orden. Para desarrollar el problema supondremos que el orden esperado es de menor a mayor. Una inversión en el orden corresponde a un par de elementos que al compararlos entre sí, no cumplen con el orden requerido. Un listado totalmente ordenado tendrá cero inversiones. Mientras que la máxima cantidad de inversiones se dará si el mismo está con el ordenamiento inverso. Formalicemos el concepto de inversión.

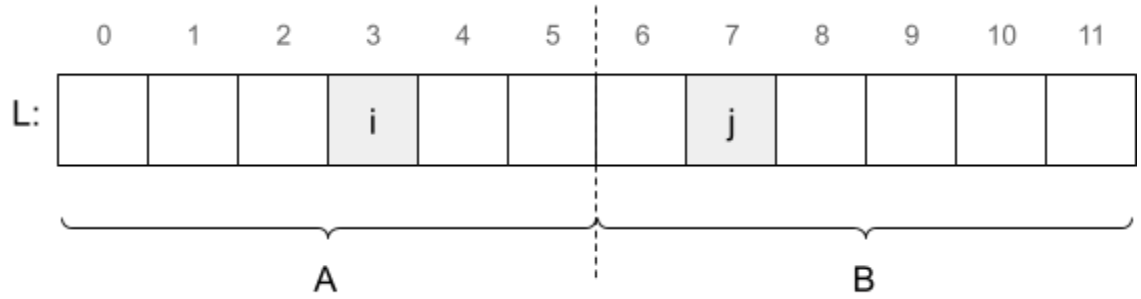
Llamaremos l_i al elemento en la lista L que se encuentra en la posición i. Si la lista de elementos cuenta con "n" elementos "i" podrá tomar un valor entre 0 y n-1. Al comparar el elemento l_i y l_j, con i<j, si $l_i > l_j$ entonces contamos una inversión. El elemento en la posición i puede tener como mucho n-i inversiones.

Nuestro problema lo podemos expresar como:

Conteo de inversiones
Dado "L" un listado de "n" elementos que esperamos que esté ordenado de menor a mayor. Deseamos conocer la cantidad total de inversiones de elementos que contiene

Una manera sencilla de obtener la totalidad de las inversiones corresponde a iterar por cada una de los elementos. Con cada uno de ellos verificaremos con cuántos de los posteriores está invertido. Esto corresponde a un proceso $\mathbf{O}(n^2)$. Buscaremos de la mano de la división y conquista disminuir este tiempo. Lo hallaremos utilizando una variante del mergesort.

Supongamos que dividimos la lista de elementos "L" por la mitad. Asumamos que tanto en la primera mitad "A" como en la segunda "B" no hay elementos invertidos. Cada mitad está ordenada totalmente de menor a mayor. Eso significa que en el conjunto total las únicas inversiones posibles corresponden a elementos de "A" en relación a elementos de "B". Si el elemento i de "A" presenta una inversión con el elemento j de "B" entonces $l_i > l_j$ y además i<j. Pero además, por el orden en las mitades, todos los elementos posteriores a i en A también estarán invertidos al respecto de j.



De forma similar, si el elemento i en A no presenta una inversión con el elemento j en B, entonces no tendrá inversiones con los elementos anteriores a j en B. Uniendo estas dos observaciones se puede armar un proceso basado en el merge del mergesort para contar las inversiones de cada elemento y las totales de la lista. Con un adicional que resultará fundamental, el resultado además tendrá las sublistas A y B unidas de forma ordenada. A este proceso le corresponde una complejidad de $\mathbf{O}(n)$

Comenzamos suponiendo que teníamos dos conjuntos que llamamos A y B que estaban ordenados. Aunque, no podemos pretender que lo estén. Pero podemos utilizar este proceso recursivamente. En el caso base cada lista tiene sólo un elemento y por lo tanto está ordenada. El primer merge corresponde a unir dos listas de un solo elemento. El resultado será una nueva lista ordenada. Luego por cada combinación de resultados también tendremos listas ordenadas. Al ser un proceso de división en mitades, nuevamente tendremos una recursión máxima de log(n). Pongamos todo junto el el siguiente pseudocódigo

Conteo de inversiones por división y conquista

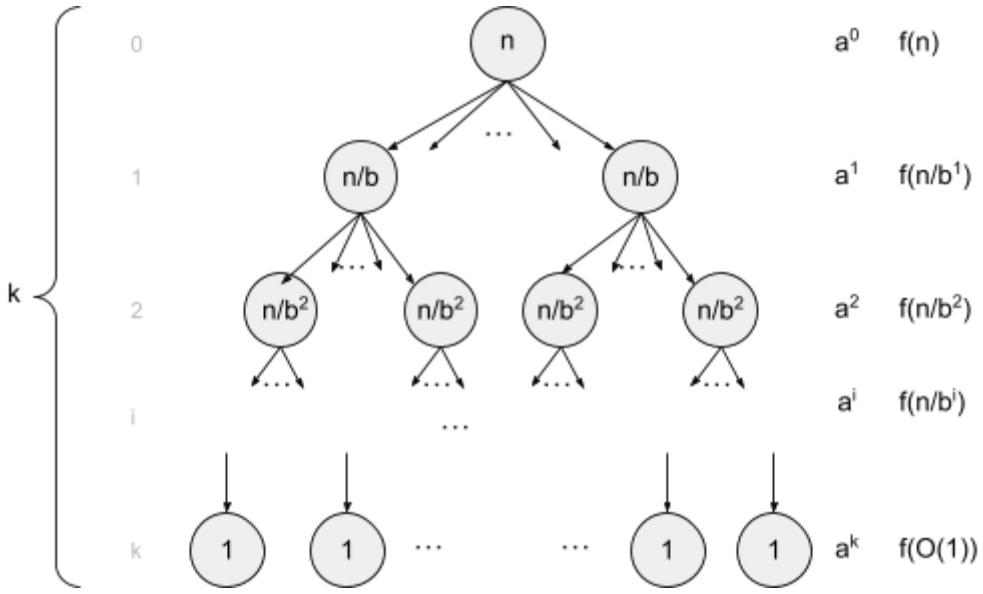
```
Sea L una lista de “n” elementos
TotalInv = 0

ContInv(L):
        Si |L|=1 Retornar L

        Sea A y B la primera y segunda mitad de L respectivamente
        Retornar Contar ( ContInv(A) , ContInv(B) )

Contar(A,B):
        Sea M una lista vacía
```

```
        totB = |B|
        Repetir
                Sea a primer elemento de A
                Sea b primer elemento de B
                Si a<b
                        insertar a en M
                        quitar a de A
                Sino
                        insertar b en M
                        quitar b de B
                        TotalInv += elementos Restantes en B

        Mientras queden elementos en A o en B

        TotalInv += totB * cantidad de elementos restantes en A
        Insertar los elementos restantes de A en M
        Insertar los elementos restantes de B en M
        Retornar M

ContInv(L)
```

Nuevamente tenemos 2 subproblemas generados con la mitad de los elementos. Además un $\mathbf{O}$(n) por la combinación. Con esto podemos expresar la relación de recurrencia como $T(n) = 2T(\frac{n}{2}) + O(n)$ con el caso base de $T(1) = O(1)$. Misma complejidad que el mergesort y por lo tanto $\mathbf{O}$(n logn)

4. Análisis de relaciones de recurrencia

Cuando analizamos la complejidad de una gran cantidad de los algoritmos de división y conquista llegamos a una expresión de una relación de recurrencia del tipo $T(n) = aT(\frac{n}{b}) + f(n)$. Recordemos para los últimos dos problemas que valores toman los parámetros a, b y f(n). En la búsqueda binaria a=1, b=2 y f(n)=O(1). En mergesort a=2, b=2 y f(n)=O(n). Próximamente veremos otras variantes. Por eso es de utilidad tratar de estudiar cómo se puede resolver las mismas. En la imagen siguiente se puede ver un esquema del crecimiento de la cantidad de problemas y su complejidad asociada.



La profundidad de la recursión estará atada, como antes, a la cantidad de niveles requeridos para que en el problema solo quede un único elemento (o los elementos mínimos para el caso base). En ese caso tendremos $\frac{n}{b^k} = 1$, que corresponde a cuando $log_b n = k$. La complejidad en el nivel i estará dada por la cantidad de subproblemas a^i y el trabajo realizado en cada uno de estos $f(n/b^i)$. La complejidad total estará dada por la sumatoria de las complejidades de todos los niveles de recursión $\sum_{i=0}^{log_b n} a^i * f\left(\frac{n}{b^i}\right)$.

Matemáticamente se debe operar la expresión para llegar a un resultado global. Este proceso para aquellos no experimentados puede ser tedioso y llevar a errores. Por suerte algunos autores han encontrado algunos métodos que ayudan a facilitar su cálculo.

El método unificador[4] ("unifying method") es el más conocido entre todos ellos. Tal vez no por el nombre que le dieron sus autores, sino por el que se popularizó por el libro

[4] A general method for solving divide-and-conquer recurrences, Bentley Jon Louis; Haken Dorothea; Saxe James B., Septiembre 1980, ACM SIGACT News, 12 (3): pp 36–44

Introduction to Algorithms[5] (popularmente "el cormen"): Método maestro. Por eso lo llamaremos en adelante de esa última manera

El método maestro requiere que la ecuación de recurrencia se exprese como $T(n) = aT(\frac{n}{b}) + f(n)$ con a≥1, b≥1 y T(0) una constante. Presenta 3 casos de aplicación. Estos casos dependen de la relación entre f(n) y $n^{log_b a}$. Dependiendo cual de los dos domina para valores grandes de n, será el resultado de la ecuación de recurrencia.

- caso 1: $f(n) = O(n^{log_b a - \epsilon})$ con ε>0 ⇒ $T(n) = \Theta(n^{log_b a})$

  Si la función f(n) es acotada superiormente por la función $n^{log_b a - \epsilon}$, es decir está acotada por una función que a la vez está acotada por $n^{log_b a}$. Entonces la complejidad estará dada por $\Theta(n^{log_b a})$.
- caso 2: $f(n) = \Theta(n^{log_b a})$ ⇒ $T(n) = \Theta(logn * n^{log_b a})$

  Si la función f(n) se encuentra acotada tanto superior como inferiormente a $n^{log_b a}$.

  Por propiedad de las cotas asintóticas también se puede decir que $n^{log_b a}$ acota superior como inferiormente a f(n). En este caso la complejidad estará dada por $\Theta(logn * n^{log_b a})$.
- caso 3: $f(n) = \Omega(n^{log_b a + \epsilon})$ con ε>0 y a$f(\frac{n}{b}) \leq cf(n)$ con c<1 y para todo valor de n a partir de un valor de n_0 ⇒ $T(n) = \Theta(f(n))$

  En este último caso se tiene que evaluar más de una condición. En primer lugar si la función f(n) es acotada inferiormente por la función $n^{log_b a + \epsilon}$, es decir está acotada por una función que a la vez está acotada por $n^{log_b a}$. En segundo lugar se evalúa la naturaleza de la función f(n) y su crecimiento a medida que el valor n crece. En este caso la complejidad estará dada por $\Theta(f(n))$. Si la última condición no se cumple, no podremos aplicar el teorema maestro.

[5] Introduction to Algorithms 4th edition, Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest; Clifford Stein,2022, The MIT Press

No vamos a realizar una demostración del teorema matemáticamente. Una excelente y extensa demostración está presente en el libro de Cormen.

