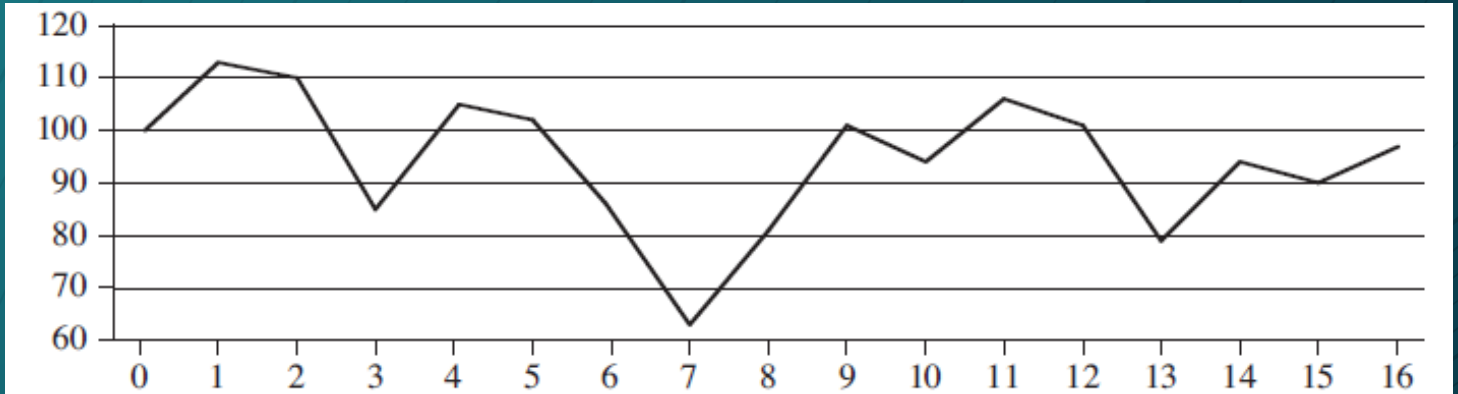
# El problema del máximo subarreglo

# El problema del máximo subarreglo

- Suponga que se le ha ofrecido la oportunidad de invertir en una corporación de químicos volátiles.
- Al igual que los productos químicos que produce la empresa, el precio de las acciones de la corporación es bastante volátil.
- Se le permite comprar una unidad de existencias solo una vez y luego venderlas en una fecha posterior, comprando y vendiendo después del cierre de operaciones del día.
- Para compensar esta restricción, se le permite conocer cuál será el precio de las acciones en el futuro.
- Tu objetivo es maximizar tu beneficio.

- En la Figura 1 se muestra el precio de la acción durante un período de 17 días.
- Tú puedes comprar las acciones en cualquier momento, comenzando después del día 0, cuando el precio es de $ 100 por acción.
- Por supuesto, querrá "comprar barato y vender caro": comprar al precio más bajo posible y luego vender al precio más alto posible, para maximizar su ganancia.
- Desafortunadamente, es posible que no pueda comprar al precio más bajo y luego vender al precio más alto dentro de un período determinado.

- En la siguiente Figura 1, el precio más bajo ocurre después del día 7, que ocurre después del precio más alto, después del día 1.




| Day | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Price | 100 | 113 | 110 | 85 | 105 | 102 | 86 | 63 | 81 | 101 | 94 | 106 | 101 | 79 | 94 | 90 | 97 |
| Change | | 13 | −3 | −25 | 20 | −3 | −16 | −23 | 18 | 20 | −7 | 12 | −5 | −22 | 15 | −4 | 7 |

**Figura 1.** Precio de las acciones después del cierre de negociación durante un período de 17 días. El eje horizontal indica el día y el eje vertical muestra el precio.
La fila inferior de la tabla muestra el cambio en el precio del día anterior.

- Podría pensar que siempre puede maximizar las ganancias comprando en el precio más bajo o vender al precio más alto.

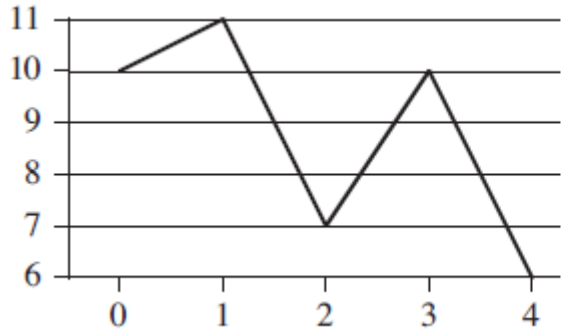
- Por ejemplo, en la Figura 1, deberíamos maximizar las ganancias comprando al precio más bajo, después del día 7.