Un ejemplo donde el teorema maestro no funciona es la siguiente relación de recurrencia. $T(n) = 2T(\frac{n}{2}) + O(nlogn)$. Si se prueba no entra en ninguno de los tres casos. Para esto se debe proceder o realizando los cálculos completos manuales. Se puede demostrar un caso general donde si f(n)=**Θ**($n^{log_b a} log^k n$), la solución a la recurrencia es **Θ**($n^{log_b a} log^{k+1} n$). Esta recurrencia aparecerá más adelante y esta información nos será útil

Tendremos otros tipos de recurrencias que aparecerán en diferentes algoritmos. Se han generado otros teoremas que resultan útiles en estos casos. Por ejemplo el método Akra-Bazzi[6] para algunas recurrencias donde en un mismo subproblema se dividen en subproblemas de diferentes tamaños. Para otros tipos de recurrencia hay ejemplos de cómo resolverlas en varios textos de estudio. Uno de ellos con versión en castellano "Fundamentos de algoritmia"[7] de Brassard y Bratley

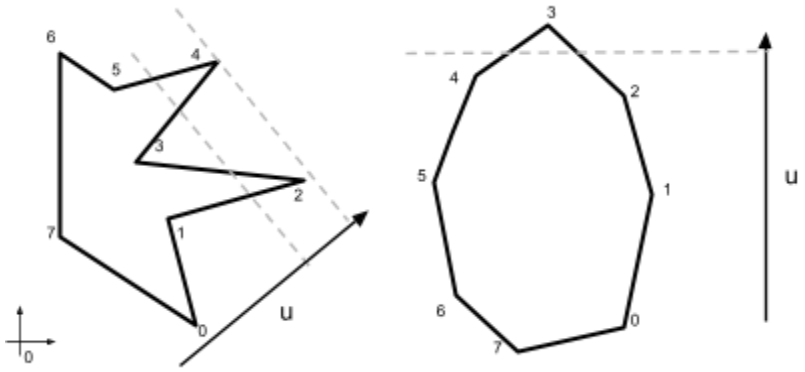
5. Punto extremo en polígono convexo

Dentro de la computación gráfica se trabaja con diferentes figuras geométricas. Sobre ellas se realizan diferentes tareas y transformaciones. Entre ellas detectar colisiones, encontrar cajas contenedoras de elementos, trazar trayectorias, entre otras. En muchas de ellas podemos encontrar problemas comunes a resolver. Una de ellas corresponde a encontrar puntos extremos de una figura. Es decir, siguiendo una dirección (matemáticamente un vector) queremos conocer cuál es el punto más lejano de la figura que podemos encontrar. Un caso simple que abordaremos corresponde a trabajar en dos dimensiones y con polígonos.

Recordemos que un polígono corresponde a una secuencia finita de segmentos que encierran una región del plano. Alternativamente lo podemos ver como una secuencia de 3 o más puntos p_0=(x,y), p_2, …, p_{n-1} correspondientes a los vértices de la figura. Cada par de puntos p_i y $p_{i+1(mod\ n-1)}$, con i de 0 a n-1 unidos por un segmento.

[6] On the solution of linear recurrence equations, Mohamad Akra; Louay Bazzi, 1998, Computational Optimization and Applications 10
[7] Fundamento de algoritmia, G. Brassard; P. Bratley, 2004, Pearson

En la siguiente imagen podemos ver 2 polígonos de 8 puntos (y 8 segmentos). El primero de ellos un polígono cóncavo y el segundo convexo. Determinar si es de un tipo u otros es muy sencillo: un polígono es convexo si sus ángulos interiores miden a lo sumo 180 grados. Adicionalmente se puede ver que si es convexo también es monótono respecto a cualquier recta u. Es decir que cualquier línea ortogonal a u corta al polígono a lo sumo en dos puntos. Si no cumplen las condiciones anteriores estaremos frente a un polígono cóncavo.



El polígono de la izquierda es cóncavo. El ángulo interno correspondiente al punto 1 es mayor a 180 grados. Además se puede ver que la línea ortogonal a u corta al polígono en 4 puntos. El polígono de la derecha es convexo. El ángulo interno de todos los puntos es menor a 180. Además para ninguna posible recta u se lograra que corte en más de 2 puntos al polígono una línea ortogonal a esta.

Utilicemos como recta al vector dirección del cual queremos obtener el punto máximo. Visualmente podemos desplazar una línea ortogonal a esta sobre el polígono. Veremos que siempre al menos un punto que define el polígono corresponderá al último punto visitado por esta línea. Por lo tanto para encontrar el punto extremo del polígono (o al menos uno de ellos) bastará con analizar los puntos que definen al mismo.

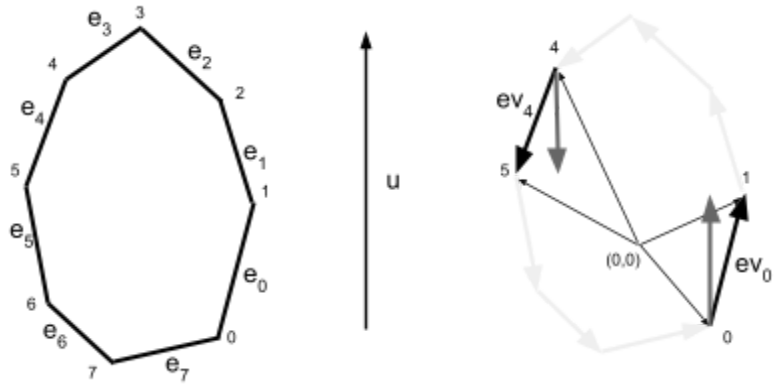
La primera solución que se nos puede ocurrir es representar cada punto como un vector y proyectarlo con el vector dirección sobre el queremos resolver el problema. Nos quedaremos con el máximo. Para un polígono de n puntos tendremos un proceso O(n). Corresponde a un proceso lineal y al mismo tiempo a una resolución por fuerza bruta. Esto podría parecer suficiente, sin embargo en ocasiones debemos realizar esta búsqueda para polígonos de tamaño n muy grandes o más comúnmente repetir este proceso para un gran

número de diferentes polígonos de tamaño n en un tiempo muy pequeño. Por lo que es deseable encontrar una solución superadora.

La misma es posible pero solo para un subconjunto de los polígonos: aquellos que son convexos. Por este motivo y gracias a propiedades matemáticas de estos últimos es que muchas veces se intenta trabajar con ellos. Y cuando esto no es posible termina descomponiendo los polígonos cóncavos en un conjunto de polígonos convexos. Antes de continuar, terminemos de definir nuestro problema

Punto extremo en un polígono convexo
Dado “P” un polígono convexo de n vértices y un vector dirección “u”. Deseamos encontrar el punto extremo de P según la dirección “u”

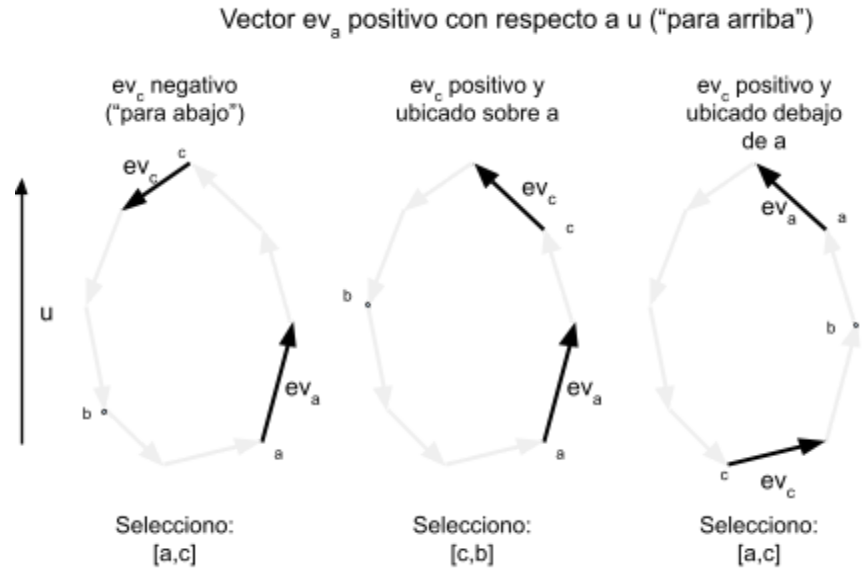
Por unos instantes imaginemos el polígono como una pista de ciclismo. Cuando el polígono es cóncavo si recorremos la misma en ocasiones deberemos doblar en el sentido horario y en otras en sentido antihorario. Sin embargo si la pista es un polígono convexo el giro siempre será en el mismo sentido. Dependiendo para qué lado iniciemos todos los giros serán antihorarios u horarios. De aquí en adelante tomaremos el recorrido a realizar en sentido antihorario.

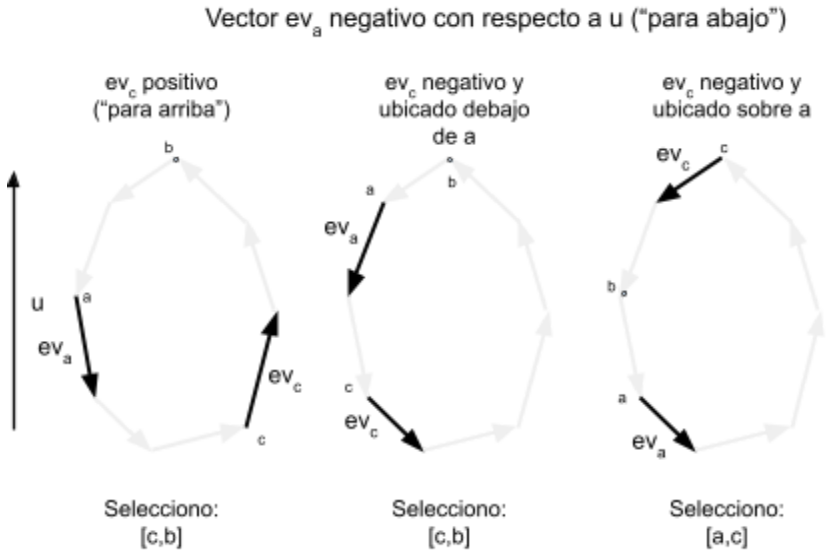


En la imagen podemos ver el mismo polígono convexo del problema anterior en su recorrido antihorario. Llamaremos e_i al i-esimo segmento que une los puntos p_i y $p_{i+1(\text{mod } n-1)}$. Podemos expresar vectorialmente cada uno de ellos como la diferencia entre los puntos. Concretamente: $ev_i = p_{i+1(\text{mod } n-1)} - p_i$. En el análisis de esos vectores proyectados a la dirección u del máximo veremos que algunos tienen valor positivo y otros negativos. Por

ejemplo ev_0 tiene un valor positivo y ev_4 negativo. Siguiendo la evolución de la proyección podemos ver que el punto máximo es el último que tiene un valor positivo. Obviamente esto solo si es convexo. Con esto en mente podemos evitar recorrer todos los puntos para encontrar nuestra solución.

Supongamos que sabemos que el punto máximo se encuentra en un segmento del polígono entre los puntos p_a y p_b. Partimos ese segmento en dos partes de igual tamaño y llamaremos al punto intermedio p_c. El máximo deberá estar en alguno de los dos segmentos. Buscaremos desechar la mitad del segmento seguros que el máximo se no se encuentra en ellos. Recursivamente con el segmento restante continuaremos dividiéndolo en mitades hasta encontrar el punto máximo. El criterio de elección del segmento con el máximo debe tener en cuenta 6 casos que se muestran en la imagen a continuación.





Determinar cuál de los casos es aplicable se realiza comparando solo dos puntos: p_a y p_c. Nos encontramos por lo tanto con una operación $\mathbf{O}(1)$. Con la mitad de los puntos repetiremos el proceso de división hasta quedarnos con solo 3 o 2. Habremos llegado al caso base. En él debemos comparar los puntos entre ellos y retornar el que corresponde al máximo. El caso base tiene una complejidad también $\mathbf{O}(1)$.

Podemos resumir todo el algoritmo en el siguiente pseudocódigo.

Punto extremo en un polígono convexo con División y conquista

```
Sea P puntos del polígono convexo de n vértices
Sea u dirección de búsqueda del máximo

BuscarMaximo(P,a,b):
        Si b-a <= 3
                Retornar máximo entre los puntos en P entre a y b proyectados a u

        Sea c el valor intermedio entre inicio y fin

        Si p[a] es positivo proyectado en u
                Si p[c] es negativo proyectado en u
                        retornar BuscarMaximo(P,a,c)
                Si P[c] ubicado sobre P[a] en relación a proyección u
                        retornar BuscarMaximo(P,c,b)
                retornar BuscarMaximo(P,a,c)
```

```
        Si p[c] es positivo proyectado en u
                retornar BuscarMaximo(P,c,b)
        Si P[c] ubicado debajo de P[a] en relación a proyección u
                retornar BuscarMaximo(P,c,b)
        retornar BuscarMaximo(P,a,c)

BuscarMaximo(P,0,n-1)
```

El algoritmo comienza con todos los puntos y recursivamente se subdivide en 1 subproblema con la mitad de los puntos. El trabajo de combinación es simplemente retornar el resultado encontrado en el subproblema menor. Con esta información podemos expresar la relación de recurrencia del problema como:

T(n) = T(n/2) + O(1)
T(2)=T(3)=O(1)

Utilizando el teorema maestro tenemos que a=1, b=2 y f(n)=O(1). Vemos que si aplicamos el caso dos f(n)=O(1) =**Θ**($n^{log_b a}$)=**Θ**($n^{log_2 1}$)=**Θ**(n^0). Por lo tanto T(n)=**Θ**(log n). El algoritmo mejora la complejidad de la propuesta de resolución por fuerza bruta.

Antes de finalizar resta aclarar una cuestión. Podría ocurrir que el punto extremo del polígono convexo no sea solo uno sino infinito. Esto ocurre cuando dos son los vértices más extremos según la dirección de búsqueda. En un polígono convexo estos son vértices contiguos. Y cada punto del segmento que los une sería un punto más extremo. El algoritmo propuesto igualmente encontraría uno de los dos vértices extremos. Con una verificación O(1) se podría verificar si alguno de los dos vértices adyacentes lo es también.

6. Envolvente convexa

Trabajaremos a continuación en el problema llamado envolvente convexa. También conocido como envoltura convexa, cápsula convexa o convex Hull (en inglés). El problema forma parte de la rama de geometría computacional. Parte de un conjunto de puntos en el espacio “d” dimensional y quiere encontrar la figura geométrica más pequeña que contenga a todos ellos. Para esta figura se debe cumplir que para todo par de puntos

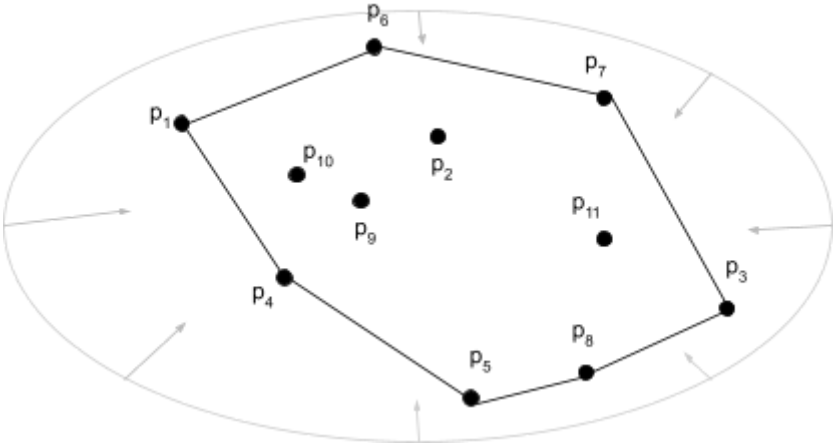
dentro en su interior se pueda trazar un segmento que los una sin abandonarla. Es decir, que tiene que ser convexa.

Este problema tiene una amplia aplicabilidad en gran cantidad de situaciones. Ciertamente por eso explica la cantidad y variedad de propuestas algorítmicas para su resolución y el extenso y recurrente tratamiento del tema en muchos investigadores computacionales. Algunas de estas aplicaciones incluyen planeamiento de recorridos mínimos, reconocimiento de formas, problemas de partición o de separación, procesamiento de imágenes, análisis de formas y muchos más.

Nos concentramos en una versión reducida del problema, solo trabajaremos en el plano: 2 dimensiones. Quedará enunciado de la siguiente manera:

Envolvente convexa (Convex Hull)
Dado un conjunto “P” de “n” puntos en el plano. Deseamos obtener el menor polígono convexo que contenga a todo el conjunto.

Podemos hacer un ejercicio de imaginación (o si cuentan con los materiales y el interés, un ejercicio práctico). Veremos nuestro plano como una plancha de madera y a los puntos como clavos que sobresalen de la misma. Utilizamos una banda elástica y la estiramos de forma que abarque en su interior todos los clavos. Al liberar la banda está intentará volver a su tamaño natural. No podrá hacerlo por encontrar en su camino a los clavos. La banda reposara en tensión sobre un subconjunto de estos. Estos puntos conforman los vértices del polígono convexo o puntos extremos que intentamos encontrar.

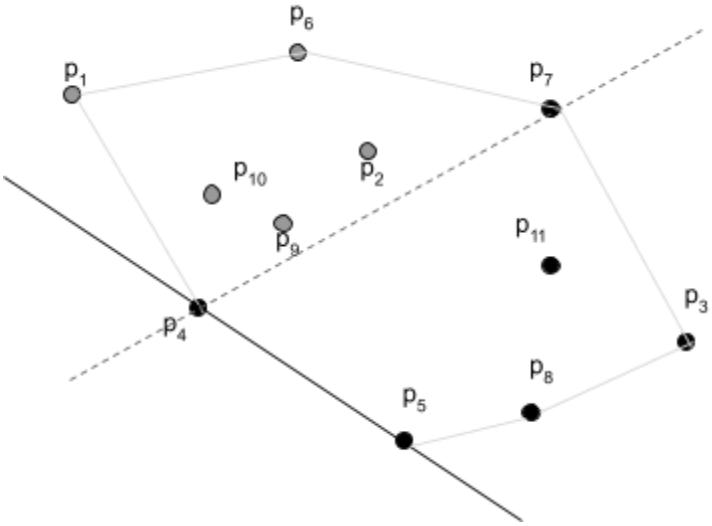


Este enfoque no resulta muy práctico si nuestro material de trabajo es una computadora. Sin embargo, observando atentamente el resultado podemos deducir ciertas propiedades que nos ayuden a construir una solución algorítmica.

Tomemos dos puntos extremos unidos por un segmento del polígono convexo. Extendemos ese segmento a una recta en el plano. Esa recta divide al plano en dos semiplanos. Se puede observar que todos los puntos dentro de la componente convexa quedan en solo uno de los dos semiplanos. Con esto en mente podemos definir una envoltura convexa como la intersección de 3 o más semiplanos que conforman una región no vacía y finita.

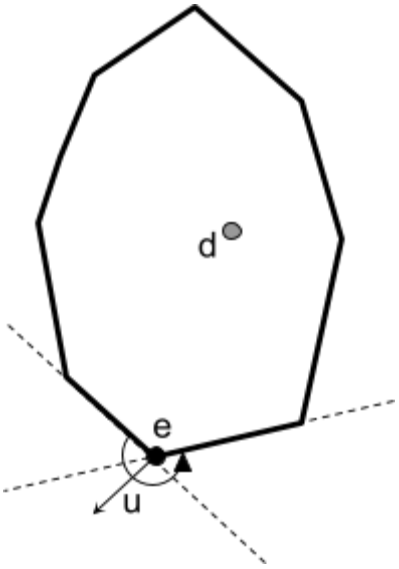
Utilizando esta definición podemos construir nuestro primer algoritmo para generar el convex Hull: mediante fuerza bruta. Probaremos todo par de puntos de nuestro conjunto P. Es decir posibles $\frac{n^2}{2}$ combinaciones. Para cada par construiremos la recta que pasa por ellos. Luego verificamos si los n-2 puntos restantes quedan todos en el mismo semiplano. En caso afirmativo esos dos puntos y el segmento que los une pertenece a la envolvente conexa. Tenemos nuestro primer algoritmo con complejidad temporal $\mathbf{O}(n^3)$

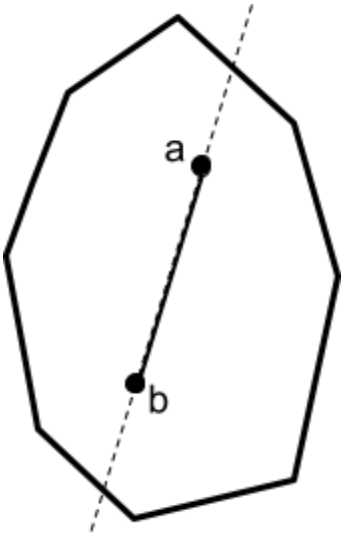
En el ejemplo de la imagen podemos ver que los puntos p_4 y p_5 conforman un semiplano que contiene al resto de los puntos. Sin embargo, para el par de puntos P_4 y p_7, no se puede afirmar lo mismo. El segmento que une p_4 y p_5 pertenece a la envolvente convexa a diferencia del segmento que une p_4 y p_7.



Los siguientes algoritmos que analizaremos utilizan la metodología de división y conquista. Pero previamente deberemos analizar el problema y ciertas propiedades que contiene para poder explicar los principios de su funcionamiento.

Tomemos un punto “e” extremo de la componente convexa. Este punto pertenece a dos segmentos de la componente. Por lo tanto existen mínimamente dos rectas que pasan por él que logran que todos los puntos de P queden en un único semiplano. En rigor son infinitas rectas que pasan por este punto y lo logran. Estas rectas tienen que tener como pendiente un valor entre la pendiente de la primera y segunda recta. Para cada una de esas rectas podemos calcular un vector “u” dirección perpendicular a esta. Si observamos el vector “u” y su comportamiento con todos los puntos extremos de la componente convexa veremos que realiza un giro de 360 grados. Generalizando, podemos observar que para cualquier dirección en el plano existe un punto extremo de P será un punto extremo de la envolvente convexa de P

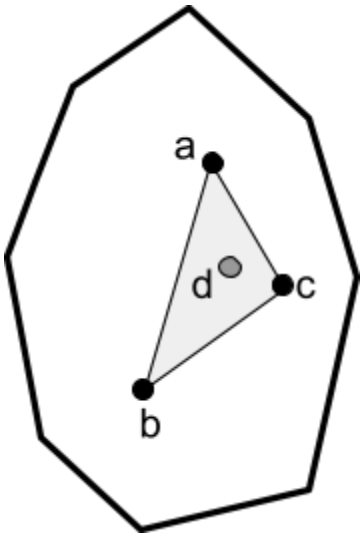




Sean dos puntos “a” y “b” internos a la componente convexa. Entonces el segmento que los une debe estar también dentro del componente. Recordamos la definición de polígono convexo que utilizamos en el problema de puntos extremos en polígono convexo. En ese caso decíamos que no existe una recta que corte en más de 2 puntos al polígono. Si extendemos ese segmento a una recta en algún momento debe entrar y salir de este. Ahí encontramos los dos puntos y por lo tanto el segmento debe ser totalmente interno y no abandonarlo en ningún otro momento.

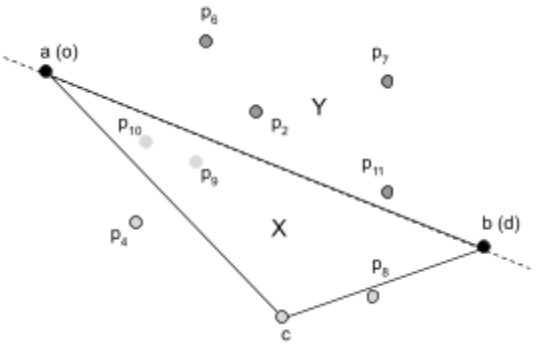
Sea un subconjunto de puntos no colineales $S \subseteq P$. Entonces la cobertura convexa C_s de S está incluida o es igual a la cobertura convexa C_p de P. Al ser los puntos de C_s interiores o de la frontera de C_p, entonces todos sus segmentos son también internos o parte de la frontera de C_p. Por lo tanto se cumple el enunciado.