- Si esta estrategia siempre funciona, entonces sería fácil determinar cómo maximizar las ganancias: encuentre el mayor y el menor precio, y luego trabaje a la izquierda desde el precio más alto hasta encontrar el precio previo más bajo, trabaje a la derecha, desde el precio más bajo para encontrar el precio más alto después, y tome el par con la mayor diferencia.

- La siguiente Figura 2 muestra un contraejemplo simple, que demuestra que el beneficio máximo a veces no se obtiene ni al comprar al precio más bajo ni vendiendo al precio más alto.

n = 4

f(n) = n(n-1) / 2
f(4) = 4(4-1) / 2
f(4) = 4(3) / 2
f(4) = 12 / 2
f(4) = 6




| Day | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Price | 10 | 11 | 7 | 10 | 6 |
| Change | | 1 | −4 | 3 | −4 |

(1,2)
(1,3)
(1,4)
(2,3)
(2,4)
(3,4)

Figura 2. Un ejemplo que muestra que el beneficio máximo no siempre comienza con el precio más bajo o termina con el precio más alto.
Nuevamente, el eje horizontal indica el día y el eje vertical muestra el precio.
Aquí, la ganancia máxima de $ 3 por acción se obtendría comprando después del día 2 y vendiendo después del día 3.
El precio de $ 7 después del día 2 no es el precio más bajo en general, y el precio de $10 después del día 3 no es el precio más alto en general.

# Una solución por fuerza bruta

- Podemos idear fácilmente una solución de fuerza bruta para este problema: simplemente intente todo los pares posibles de fechas de compra y venta en las que la fecha de compra precede a la fecha de venta.

$f(n) = (n(n-1)) / 2$
$f(n) = (n^2 - n) / 2$

- Un período de **n** días tiene $\binom{n}{2}$ pares de fechas.
- Desde $\binom{n}{2}$ es $\Theta(n^2)$ y lo mejor que podemos esperar es evaluar cada par de fechas en tiempo constante, este enfoque tomaría un tiempo $\Omega(n^2)$. ¿Podemos hacerlo mejor?

# Una transformación

- Para diseñar un algoritmo con un tiempo de ejecución $o(n^2)$, se verá la entrada de una manera ligeramente diferente.
- Queremos encontrar una secuencia de días durante la cual el cambio neto desde el primer día hasta el último es máximo.
- En lugar de mirar los precios diarios, consideremos el cambio diario en el precio, donde el cambio en el día i es la diferencia entre los precios después del día ***i - 1*** y después del día ***i***.
- En la Figura 1 se muestran estos cambios diarios en la fila inferior.

- Si tratamos esta fila como un arreglo A, mostrado en la Figura 3, ahora queremos encontrar el arreglo no vacío, contiguo de A cuyos valores tienen la mayor suma.
- Llamamos a este subarreglo contiguo el **subarreglo máximo**.
- Por ejemplo, en el arreglo de la Figura 3, el máximo subarreglo de A[1..16] es A[8..11], con la suma 43.
- Por lo tanto, querrá comprar las acciones justo antes del día 8 (es decir, después del día 7) y venderlas después del día 11, teniendo una ganancia de $ 43 por acción.

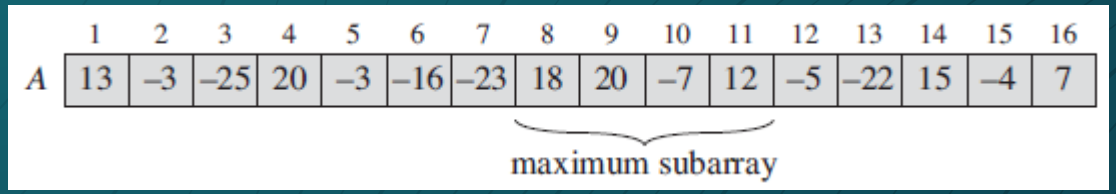



**Figura 3**. El cambio en los precios de las acciones como un problema de subarreglo máximo. Aquí, el subarreglo A[8..11], con suma 43, tiene la mayor suma de cualquier subarreglo contiguo del arreglo A.