Sea “d” un punto interno interno de la cobertura convexa C_s de un subconjunto de puntos no colineales S ⊆ P. Entonces “d” es un punto interno de la cobertura convexa C_p de P. Esto se desprende de los dos puntos mostrados anteriormente. Como “d” es interno a C_s (no forma parte de su frontera) entonces es interno a C_p (tampoco forma parte de su frontera)



Ahora estamos listos para el primer algoritmo de división y conquista. Se lo conoce con el nombre de **QuickHull**. Fue creado casi en simultáneo (y de forma independiente) para problemas dos dimensionales por Eddy William[8] en 1977 y por A. Bykat[9] en 1978. Posteriormente fue extendido a más dimensiones por Barber, Dobkin y Huhdanpaa[10]. Su nombre se debe a la semejanza e influencia del algoritmo QuickSort en su elaboración.

En primer lugar seleccionamos dos vectores dirección “u” arbitrarios. Con cada uno de ellos obtendremos en O(n) cual es el punto más extremo en estas direcciones. Los llamaremos “a” y “b” respectivamente. Sabemos que ellos deben pertenecer a los puntos extremos de la envolvente convexa de P.



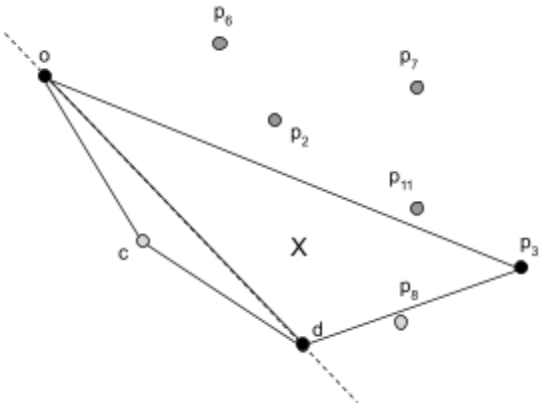
Construimos una recta con “a” y “b”. Estos dos puntos conforman dos semiplanos “X” e “Y”. Nos referiremos a los semiplanos según quedan a la derecha o izquierda del segmento conformado por un punto de origen y otro de destino. Así es que el semiplano “X” queda a la derecha del segmento “a-b” y el semiplano “Y” queda a la derecha del segmento “b-a”. Para cada uno de

[8]A New Convex Hull Algorithm for Planar Sets, Eddy William F.,1977, ACM Trans. Math. Softw. 3 pp 398-403.
[9]Convex hull of a finite set of points in two dimensions, BYKAT, A, 1978, Inf. Process. Lett. 7, pp 296 -298.
[10]The quickhull algorithm for convex hulls, Barber C. Bradford; Dobkin David P.; Huhdanpaa Hannu., 1 de diciembre de 1996, ACM Transactions on Mathematical Software 22 (4): pp 469-483

los semiplanos verificamos que puntos de P están en ellos. Ahora se separa el problema en dos subproblemas que resolveremos cada una de forma recursiva.

El subproblema recibe el segmento definido por dos puntos (origen y destino) y un conjunto de puntos a analizar. Si el conjunto de puntos está vacío, el proceso finaliza. Significa que ningún punto extremo se puede obtener ahí y que el segmento pertenece a la envoltura convexa. De lo contrario se procede a buscar al punto "c" más distante a la derecha del segmento. Al encontrarlo tenemos un punto extremo que debe pertenecer a la componente convexa. Además este punto conforma el triángulo con los dos puntos anteriores que podemos llamar "o-c–d". El triángulo es la figura convexa más simple y será interior (o igual) a la envoltura convexa de P. Por lo tanto todos los puntos que queden dentro de ese triángulo no pueden ser extremos y pueden ser desechados. El resto de los puntos podrían llegar a ser extremos.

Al subconjunto de puntos que se encuentra en el semiplano a la derecha de la recta que pasa por el segmento o-c los llamamos P1. Similarmente, llamamos P2 a los puntos del semiplano a la derecha de la recta que pasa por el segmento c-d. Con cada uno de esos semiplanos y sus puntos de origen y destino repetimos recursivamente el proceso hasta finalizar.



A continuación el pseudocódigo que resume el proceso:

QuickHull

Sea P un conjunto de n puntos en el plano
Sea CH = ∅

QuickHull(P):

Seleccionar "a" punto de P como punto extremo con dirección "más a la derecha"
Seleccionar "b" punto de P como punto extremo con dirección "más a la izquierda"

CH = a → b → a

```
        Seleccionar Px los puntos de P a la derecha del segmento “a-b”
        Seleccionar Py los puntos de P a la derecha del segmento “b-a”

        ObtenerCH(a,b,Px)
        ObtenerCH(b,a,Py)

ObtenerCH(o,d,P):

        Seleccionar “c” punto de P como punto extremo con dirección “más a la derecha”
        Agregar en CH :  c entre o y d

        Sean P1 los puntos de P a la derecha de o-c
        Sean P2 los puntos de P a la derecha de c-d
        Los puntos en P - P1 - P1 son aquellos desechados por estar en el triángulo o-c-d

        ObtenerCH(o,c,P1)
        ObtenerCH(c,d,P2)
```

En resumen estamos realizando la construcción del convex hull utilizando las propiedades que describimos. Este proceso constructivo se realiza dividiendo el problema en dos subproblemas. Cada uno de estos es menor que volvemos a dividir recursivamente en dos partes. Repetimos hasta que no queden puntos, nuestro caso base. En cada subproblema realizamos procesos de comprobación por cada punto. Esos procesos son $\mathbf{O}(1)$. Por lo que el trabajo realizado es lineal con la cantidad de puntos por subproblema.

No podemos expresar la recurrencia como $T(n)=2T(n/b) + \mathbf{O}(n)$. Dado que nuestros dos subproblemas no tienen la misma cantidad de puntos. Podrían tener la mitad cada uno. También podría pasar que uno no tenga ningún punto y el otro todo el resto. Entonces podemos expresar la recurrencia como $T(n) = T(a) + T(b) + \mathbf{O}(n)$ con $0 \leq a+b<n$. Si justo dividimos los puntos en partes iguales (a=b) y no desechamos ninguno conseguimos el valor O(nlogn) conocido del mergesort. Pero podría pasarnos el peor caso posible. Corresponde al caso donde en cada iteración no se elimina ningún punto y que además tengamos un subproblema vacío y el otro con los n-1 puntos siguientes. En ese caso tendríamos, trabajando sobre la recurrencia:

$T(n) = T(n-2) + T(0) + \mathbf{O}(n)$ (En la primera iteración tomamos dos puntos extremos, luego solo uno)

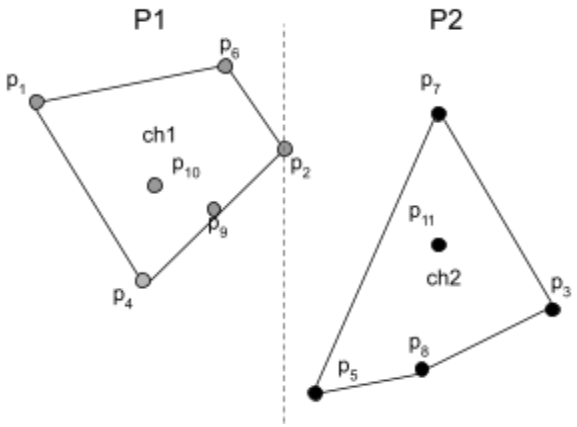
T(n-2) = T(n-3) + T(0) + **O**(n-2)

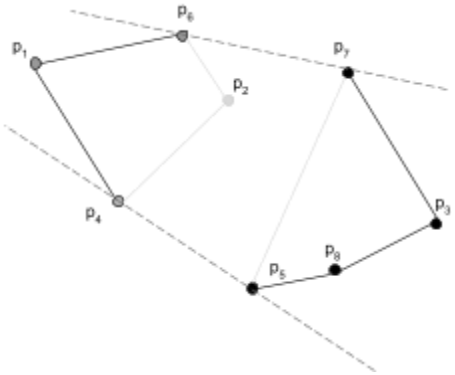
…

T(0)=**O**(1).

Unificando T(n) = **O**(n) + **O**(n-2) + **O**(n-3) + … (n-1)*T(1). Que unificando podemos ver que corresponde a T(n)= **O**(n^2). Tal cual se comporta el quicksort en su peor caso. De todas formas redujimos la complejidad lograda con el método por fuerza bruta.

La siguiente metodología que veremos corresponde a la propuesta por **Preparata y Hong** [11]. No recibe un nombre un nombre específico en la bibliografía general. Se la suele referenciar simplemente como "utilizando división y conquista". En este caso la idea es dividir los puntos en dos subproblemas. Para la división se ordenan los puntos por una coordenada (por ejemplo la X) y se los parte en dos: P1 y P2. Recursivamente se vuelve a dividir hasta llegar a 5 o menos puntos. Corresponde al caso base donde se puede utilizar el algoritmo por fuerza bruta (u otro seleccionado) para construir una envolvente convexa. Resueltos los dos subproblemas obtenemos dos componentes convexas: ch1 y ch2. Se procede a un proceso de combinación entre las dos para hallar una que contenga a ambas.



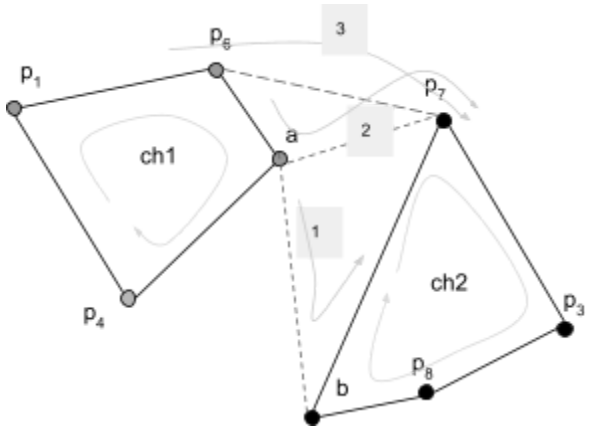


El proceso de combinación busca encontrar dos rectas tangentes a los polígonos. Una superior y la otra inferior. Cada recta pasa por un punto de cada una de las componentes. Dejan al resto de los puntos en solo una de los semiplanos que define. Los segmentos de estas rectas que entre los dos puntos de los componentes convexos son aquellos que unen a estos

[11] Convex Hulls of Finite Sets of Points in Two and Three Dimensions, F. P. Preparata; S. J. Hong, Febrero 1977, Communications Of the ACM Volume 20 Number 2

para conformar la nueva envoltura que incluya a todos los puntos. Algunos de los puntos incluidos en cada una de las componentes convexas posiblemente pasen a ser interiores a la componente contenedora y deben ser desechados.

Encontrar las rectas tangentes por fuerza bruta implicaría tomar pares de puntos de cada componente, calcular la recta y verificar de qué lado queda cada uno de los puntos. Volvemos al $\mathbf{O}(n^3)$ que queremos batir. Por lo que buscaremos una alternativa más eficiente.



Buscaremos al punto “a” más a la derecha del convex hull ch1. Similarmente, realizaremos la búsqueda del punto “b” más a la izquierda del convex hull ch2. Construiremos el segmento a-b como punto de partida de la construcción de las rectas tangentes. Recordemos la propiedad de los polígonos convexos. Si recorremos el perímetro del mismo, el giro siempre será en el mismo sentido. Las tangentes que unen las componentes convexas en una nueva deben también cumplir con lo mismo en la componente resultante.