- Así que busquemos una solución más eficiente al problema de subarreglos máximos.
- Al hacerlo, normalmente hablaremos de "un" subarreglo máximo en lugar de "el" subarreglo máximo, ya que podría haber más de un subarreglo que alcance la suma máxima.
- El problema del subarreglo máximo es interesante solo cuando el arreglo contiene algunos números negativos.
- Si todas las entradas de la matriz no fueran negativas, entonces el problema del subarreglo máximo no presentaría ningún desafío, ya que el arreglo completo daría la mayor suma.

# Una solución usando el paradigma “Divide y vencerás”

# Una solución usando “Divide y vencerás”

- Supongamos que queremos encontrar un subarreglo máximo
- del subarreglo *A[low .. high]*.

- Divide y vencerás sugiere que se divida el subarreglo en dos subarreglos del mismo tamaño como sea posible.

- Es decir, encontramos el punto medio, digamos *mid*, del subarreglo, y considere los subarreglos:
  - *A[low .. mid]*
  - *A[mid+1 .. high]*

- Como muestra la Figura 4(a), cualquier subarreglo contiguo ***A[i..j]*** de ***A[low..high]*** debe estar exactamente en uno de los siguientes lugares:
  - completamente en el subarreglo ***A[low..mid]***, de modo que:
    $$low \le i \le j \le mid$$
  - completamente en el subarreglo ***A[mid+1..high]***, de modo que:
    $$mid < i \le j \le high$$
  - cruzando el punto medio, de modo que: $low \le i \le mid < j \le high$

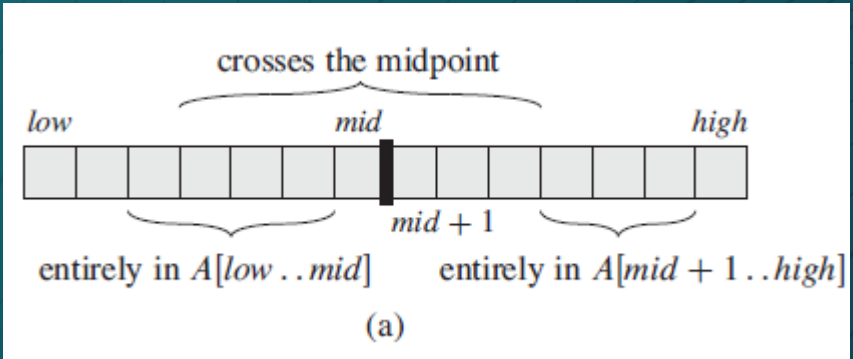



**Figura 4(a)**. Posibles ubicaciones de subarreglos de ***A[low..high].*** Completamente en ***A[low..mid]***, completamente en ***A[mid+1..high]***, o cruzando el punto medio ***mid***.

- Por lo tanto, un subarreglo máximo de $A[low..high]$ debe estar exactamente en uno de estos lugares.

- De hecho, un subarreglo máximo de $A[low..high]$ debe tener la mayor suma sobre todos los subarreglos completamente en $A[low..mid]$, completamente en $A[mid+1..high]$, o cruzando el punto medio.

- Podemos encontrar subarreglos máximos de $A[low..mid]$ y $A[mid+1..high]$ de forma recursiva, porque estos dos subproblemas son instancias más pequeñas del problema de encontrar un subarreglo máximo.

- Por lo tanto, todo lo que queda por hacer es encontrar un subarreglo máximo que cruce el punto medio y tomar un subarreglo con la mayor suma de los tres.

- Podemos encontrar fácilmente un subarreglo máximo que cruce el punto medio en tiempo lineal en el tamaño del subarreglo *A[low..high]*.

- Este problema no es una instancia menor de nuestro problema original, porque tiene la restricción adicional de que el subarreglo elegido debe cruzar el punto medio.

- Como muestra la Figura 4(b), cualquier cruce de subarreglos en el punto medio está formado por dos subarreglos ***A[i .. mid] y A[mid+1 .. j]***, donde: $low \le i \le mid$

  $mid < j \le high$

- Por lo tanto, solo necesitamos encontrar el máximo subarreglo de la forma ***A[i .. mid] y A[mid+1 .. j]*** y luego combinarlos.