Mostraremos el proceso para encontrar la tangente superior. sin embargo es similar para la inferior pero modificando los sentidos. Partimos del segmento a-b. obtenemos el segmento de ch1 que llega a “a” en el sentido de recorrido horario. También obtenemos el segmento de ch2 posterior a “b” en el mismo sentido. Tenemos ahora 3 segmentos. El pasaje entre ellos 3 no debe cambiar de sentido de recorrido. Si no cambia, el segmento de unión corresponde al de la tangente superior y debe pertenecer a la componente convexa unificada. Si no, vemos cuál de los segmentos cambia el sentido. Puede ser entre el primero y el segundo o el segundo y el tercero. Si es entre el primero y el segundo entonces reemplazamos el punto “a” con el punto donde inicia el primer segmento. Si es entre el segundo y el tercero entonces reemplazamos el punto “b” con el punto donde termina el tercer segmento. Volvemos a repetir el proceso hasta que se cumpla el mantenimiento del sentido.

Una vez obtenidos los segmentos superiores e inferiores resta el proceso de unión entre ambos. Para hacerlo quitamos los puntos de las componentes que quedaron entre los dos segmentos tangentes e insertamos dentro de la lista de puntos de ch1 los puntos ch2 restantes para que quede la componente convexa completa.

Todos estos procesos se pueden realizar en tiempo lineal. La búsqueda de los puntos extremos de las componentes la podemos realizar en O(logn). La búsqueda de las tangentes recorre los puntos quitando en el proceso de a un punto a la vez. Y se usa para la comparación de los sentidos un cálculo O(1) en función de los puntos. En conclusión es O(n). Y unir los resultados también corresponde a O(n).

A continuación el pseudocódigo que resume el proceso:

ConvexHull - Preparata

```
Sea P un conjunto de n puntos en el plano ordenados por coordenadas x

CalcularCH(P):
        Si |P| <= 5
                Retornar CH construida por fuerza bruta

        Sean P1 los primeros |P|/2 puntos
        Sean P2 los últimos|P|/2 puntos

        ch1 = CalcularCH(P1)
        ch2 = CalcularCH(P2)

        Retornar Unir(ch1,ch2)

Unir(ch1, ch2)
        Sea a punto más a la derecha de ch1
        Sea b punto más a la izquierda de ch2
        buscar tangente superior entre ch1 y ch2 partiendo de a y b
        buscar tangente inferior entre ch1 y ch2 partiendo de a y b

        Quitar de ch1 los puntos entre la tangente superior e inferior
        Quitar de ch2  los puntos entre la tangente superior e inferior
        Crear ch con puntos de ch1 y ch2 unidos por tangentes.
```

```
    Retornar ch
```

Nos encontramos ante un proceso similar al mergesort. Tenemos una recursión donde dividimos en dos subproblemas con la mitad de elementos cada uno. El proceso de merge se realiza en tiempo lineal. Expresamos la relación de recurrencia como $T(n)=2T(n/2) + \mathbf{O}(n)$. Utilizando el teorema maestro obtenemos que la complejidad temporal es $\mathbf{\Theta}(n \log n)$.

7. Par de puntos más cercanos en el plano

El problema que veremos a continuación es uno de interés en geometría computacional. Se conoce como el problema del par de puntos más cercanos ("closest pair of points problem"). La idea es sencilla dado un conjunto de puntos queremos encontrar al par de ellos que se encuentren a menor distancia entre sí. Los puntos pueden representar diferentes conceptos. Algunos ejemplos podrían ser, miradores en una ruta panorámica, Clubes deportivos en una ciudad, estrellas en la galaxia, personas en un estudio genético. Dependiendo del concepto tendremos diferentes espacios dimensiones, en nuestros ejemplos 1,2,3 y "d" respectivamente. Finalmente el concepto de distancia también es diferente de acuerdo al escenario. Corresponderá a una función que dados dos conceptos (puntos) retorna un valor numérico. A menor valor más cercanos los puntos. Algunos ejemplos de distancia podrían ser: Euclídea, Manhattan, Coseno, Hamming, Minkowski, Chebyshev, etc.

Sea cualquier dimensión del problema y la función distancia podemos resolver el problema por fuerza bruta simplemente comparando todos los puntos entre sí. En ese caso la complejidad será $\mathbf{\Theta}(n^2)$. Por lo tanto intentaremos encontrar una solución superadora a esta cuando sea posible.

En primer lugar restringimos el tipo de espacio que utilizaremos a los métricos. Estos corresponden a aquellos donde la distancia d es una función real, que satisface los siguientes axiomas:

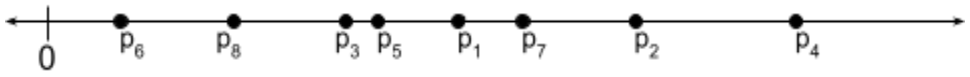
1. $d(x, y) \geq 0 \ \forall x, y \in X$
2. $d(x, y) = 0 \Leftrightarrow x = y$
3. $d(x, y) = d(y, x) \ \forall x, y \in X$

4. $d(x, z) \leq d(x, y) + d(y, z)\ \forall x, y, z \in X$

Podemos resumir: una función no negativa, simétrica y que cumple con la desigualdad triangular.

Si el espacio no es métrico generalmente podremos resolver el problema por fuerza bruta sin otra alternativa.

Una vez establecida esta restricción, comencemos analizando el caso más simple: sólo una dimensión. En ese caso representaremos a los diferentes puntos (o elementos) distribuidos sobre la recta de números reales. Se muestra un posible caso a analizar de n=8 puntos. Si comparo todos con todos se obtendrá que los puntos p_3 y p_5 son los más cercanos entre todos.

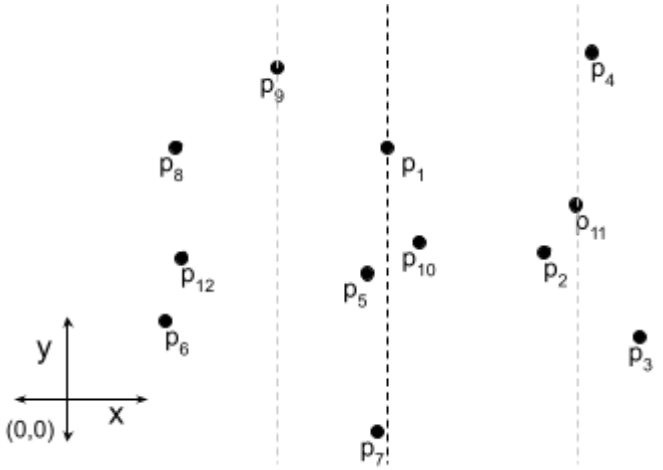


Al ser la distancia métrica, se cumple la desigualdad triangular. Esto implica que - por ejemplo - la distancia del punto p_3 a p_1 tiene que ser mayor a la distancia de p_3 a p_5. Y lo mismo la distancia entre p_5 y p_8 tiene que ser mayor a la distancia de p_5 a p_3. En conclusión si un punto pertenece al par de los puntos más cercanos, su compañero tendrá que ser de forma o el anterior o posterior a él. Si tengo los puntos ordenados, encontrar los puntos más cercanos es simplemente iterar por los puntos y calcular la distancia con el próximo. Claramente un proceso $\Theta(n)$. Si se deben ordenar entonces tendremos que el proceso completo será $\Theta(n\log n)$. De esa forma logramos una mejora en la complejidad final del proceso.

Aumentemos una dimensión. Trabajaremos en el plano de los reales. Veremos que esto complejiza un poco las cosas. Antes de continuar, formalicemos el problema.

Par de puntos más cercanos en el plano
Dado un conjunto de “n” puntos diferentes. Cada punto definido como un par $p_i = (x_i, y_i)$ en el plano. La función $d(p_i, p_j)$ que retorna el valor de la distancia euclídea entre los puntos p_i y p_j. Queremos saber qué par de puntos son los más cercanos entre sí, de todo el conjunto.

En este caso no podemos ordenar por alguna de las dimensiones, dado que dos puntos pueden estar tomando una dimensión cerca, pero lejos en una segunda dimensión. Lo vemos en el ejemplo en la siguiente imagen. Si realizamos un ordenamiento por la coordenada x, p_1 y p_7 están muy cercanos, pero lejos visualmente. Si ordenamos según la coordenada y, p_5 y p_2 están muy cercanos pero igualmente lejos visualmente.



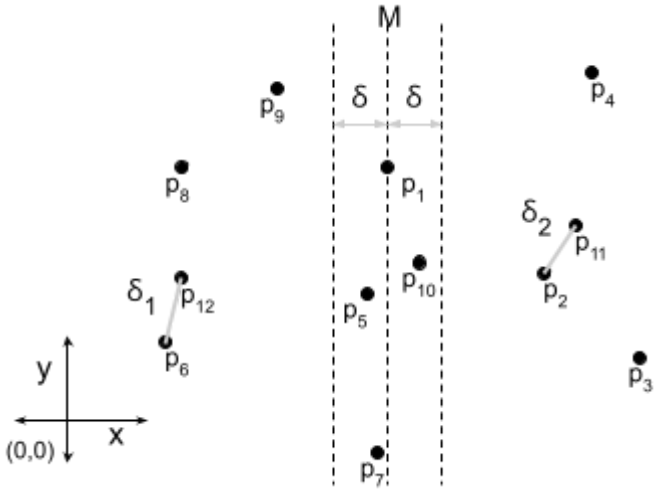
Utilizando división y conquista esta separación tal vez puede ser un camino de partida para la construcción de un buen algoritmo. Mostraremos el algoritmo propuesto por Bentley y Shamos[12] (aunque en su mismo paper los autores atribuyen el descubrimiento del procedimiento a H. R. Strong). Dividamos los puntos en dos subconjuntos tomando como separador el punto medio según la coordenada x. Cada conjunto nuevo tendrá la mitad de los puntos y buscaremos en ellos el par mínimo. Esa división la podemos hacer recursivamente. El proceso de combinación consiste en quedarnos con el par de puntos de menor distancia entre los dos subproblemas. Si expresamos esta idea mediante una relación de recurrencia tendremos $T(n) = 2T(n/2) + \mathbf{O}(1)$. Que utilizando el teorema maestro corresponde a $\mathbf{\Theta}(n\log n)$.

Esta solución tiene un problema fundamental: el par de puntos de distancias mínimos pueden quedar en diferentes subproblemas. De esta forma no lo encontraríamos y nuestro resultado no sería óptimo. Por lo tanto, al momento de la combinación de los

[12] "Divide-and-conquer in multidimensional space", Bentley, J. L., & Shamos, M. I., 1976, Proceedings of the Eighth Annual ACM Symposium on Theory of Computing - STOC '76.

subproblemas se debe considerar un tercer caso: que el par de puntos corresponda a uno de cada uno de los subconjuntos. Resolver esta situación será crucial para mantener la complejidad deseada.

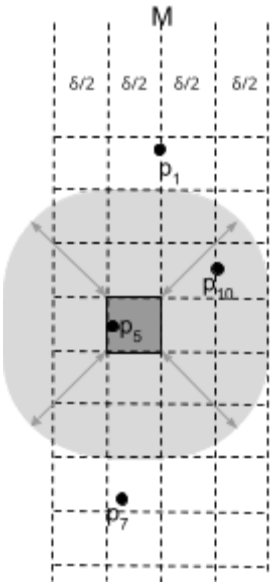
Una primera intuición es considerar que no todos los puntos son candidatos al tercer caso. Llamaremos δ_1 la distancia mínima entre puntos de la primera mitad de puntos y δ_2 al equivalente en la segunda mitad. "M" a la recta perpendicular al eje X que pasa por el punto que divide a los dos conjuntos. Buscamos ver si existe un punto en el primer conjunto y otros del segundo que estén a menos distancia de $\delta=\min(\delta_1,\delta_2)$. Vemos que estos puntos candidatos no pueden estar a una distancia mayor de M justamente que δ. Esto determina una región en el plano donde buscaremos los candidatos. Esta búsqueda la podemos hacer en $\Theta(n)$. En la imagen siguiente se pueden observar para la misma instancia del problema anterior las últimas definiciones.



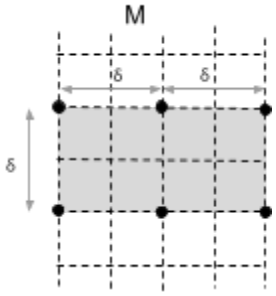
Podrá parecer que con está decisión se evita tener que comparar todos los puntos de un conjunto con todos los del otro. Sin embargo, nada evita que absolutamente todos los estos se encuentren dentro de la región en el plano de puntos candidatos. De ocurrir esa comparacion seria $\mathbf{O}(n^2)$, La relación de recurrencia se transforma en $T(n) = 2T(n/2) + \mathbf{O}(n^2)$. Utilizando el teorema maestro corresponde a $\Theta(n^2)$ y por lo tanto estamos regresando a la cota de complejidad de fuerza bruta.

Por fortuna, con un análisis más riguroso podemos evitar caer en este problema. Comenzamos dividiendo teóricamente el área de candidatos en cuadrados de lado $\delta/2$. al

resultado lo podemos ver como una grilla con 4 celdas por fila. Cada celda puede solo puede contener uno o ningún punto. Dos puntos en la misma celda implicaría que la distancia entre ellos es menor a δ. Un absurdo puesto que obtuvimos ese valor como la distancia mínima entre los puntos de los dos subproblemas que resolvimos. Si nos paramos dentro de una celda y nos extendemos a una distancia δ de esta observaremos que sólo un subconjunto de 19 celdas pueden contener puntos a la distancia que estamos buscando (en rigor serían menos, luego volveremos sobre este punto). En conclusión no es necesario comparar con todos los puntos, solo con aquellos en su región. Vamos a considerar los puntos candidatos ordenados según su coordenada Y. Si nos paramos en un determinado punto solo otros 11 puntos posteriores (en 1 celda cada uno) podrían estar a una distancia menor a δ. De forma análoga solo otros 11 puntos anteriores podrían estar a menor distancia que la buscada. Puede entonces iniciar en el primer punto (aquel con la coordenada Y mayor) y compararlos con los 11 puntos siguientes. Claramente no tiene anteriores. Luego al pararse en el segundo punto no es necesario comparar con los anteriores y solo requiere comparar nuevamente con los 11 puntos siguientes.



El resumen del proceso para obtener una posible distancia mínima entre puntos de diferente subconjunto es el siguiente. Restringimos como puntos candidatos a aquellos que están a distancia de la recta M a no más de δ. A la lista obtenida la ordenamos según su coordenada Y. Para cada punto lo comparamos con los 11 puntos siguientes. Este proceso tendrá en conjunto una complejidad de $\mathbf{O}$(nlogn). Nuestra recurrencia quedará expresada como T(n) = 2T(n/2) + $\mathbf{O}$(nlogn). No se puede usar el teorema maestro para obtener la complejidad, aunque por otro métodos esta nos quedará $\mathbf{\Theta}$(nlog^{2}n). Que es mejor que utilizar el método de fuerza bruta. No obstante, podemos mejorarlo con algo de ingenio.



Una (aparente) pequeña mejora que podemos realizar es reducir las 11 comparaciones a solo 7. Puede parecer mínimo, pero con una cantidad de puntos grande en la implementación se termina sintiendo. En el análisis anterior determinamos que solo había un punto por celda. Pero

no todas las celdas contiguas pueden estar ocupadas. De cada lado de M, los puntos están separados en δ_1 y δ_2 respectivamente (que son igual o mayores que δ). Supongamos que $\delta_1 = \delta_2 = \delta$. En ese caso si consideramos un cuadrado de $\delta x \delta$ solo puede contener como mucho 4 puntos. Esto ocurre de ambos lados de M. Por lo que para un determinado candidato, solo podría tener 3 puntos de su lado y otros 4 del otro. Con solo ellos se debe comparar para verificar su distancia.

Otra mejora - en este caso significativa - es evitar tener que reordenar continuamente los puntos por la coordenada Y. Si bien al realizar la división de los problemas es importante que los puntos estén ordenados por la coordenada X, luego esto deja de ser trascendente. Al regresar de los subproblemas, en la etapa de combinación es importante que el ordenamiento sea por la coordenada Y. Si cada subproblema retorna sus puntos ordenados por este último criterio, entonces tendremos dos subconjuntos ordenados y con un merge en $\mathbf{O}(n)$ construiremos la lista total ordenada (Conceptualmente similar a conocido algoritmo mergesort). En el caso base ordenar dos o tres puntos es trivial.

Podemos ver el pseudocódigo de la solución completa a continuación

Par de puntos más cercanos en el plano - División y conquista

Sea P un conjunto de n puntos en el plano

masCercanos(S):
    Si |S| ≤ 3
        Obtener p_1,p_2 par de puntos más cercanos mediante función d
        Ordenar S según coordenada Y
        Retornar p_1, p_2, S

    Sea M la recta vertical que pasa por el punto medio de S
    Sea A los primeros |S|/2 puntos
    Sea B los últimos |S|/2 puntos

    (p_{a1}, p_{a2}, A) = masCercanos(A)
    (p_{b1}, p_{b2}, B) = masCercanos(B)

    δ = mínimo entre δ_1=d(p_{a1}, p_{a2}) y δ_2=d(p_{b1}, p_{b2})>
    Sea m_1, m_2 los puntos de menor distancia entre (p_{a1}, p_{a2}) y (p_{b1}, p_{b2})

    Sea S' el resultado de mergear A y B según coordenada Y

```
        Construir C candidatos con los puntos de S' a igual o menor distancia δ de recta M

        Por cada punto c1 de C
                Por cada punto c2 posterior a c1 en C en no más de 7 posiciones
                        Si d(c1,c2) < δ
                                Establecer c1 y c2 como m1 y m2. y δ=d(c1,c2)

        Retornar (m1,m2, S')

Ordenamos P según su coordenada X
masCercanos(P)
```

Tenemos el costo inicial $\mathbf{O}$(nlogn) de ordenar los puntos por coordenada X. Dividimos cada problema en dos subproblemas con la mitad de los puntos y definiendo con la coordenada X del punto medio la recta M. En el caso base retornamos la distancia mínima entre dos o tres puntos y los puntos ordenados por coordenada Y. Todo esto en tiempo $\mathbf{O}$(1). Con los resultados de los dos subproblemas tenemos los pares de puntos mínimos de ellos y dos listado de puntos ordenados por la coordenada Y. Realizamos un merge manteniendo el ordenamiento. Realizamos una selección de aquellos puntos candidatos (a no más distancia de δ de la recta M) y finalmente en forma lineal comparamos cada punto con un número constante de puntos siguientes. Nuestra recurrencia es de la forma $T(n) = 2T(n/2) + \mathbf{O}(n)$. Utilizando el teorema maestro nos quedará $\mathbf{\Theta}$(nlogn) consiguiendo nuestro el objetivo con el que iniciamos.

En el mismo paper y en el libro "Computational Geometry An Introduction"[13] con más didáctica, Shamos explica cómo llevar este problema a un espacio de "d" dimensiones. Lo hacen también con división y conquista. En los subproblemas van progresivamente reduciendo las dimensiones así como dividiendo los puntos. De igual manera logran una cota de complejidad de $\mathbf{\Theta}$(nlogn).

8. Multiplicación rápida: Karatsuba

Multiplicar dos números enteros de n dígitos cada uno es un procedimiento que dominamos con soltura. Con valores pequeños es un proceso mental automático. Si el

[13] Computational Geometry: An Introduction, Franco P. Preparata y Michael Ian Shamos,1985, Springer

requerimiento es con cantidades más grandes, nos podemos ayudar con lápiz y papel.... o le confiamos el resultado a una aplicación de nuestro teléfono celular.

Aprendemos la mecánica de la multiplicación en los primeros años de escolaridad. El concepto es sumar "P" veces el número "Q" (suma reiterada) y lo expresamos mediante notación matemática como P x Q. Cada vez que nos encontramos con la necesidad de este cálculo, utilizamos un procedimiento o algoritmo de multiplicación al que llamaremos "tradicional". Utilizamos para representar los números 10 símbolos que nos llegaron de la cultura árabe (que a la vez se acepta que estos lo refinaron del “lejano oriente”). Estos símbolos representan diferentes valores de acuerdo a su posición (unidades, decenas, centenas.... etc). Nos apoyaremos en un conjunto de operaciones elementales que seguramente todos hemos memorizado en algún momento de nuestras vidas: las tablas de multiplicar del 0 al 9. Emparejamos ambos números y realizamos un total de O(n2) multiplicaciones elementales y un número similar de sumas básicas.

Este método no es el único conocido en la historia. Gracias al papiro de Rhind, descubierto por Alexander Henry Rhind en 1858 en Luxor (Egipto) conocemos como multiplicaban los antiguos egipcios. También se ha podido reconstruir cómo multiplicaban los babilonios (que usaban 60 símbolos para la numeración). Además existen otros mecanismos que sobreviven en nuestros días como la multiplicación japonesa (o china), el método hindú (o de celdillas) entre otros. Todos estos métodos tienen en común que comparten la misma complejidad.

No es de extrañar que cuando se comienza a teorizar sobre el algoritmo de multiplicar y su complejidad muchos hayan supuesto que eso era lo mejor que se podría lograr. Entre ellos sobresale en la historia Andrey Kolmogorov. Este importante matemático de origen Sovietico postuló la conjetura n^2 también conocida como Conjetura Kolmokorov. La misma afirma que ningún algoritmo de multiplicación posible puede requerir menos de n^2 operaciones elementales. Es decir que tienen una cota de $\mathbf{\Omega}(n^2)$. Consideraba la antigüedad del problema (milenios) y la necesidad económica que mueve a la humanidad para encontrar caminos que reduzcan su esfuerzo. En síntesis: Si existiese mejor método, ya tendría que haber sido encontrado.

Una conjetura corresponde a una afirmación que se supone cierta pero que no ha podido ser ni probada ni refutada. En caso de ser probada pasa a ser un teorema y se puede utilizar dentro del andamiaje matemático (si se refuta pasa a ser desechada).

Resultó falsa. En otoño de 1960 en un seminario de problemas matemáticos en cibernética, dictado en la Facultad de mecánica y matemáticas de la universidad de Moscú donde Kolmokorov la pronunció por última vez. Dentro de los presentes se encontraba un joven estudiante de 23 años a quien le bastó una semana para refutar: Anatoly Karatsuba. Le presenta a Kolmogorov su hallazgo. Este último agitado se muestra entusiasmado y dedica tiempo de su disertación a darle difusión.

Dos años después (1962) se publicaba "Multiplication of Many-Digital Numbers by Automatic Computers"[14] Los autores de este paper eran A. Karatsuba y Yu. Ofman. Extrañamente, Karatsuba no tuvo nada que ver con la publicación y se enteró de la misma cuando ya estaba impresa. Esta historia la cuenta años después[15] en otra publicación esta vez propia.

Formalicemos el problema. En lugar de representación decimal, utilizaremos representación binaria. Esta es similar a la utilizada por nuestras computadoras modernas. Sin embargo lo mismo vale para cualquier base.

Multiplicación de dos números de n bits cada uno
Dado A y B números de n bits. Deseamos hallar el número resultante de multiplicar A por B.

La solución de karatsuba comienza descomponiendo cada número en dos partes. Por un lado la mitad de sus cifras más representativas y por otro las menos representativas.