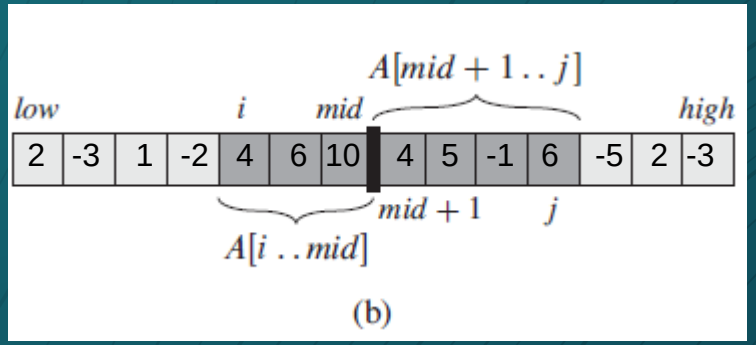



**Figura 4(b)**. Cualquier subarreglo de ***A[low..high]*** el punto medio comprende dos subarreglos ***A[i..mid]*** y ***A[mid+1..j]***, donde ***low ≤ i ≤ mid*** y ***mid < j ≤ high***.

El procedimiento FIND-MAX-CROSSING-SUBARRAY toma como entrada el arreglo A y los índices ***low***, ***mid*** y ***high***, y devuelve una tupla que contiene los índices que delimitan un subarreglo máximo que cruza el punto medio, junto con la suma de los valores en un subarreglo máximo.

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       sum = sum + A[i]
 5       if sum > left-sum
 6            left-sum = sum
 7            max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

- Las líneas 1-7 encuentran un subarreglo máximo de la mitad izquierda, *A[low..mid]*.

- Dado que este subarreglo debe contener *A[mid]*, el bucle for de las líneas 3 a 7 inician en el índice **i** en ***mid*** y van hacia ***low***, de modo que cada subarreglo se considera que es de la forma *A[i..mid]*.

- Las líneas 1 a 2 inicializan las variables **left-sum**, que contiene la mayor suma encontrada hasta ahora, y **sum** que contiene la suma de las entradas en *A[i..mid]*.

- Siempre que encontremos, en la línea 5, un subarreglo *A[i..mid]* con una suma de valores mayores que la **left-sum**, actualizamos **left-sum** a la suma de este subarreglo en la línea 6, y en la línea 7 actualizamos la variable **max-left** para registrar este índice **i**.

- Las líneas 8-14 funcionan análogamente para la mitad derecha, *A[mid+1..high]*.

- Aquí, el bucle for de las líneas 10-14 inicia el índice ***j*** en ***mid+1*** e itera hasta ***high***, de modo que cada subarreglo que considera tiene la forma *A[mid+1..j]*.

- Finalmente, la línea 15 devuelve los índices ***max-left*** y ***max-right*** que delimitan un subarreglo máximo que cruza el punto medio, junto con la suma ***left-sum + right-sum*** de los valores en el subarreglo:

*A[max-left .. max-right]*

- Si el subarreglo *A[low..high]* contiene **n** entradas

$$n = high - low + 1$$

- afirmamos que la llamada

FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)

- toma un tiempo **Θ(n)**.
- Dado que cada iteración de cada uno de los dos bucles for toma un tiempo **Θ(1)**, solo necesitamos contar cuántas iteraciones hay en total.
- El bucle for de las líneas 3-7 hace ***mid – low + 1*** iteraciones, y el bucle for de las líneas 10-14 hace **high – mid** iteraciones, por lo que el número total de iteraciones es:

$$\begin{aligned}(mid - low + 1) + (high - mid) &= high - low + 1 \\ &= n\ .\end{aligned}$$

- Con este procedimiento FIND-MAX-CROSSING-SUBARRAY de tiempo lineal, se puede escribir pseudocódigo para un algoritmo de "Divide y vencerás" para resolver el problema del subarreglo máximo:

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2       return (low, high, A[low])          // base case: only one element
 3  else mid = ⌊(low + high)/2⌋
 4       (left-low, left-high, left-sum) =
              FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5       (right-low, right-high, right-sum) =
              FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 6       (cross-low, cross-high, cross-sum) =
              FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7       if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8            return (left-low, left-high, left-sum)
 9       elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10            return (right-low, right-high, right-sum)
11       else return (cross-low, cross-high, cross-sum)
```

- La llamada inicial FIND-MAXIMUM-SUBARRAY(A, 1, A.length) encontrará un máximo subarreglo de **A[1..n]**.

- Similar a FIND-MAX-CROSSING-SUBARRAY, el procedimiento recursivo FIND-MAXIMUM-SUBARRAY devuelve una tupla que contiene los índices que delimitan un subarreglo máximo, junto con la suma de los valores en un subarreglo máximo.

- La línea 1 prueba el caso base, donde el subarreglo tiene solo un elemento.

- Un subarreglo con un solo elemento tiene solo un subarreglo, él mismo, por lo que la línea 2 devuelve una tupla con los índices inicial y final de un solo elemento, junto con su valor.

- Las líneas 3-11 manejan el caso recursivo.
- La línea 3 hace la parte de la división, calculando el índice ***mid*** del punto medio.
- Vamos a referirnos al subarreglo ***A[low..mid]*** como el subarreglo **izquierdo** y ***A[mid+1..high]*** como el subarreglo **derecho**.
- Porque sabemos que el subarreglo ***A[low..high]*** contiene al menos dos elementos, cada uno de los subarreglos izquierdo y derecho deben tener al menos un elemento.
- Las líneas 4 y 5 conquistan de forma recursiva encontrar los subarreglos máximos dentro de los subarreglos izquierdo y derecho, respectivamente.

- Las líneas 6 a 11 forman la parte combinada.
- La línea 6 encuentra un subarreglo máximo que cruza el punto medio. (Recuerde que debido a que la línea 6 resuelve un subproblema que no es una instancia del problema original, consideramos que está en la parte combinada).
- La línea 7 comprueba si el subarreglo izquierdo contiene un subarreglo con la suma máxima, y la línea 8 devuelve ese subarreglo máximo.
- De lo contrario, la línea 9 comprueba si el subarreglo derecho contiene un subarreglo con la suma máxima, y la línea 10 devuelve ese máximo subarreglo.
- Si ni los subarreglos izquierdo ni derecho contienen un subarreglo que logre la suma máxima, entonces un subarreglo máximo debe cruzar el punto medio, y la línea 11 lo devuelve.

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2       return (low, high, A[low])          // base case: only one element
 3  else mid = ⌊(low + high)/2⌋
 4       (left-low, left-high, left-sum) =
              FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5       (right-low, right-high, right-sum) =
              FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 6       (cross-low, cross-high, cross-sum) =
              FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7       if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8            return (left-low, left-high, left-sum)
 9       elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10            return (right-low, right-high, right-sum)
11       else return (cross-low, cross-high, cross-sum)
```

# Analizando el algoritmo “Divide y vencerás”

## El problema del máximo subarreglo

# Analizando el algoritmo “Divide y vencerás”

- Se configura una recurrencia que describe el tiempo de ejecución del procedimiento recursivo FIND-MAXIMUM-SUBARRAY.
- Se hace la suposición de que el tamaño original del problema es una potencia de 2, por lo que todos los tamaños de subproblemas son números enteros.
- Denotamos por **T(n)** el tiempo de ejecución de FIND-MAXIMUM-SUBARRAY en un subarreglo de **n** elementos.
- La línea 1 lleva un tiempo constante.
- El caso base, cuando n = 1, es fácil.
- La línea 2 toma un tiempo constante, por lo que: $T(1) = \Theta(1)$

Ecuación 1

- El caso recursivo ocurre cuando n > 1.
- Las líneas 1 y 3 toman un tiempo constante.
- Cada uno de los subproblemas resueltos en las líneas 4 y 5 está en un subarreglo de **n/2** elementos (la suposición de que el tamaño del problema original es una potencia de 2 asegura que **n/2** es un entero), por lo que se gasta un tiempo de **T(n/2)** resolviendo cada uno de ellos.
- Porque tenemos que resolver dos subproblemas, para el subarreglo izquierdo y para el subarreglo derecho, la contribución al tiempo de ejecución de las líneas 4 y 5 es de **2T(n/2)**.

- Como ya hemos visto, la llamada a FIND-MAX-CROSSING-SUBARRAY en la línea 6 toma un tiempo **Θ(n)**.
- Las líneas 7–11 toman un tiempo **Θ(1)**.
- Para el caso recursivo, por lo tanto, se tiene:

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &= 2T(n/2) + \Theta(n) \ . \end{aligned}$$

Ecuación 2

- La combinación de las ecuaciones (1) y (2) nos da una recurrencia para el tiempo de ejecución de FIND-MAXIMUM-SUBARRAY de:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Ecuación 3

- Esta recurrencia es la misma que la recurrencia para la ordenación por mezcla (Merge Sort).
- Por el método maestro esta recurrencia tiene la solución:

$$\mathbf{T(n) = \Theta(n \lg n)}.$$

- Por lo tanto, vemos que el método divide y vencerás produce un algoritmo que es asintóticamente más rápido que el método de fuerza bruta.
- Con el ordenamiento por mezcla y ahora el problema de subarreglo máximo, comenzamos a tener una idea de cuán poderoso es el método de divide y vencerás.
- A veces producirá el algoritmo asintóticamente más rápido para un problema, y otras veces se puede hacer incluso mejor.