$A = a_1 * 2^{n/2} + b_0$

[14] "Multiplication of Many-Digital Numbers by Automatic Computers", A. Karatsuba; Yu. Ofman, 1962, Proceedings of the USSR Academy of Sciences. 145: 293–294.
[15] "The Complexity of Computations", A. A. Karatsuba, 1995, Proceedings of the Steklov Institute of Mathematics. 211: 169–183

$B = b_1 * 2^{n/2} + v_0$

Cuando realicemos la multiplicaciones estaremos realizando:

$A*B = (a_1 * 2^{n/2} + b_0) * (b_1 * 2^{n/2} + v_0)$

$= (a_1 * b_1)* 2^n + (a_0 * b_1 + a_1 * b_0) * 2^{n/2} + (a_0 * b_0)$

Como se puede ver - luego de aplicar la propiedad distributiva - nos quedamos con 4 multiplicaciones de n/2 bits cada una. No contamos como una de ellas a las asociadas con los términos 2^n y $2^{n/2}$. Estas corresponden a corrimientos (shits) del valor calculado. Podemos ver a cada una de las multiplicaciones como un nuevo subproblema que podremos solucionar recursivamente. El caso base corresponde a la multiplicación de dos números de 1 bit. El problema de combinar los resultados corresponde a los shits y las sumas. La cantidad de sumas elementales está acotada por el número de bits y por lo tanto es un proceso $\mathbf{O}(n)$. Con eso construimos un algoritmo para resolver la multiplicación utilizando división y conquista. Antes de continuar analicemos la relación de recurrencia resultante:

$T(n) = 4T(n/2) + \mathbf{O}(n)$
$T(1)=\mathbf{O}(1)$

Utilizando el teorema maestro tenemos que a=4, b=2 y $f(n)=\mathbf{O}(n)$. Aplica el caso 1. $f(n)=\mathbf{O}(n) =\mathbf{O}(n^{\log_b a-e})=\mathbf{O}(n^{\log_2 4-e})=\mathbf{O}(n^{2-e})$ con por ejemplo 0,1. Por lo tanto $T(n)=\mathbf{\Theta}(n^2)$. No tenemos mejoras frente al proceso original. Sin embargo…

La genialidad de Karatsuba fue darse cuenta que podía reducir a solo 3 los subproblemas a calcular. Copiamos nuevamente la multiplicación para tenerla a mano

$A*B = (a_1 * b_1)* 2^n + (a_0 * b_1 + a_1 * b_0) * 2^{n/2} + (a_0 * b_0)$

Llamemos a los 4 subproblemas (los últimos dos juntos):

$Sub_1 = (a_1 * b_1)$

$Sub_2 = (a_0 * b_0)$

$Sub_{3\_4} = (a_0 * b_1 + a_1 * b_0)$

Se puede plantear la siguiente igualdad:

(1) $(a_0 + a_1) * (b_0 + b_1) = (a_0 * b_0) + (a_0 * b_1 + a_1 * b_0) + (a_1 * b_1)$

Reemplazamos con los nombres que le dimos a los subproblemas de nuestro problema original

$(a_0 + a_1) * (b_0 + b_1) = Sub_2 + Sub_{3\_4} + Sub_1$

Despejando Sub_{3_4}:

$Sub_{3\_4} = (a_0 + a_1) * (b_0 + b_1) - Sub_1 - Sub_2$

Es decir que podemos evitar calcular los subproblemas 3 y 4 utilizando los resultados de los subproblemas 1 y 2. Más un nuevo subproblema $Sub' = (a_0 + a_1) * (b_0 + b_1)$ donde tengo solo una multiplicación de n/2 bits. En total nos quedarán 3 subproblemas de n/2 bits. Además de algunas sumas adicionales que no teníamos antes.

Lo expresamos mediante pseudocódigo:

Multiplicación rápida: Karatsuba

```
Sea X y Y dos números de n bits

Karatsuba(a,b):
        Si a y b se puede multiplicar de forma elemental
                Retornar a*b
        Expresamos a como a1 * 2^(n/2) + a0
        Expresamos b como b1 * 2^(n/2) + b0
        Calcular c = a1 + a0
        Calcular d = b1 + b0

        Sub' = Karatsuba (c ,d)
        Sub1 = Karatsuba (a1 , b1 )
        Sub2 = Karatsuba (a0 , b0 )

        Retornar Sub1* 2^n + (Sub' - Sub1- Sub2) * 2^(n/2) + Sub2
```

Karatsuba(X,Y)

Ahora si, veamos que tenemos una mejora comparado con la multiplicación tradicional. La relación de recurrencia queda:

T(n) = 3T(n/2) + $\mathbf{O}$(n)
T(1)=$\mathbf{O}$(1)

Utilizando el teorema maestro tenemos que a=3, b=2 y f(n)=$\mathbf{O}$(n). Aplica el caso 1. f(n)=$\mathbf{O}$(n) =$\mathbf{O}(n^{log_b a-e})$=$\mathbf{O}(n^{log_2 3-e})$=$\mathbf{O}(n^{1.58496-e})$ con por ejemplo 0,1. Por lo tanto T(n)=$\mathbf{\Theta}(n^{1,59})$ y la conjetura n^2 por lo tanto es falsa.

Es importante recordar que el análisis asintótico de complejidad asegura que a partir de cierto tamaño de la instancia hasta el infinito el comportamiento del algoritmo estará acotado por cierta función matemática multiplicada por una constante. Sin embargo para valores menores a ese tamaño esto no necesariamente es así. Eso se puede ver claramente en el algoritmo de karatsuba donde en instancias pequeñas se comporta peor que el algoritmo tradicional de multiplicación que es $\mathbf{\Theta}(n^2)$. La causa son las operaciones $\mathbf{O}$(n) - como las sumas y corrimientos - que están presentes en los subproblemas. Pasado cierto tamaño dejan de ser preponderantes. Es por esto que los principales lenguajes de programación no lo utilizan para multiplicar números enteros de pocos bits. Pero estos mismos lenguajes en sus librerías aritméticas de grandes números suelen utilizar este algoritmo u otro que se construyó mejorando aún más la complejidad.

También por este último motivo es que las implementaciones de karatsuba no consideran como caso base la multiplicación de números de solo un bit. Sino que llegados a un determinado tamaño del subproblema en vez de seguir la recursión, multiplican y retornan el resultado utilizando la multiplicación tradicional. Este comportamiento es común en la mayoría de los programas que utilizan división y conquista y redundan en una mejora en los tiempos reales de ejecución para la resolución del problema.

El descubrimiento de este algoritmo provocó un auge de los algoritmos que fueron conocidos como “fast”. En los años siguientes se fueron descubriendo otros algoritmos asintóticamente más eficientes. En el año 2019, David Harvey y Joris van Der Hoeven

lograron construir un algoritmo para multiplicar con una eficiencia de $\mathbf{O}(n \log n)$[16]. En la siguiente tabla un resumen de algunos de los más conocidos.

Año	Algoritmo	Complejidad
1960	Karatsuba	$\mathbf{O}(n^{1,58})$
1963	Toom Cook	$\mathbf{O}(n^{1,46})$
1971	Schonhage Strassen[17]	$\mathbf{O}(n \log n \log \log n)$
2007	Furer[18]	$\mathbf{O}(n \log n\ 2^{O(\log n)})$
2008	DKSS[19]	$\mathbf{O}(n \log n\ 2^{O(\log n)})$
2015	Covanov and Thomé[20]	$\mathbf{O}(n \log n\ 2^{2\log n})$
2019	Harvey y van der Hoeven	$\mathbf{O}(n \log n)$

Muchos lenguajes utilizan diferentes algoritmos según el tamaño de los números a multiplicar. El presentado por Schonhage Strassen es el de menor complejidad que llega a utilizarse. Los posteriores con mejores cotas logran ventaja para números extremadamente grandes. Tanto que en la práctica son más lentos que los anteriores.

En 1971 Strassen conjeturó que $\mathbf{O}(n \log n)$ es lo máximo que se puede lograr reducir la complejidad en la multiplicación. Tal vez tengamos la posibilidad de comprobar si esta resulta una afirmación errada o no en los próximos años.

9. Multiplicación de matrices

El paper de Karatsuba, sobre la multiplicación de enteros provocó una vuelta a aquellos problemas que se creían resueltos de forma inmejorable. La búsqueda de los algoritmos "fast" había comenzado. Uno de los problemas con lo que se trabajó fue la multiplicación

[16] "Integer multiplication in time O(n log n)", David Harvey; Joris van Der Hoeven. 2019. Annals of Mathematics, Princeton University, Department of Mathematics.
[17] "Fast multiplication of large numbers", Schönhage A.; Strassen V., 1971, Computing 7, 281–292.
[18] "Faster integer multiplication", Martin Fürer, 2007, STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing
[19] "Fast integer multiplication using modular arithmetic", Anindya De; Piyush P. Kurur; Chandan Saha; Ramprasad Saptharishi, 2008, STOC '08: Proceedings of the fortieth annual ACM symposium on Theory of computing
[20] "Fast integer multiplication using generalized Fermat primes", Svyatoslav Covanov, Emmanuel Thome, 2015

de matrices. Este proceso es de gran interés porque es utilizado en una gran variedad de problemas como parte del proceso de cálculo. La definición matemática para el calcular el resultado de la operación en matrices cuadradas de nxn es **Θ**(n^3). En ese sentido tal vez no era esperable encontrar algo superador. Sin embargo fue Volker Strassen en su publicación "Gaussian elimination is not optimal"[21] de 1969 quien probó que había lugar para mejorar.

Formalizando el problema, anunciaremos:

Multiplicación de dos matrices cuadradas de nxn
Dado A y B matrices de nxn. Deseamos calcular la matriz resultante de multiplicar A por B.

Recordemos el método de multiplicación tradicional o "naive". Partimos de dos matrices A y B ambas de nxn. Realizaremos AxB y llamaremos C a su resultado. Este será también una matriz cuadrada de nxn. Donde cada celda c de fila i y columna j se calculará como $C_{ij} = \sum_{k=1}^{n} A_{ik} x\, B_{kj}$. Con exactitud podemos ver que son n^3 multiplicaciones y n^2*(n-1) sumas. Lo que sumarizan $2n^3 - n^2$ operaciones elementales para obtener el resultado final.

Strassen propone resolver la multiplicación mediante división y conquista. El método funciona si "n" es potencia de 2. Aunque se puede adaptar para que funcione si no lo es. A continuación su idea será explicada paulatinamente.

Empezaremos con una propuesta que realmente no funciona, pero que sentará las bases posteriores. Cada matriz es dividida en 4 submatrices de n/2*n/2 celdas según muestra la siguiente figura

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11}*B_{11} + A_{12}*B_{21} & A_{11}*B_{12} + A_{12}*B_{22} \\ A_{21}*B_{11} + A_{22}*B_{21} & A_{21}*B_{12} + A_{22}*B_{22} \end{bmatrix}$$

El problema se transforma en resolver 8 multiplicaciones de matrices de n/2*n/2. Que podemos resolver recursivamente con este mismo esquema. Luego en la combinación

[21] Gaussian elimination is not optimal,Strassen V., 1969, Numer. Math. 13(4), 354–356.

realizar 4 sumas de matrices del mismo tamaño anterior. En el caso base tendremos un $\mathbf{O}(1)$ de sumas y multiplicaciones. Quedando una relación de recurrencia:

$T(n) = 8T(n/2) + \mathbf{O}(n^2)$
$T(1)=\mathbf{O}(1)$

Podemos usar el teorema maestro para resolverla. Obtenemos $\mathbf{O}(n^3)$ y solo complejizamos la operatoria para terminar con la misma cota de complejidad. Sin embargo, Strassen se dio cuenta que podría reescribir los cálculos y reducir en 1 la cantidad de subproblemas. Pasando de 8 a 7 y mejorando el resultado final.

Utiliza la misma división de las matrices. Realiza las siguientes 10 adiciones de matrices cuadradas de n/2

$$S_1 = B_{12} - B_{22} \qquad S_5 = A_{11} + A_{22} \qquad S_8 = B_{21} + B_{22}$$
$$S_2 = A_{11} + A_{12} \qquad S_6 = B_{11} + B_{22} \qquad S_9 = A_{11} - A_{21}$$
$$S_3 = A_{21} + A_{22} \qquad S_7 = A_{12} - A_{22} \qquad S_{10} = B_{11} + B_{12}$$
$$S_4 = B_{21} - B_{11}$$

Utilizaremos estos resultados para realizar 7 multiplicaciones recursivas de matrices cuadradas de n/2

$$P_1 = A_{11} * S_1 \qquad P_5 = S_5 * S_6$$
$$P_2 = S_2 * B_{22} \qquad P_6 = S_7 * S_8$$
$$P_3 = S_3 * B_{11} \qquad P_7 = S_9 * S_{10}$$
$$P_4 = A_{22} * S_4$$

Obtenidos los resultados de los subproblemas debemos realizar unas últimas 8 sumas de matrices cuadradas de n/2 para obtener las 4 submatrices del resultado

$$C_{11} = P_5 + P_4 - P_2 + P_6 \qquad C_{21} = P_3 + P_4$$
$$C_{12} = P_1 + P_2 \qquad C_{22} = P_5 + P_1 - P_3 - P_7$$

Tenemos finalmente 7 subproblemas de matrices cuadradas de n/2 y 18 adiciones $\mathbf{O}(n^2)$. Quedando una relación de recurrencia:

$T(n) = 7T(n/2) + \mathbf{O}(n^2)$

$T(1)=\mathbf{O}(1)$

Usando el teorema maestro obtendremos $\mathbf{O}(n^{\log 7})\approx\mathbf{O}(n^{2,8074})$. Strassen logró quebrar el exponente 3. A continuación el pseudocódigo resumido del algoritmo

Multiplicación de matrices rápida: Algoritmo de Strassen

```
Sea X y Y dos matrices cuadradas de nxn

Strassen(a,b):
        Si a y b con |n|=1
                Retornar a*b

        Calcular S1, S2, ..., S10
        P1 = Strassen(A11,S1)
        P2 = Strassen(S2,B22)
        P3 = Strassen(S3,B11)
        P4 = Strassen(A22,S4)
        P5 = Strassen(S5,S6)
        P6 = Strassen(S7,S8)
        P7 = Strassen(S9, S10)

        Calcular C1, C2, C3, C4
        Retornar C

Strassen(X,Y)
```

El algoritmo de Strassen se puede extender para matrices no cuadradas o que no son de tamaño potencia de 2. Hay diversas técnicas avanzadas para realizar este paso de formas eficientes. La forma más básica de todas es simplemente buscar la potencia de 2 inmediatamente superior al valor máximo entre las filas y las columnas. Luego agregar respectivamente filas y columnas con valor cero a la matriz y hacerla apta para el algoritmo. De esa forma se podrá realizar tantas subdivisiones hasta el caso base. Una implementación inteligente del algoritmo sin embargo evitará esto. Puede armar matrices de diferentes tamaños cuadradas o hacer partes de los cálculos con multiplicaciones naive.

Otra cuestión a tener en cuenta es que si se utiliza aritmética no entera, la existencia de errores de redondeo se van acumulando a medida que se propagan de subproblemas menores a mayores. Eso lo hace inviable ante cierto tipo de aplicaciones donde la exactitud en la solución es muy importante.

Una pregunta a responder es ¿cuándo conviene utilizar este algoritmo? La complejidad del algoritmo nos informa que a determinado valor "n" su cota asintótica es menor que la del algoritmo naive. Pero, cuál sería este valor? . Hay diferentes estudios teóricos y prácticos que trabajan sobre el tema.

Vamos a realizar un análisis teórico sobre el mismo. Cabe aclarar que este es solo a fines de estudio. No se tendrán en cuenta numerosas mejoras posibles al algoritmo y la posibilidad de paralelización del mismo que termina mejorando notablemente los tiempos reales de ejecución[22] [23] [24] y permitiendo que sea más veloz que el método naive para matrices notablemente más pequeñas. No aconsejamos tomar los valores de este análisis como exactos.

Consideremos que en un determinado subproblema nos llegan 2 matrices de tamaño n. Generamos otros 7 subproblemas que llamaremos recursivamente con matrices cuadradas de tamaño n/2. Además realizar 18 sumas de matrices de n/2 x n/2. En cada suma matricial realizamos n^2 sumas elementales. Por lo que tendremos en total $18n^2$ operaciones de suma. La relación de recurrencia la podemos expresar con más detalle (aunque aun siendo una aproximación) cómo:

$T(n) = 7T(n/2) + 18n^2$

[22] Strassen's Algorithm Reloaded, Huang, J.; Smith, T. M.; Henry, G. M.; Geijn; R. A. van de ., 2016, SC16: International Conference for High Performance Computing, Networking, Storage and Analysis.
[23] Exploiting parallelism in matrix-computation kernels for symmetric multiprocessor systems: Matrix-multiplication and matrix-addition algorithm optimizations by software pipelining and threads allocation, P. D'Alberto; M. Bodrato; A. Nicolau, Dic 2011, ACM Trans. Math. Softw. vol. 38, no. 1, pp. 2:1–2:30
[24] A framework for practical parallel fast matrix multiplication, A. R. Benson; G. Ballard, 2015, Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 42–53.

Por otro lado tenemos que definir el trabajo a realizar en el caso base, que tomará matrices de tamaño "n" y realizará una multiplicación matricial naive con las siguientes operaciones elementales (multiplicaciones y sumas)

$$T(n) = 2n^3 - n^2$$

Vemos que si esperamos al menor valor posible de la matriz esto se traduce solo en multiplicar dos números y por lo tanto t(n=1)=1. El caso base como **O**(1) nos permitió la aplicación del teorema maestro.

Consideramos matrices cuadradas potencia de dos, $n=2^k$. Podemos expresar el problema y sus siguientes términos anteriores como:

$$T(2^k) = 7T(2^k/2) + 18 * 2^{k^2} = 7T(2^{k-1}) + 18 * 2^{k^2}$$

$$T(2^{k-1}) = 7T(2^{k-2}) + 18 * 2^{(k-1)^2}$$

$$T(2^{k-2}) = 7T(2^{k-3}) + 18 * 2^{(k-2)^2}$$

y llegando al caso base luego de k llamadas recursivas. Si desenrollamos la recurrencia vemos que:

$$T(2^k) = 7[7\{7T(2^{k-3}) + 18 * 2^{(k-2)^2}\} + 18 * 2^{(k-1)^2}] + 18 * 2^{k^2}$$

Que podemos expresar, teniendo en cuenta todos los términos y el caso base (el término fuera de la sumatoria) como:

$$t(2^k) = 7^k + \sum_{i=0}^{k-1} 7^i * 18 * 2^{(k-i)^2} = 7^k + 18 * 2^{k^2} \sum_{i=0}^{k-1} \left(\frac{7}{4}\right)^i$$

Que es una serie geométrica que se puede resolver como:

$$t(2^k) = 7^k + 18 * 2^{k^2}\left(\frac{1-\left(\frac{7}{4}\right)^k}{1-\frac{7}{4}}\right) = 7^k + 18 * 2^{k^2} * \frac{4}{3} * \left(\left(\frac{7}{4}\right)^k - 1\right) = 7^k + 6 * 2^{2k+2} * \left(\left(\frac{7}{4}\right)^k - 1\right)$$

Continuamos trabajando en la expresión:

$$t(2^k) = 7^k + 6 * 2^{2k+2} * \left(\frac{7}{2^2}\right)^k - 6 * 2^{2k+2} = 25 * 7^k - 24 * 2^{k^2}$$

Y finalmente podemos volver a nuestra variable original puesto que k=$\log_2(n)$ y luego recordando $a^{log_2 n} = n^{log_2 a}$

$$t(n) = 25 * 7^{log_2 n} - 24 * 2^{(log_2 n)^2} = 25\, n^{log_2 7} - 24n^2$$

Llegamos a una expresión un poco más cercana a la real. Incluso comprobamos que el teorema maestro nos entregó la cota asintótica correcta. Si igualamos esta ecuación con la correspondiente a la determinada por la multiplicación naive obtenemos $25n^{log_2 7} - 24n^2 = 2n^3 - n^2$. Al resolver esta igualdad podemos encontrar el valor de la matriz a partir del cual es conveniente utilizar un método en lugar de otro. Corresponde a una matriz cuadrada con n=524288. Este valor se puede mejorar.

Observamos que la recursión la podemos detener antes de llegar a la matriz del menor tamaño posible. Analizando la ecuación $t(2^k) = \sum_{i=0}^{k} 7^i * 18 * 2^{(k-i)^2}$, vemos que en a un determinado valor de i - que representa el nivel de recursión - tendremos número de subproblemas dado por 7^i. Entonces tendremos un gran número de problemas de multiplicación de matrices de tamaño pequeño que son realizados de forma más ineficiente que al realizarlas en forma naive. Modificando el algoritmo podemos cambiar cuando ingresamos al caso base según el tamaño “r” de la matriz. Allí se realizará una multiplicación naive. Podemos estimar el impacto de modificar el valor de corte mediante un cálculo:

$$t(n,r) = (25\, n^{log_2 7} - 24n^2) - (25\, r^{log_2 7} - 24r^2) + (r^{log_2 7} * (2r^3 + r^2)) + \quad si\ n > r$$

$$2n^3 + n^2 \qquad si\ n \leq r$$

Esta función recibe dos parámetros el tamaño inicial de la matriz y el tamaño del subproblema a partir del cual resuelve mediante una multiplicación naive. Los primeros dos términos representan la cantidad de operaciones elementales que realiza la recursión

desde la matriz de tamaño n a r. El último término representa el trabajo de multiplicación naive realizado por todos los subproblemas de tamaño r.

Se puede estimar más adecuado el valor de r en 64. Con valores menores o mayores la cantidad de operaciones elementales aumenta. Para este valor utilizando matrices de tamaño 262144 o mayores es más adecuado Strassen que la multiplicación naive.

Antes de finalizar es importante comentar que existen algoritmos posteriores a Strassen que lograron disminuir aún más la complejidad en la resolución del problema. En la siguiente tabla un resumen de algunos de los más conocidos. La tercera columna contiene el exponente sobre el parámetro n en la complejidad temporal del algoritmo.

Año	Algoritmo	Exponente
1969	Strassen	2,8074
1978	Pan[25]	2,796
1979	Bini, Capovani, Romani[26]	2,780
1981	Schönhage	2,522
1981	Romani	2,517
1981	Coppersmith, Winograd	2,496
1986	Strassen	2,479
1990	Coppersmith, Winograd[27]	2,3755
2010	Stothers	2,3737
2013	Williams	2,3739
2014	Le Gall[28]	2,3728639

[25] Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations,V. Y. Pan, 1978, 19th Annual Symposium on Foundations of Computer Science (sfcs 1978), pp. 166-176
[26] O(n2.7799) Complexity for n*n Approximate Matrix Multiplication., Bini D.; Capovani M.; Romani F.; Lotti G., 1979, Inf. Process. Lett. 8 (5), pp. 234-235.
[27] Matrix Multiplication via Arithmetic Progressions, Coppersmith D.; Winograd, S., 1990, Journal of Symbolic Computation, 9, 251-280.
[28] Powers of Tensors and Fast Matrix Multiplication, François Le Gall, 2014, Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation (ISSAC 2014), pp. 296-303

2020	Alman, Williams[29]	2,3728596

En los últimos años los progresos obtenidos son cada vez menores. Además las constantes ocultas por la expresión asintótica son inmensas en todos los últimos algoritmos. Convirtiéndolos en algoritmos galácticos. Sin embargo, los efectos de la investigación teórica son muy importantes. Cada uno empujando los límites un poco más lejos.

[29] A Refined Laser Method and Faster Matrix Multiplication., Alman, Josh; Virginia Vassilevska Williams. 2021, SODA.