Divide and Conquer

Divide y vencerás

- Enfoque de resolución de problemas basado en la recursividad.
- En cada estado de la recursión realiza 3 pasos: dividir, vencer y combinar.
- **Dividir (*divide*)** el problema en una cantidad determinada de subproblemas, que son instancias más pequeñas del mismo problema.
- **Vencer (*conquer*)** los subproblemas resolviéndolos de forma recursiva. Si el tamaño del subproblema es lo suficientemente pequeño, basta resolverlo de forma directa.
- **Combinar (*combine*)** la soluciones de los subproblemas para obtener la solución del problema original.

Dividir

- En la **mayoría** de casos basta dividir un problema en **2 subproblemas** de igual tamaño (o tamaños muy similares).

- En casos más raros puede ser conveniente dividir el problema en más de 2 subproblemas o en tamaños desiguales.

Pseudocódigo

```
Answer solve(int n) {
    // If the problem is small enough, solve it directly
    if (n <= SMALL) {
        return solveDirectly(n);
    }

    // Divide
    int leftSize = n / 2;
    int rightSize = n - leftSize;

    // Conquer
    Answer leftAnswer = solve(leftSize);
    Answer rightAnswer = solve(rightSize);

    // Combine
    return combine(leftAnswer, rightAnswer);
}
```

Time Complexity

- Sea $T(n)$ el número de operaciones que usa la función $solve$. Sea $f(n)$ el número de operaciones que usan las funciones $combine$ y $divide$. Como $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$, entonces

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + f(n).$$

- La complejidad exacta dependerá de $f(n)$. Existen algunos métodos clásicos para poder hallar la complejidad. Los veremos con más detalle más adelante con algunos ejemplos, pero una explicación rápida de estos métodos es la siguiente:

- **Recursión-tree method:** Utilizamos un árbol para representar gráficamente los niveles de recursión. En cada costo, colocamos el costo de $f(n)$. El resultado $T(n)$ es la suma de los valores en cada nodo.
- **Substitution method:** Estimamos una cota para $T(n)$ y la demostramos por inducción.

Ejemplos

Exponenciación

- **Problema: Calcular a^n con $a \in \mathbb{R}, n \in \mathbb{Z}^+ \cup \{0\}$.**

Sabemos que podemos usar fuerzabruta y resolverlo en $O(n)$. También podríamos resolverlo recursivamente con divide and conquer usando la siguiente recurrencia:

$$a^n = \begin{cases} 1, & n = 0 \\ a, & n = 1 \\ a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}, & n > 1 \end{cases}$$

```
Long power(Long a, Long n) {
    if (n == 0) return 1;
    if (n == 1) return a;
    return power(a, n / 2) * power(a, n - n / 2);
}
```

Exponenciación – Time Complexity

Notemos que nuestra lógica para hacer el *combine* es simplemente una multiplicación de 2 enteros, lo cual es $O(1)$. Entonces nuestro $f(n) = O(1)$, es decir $f(n) \leq c$ (donde c es una constante).

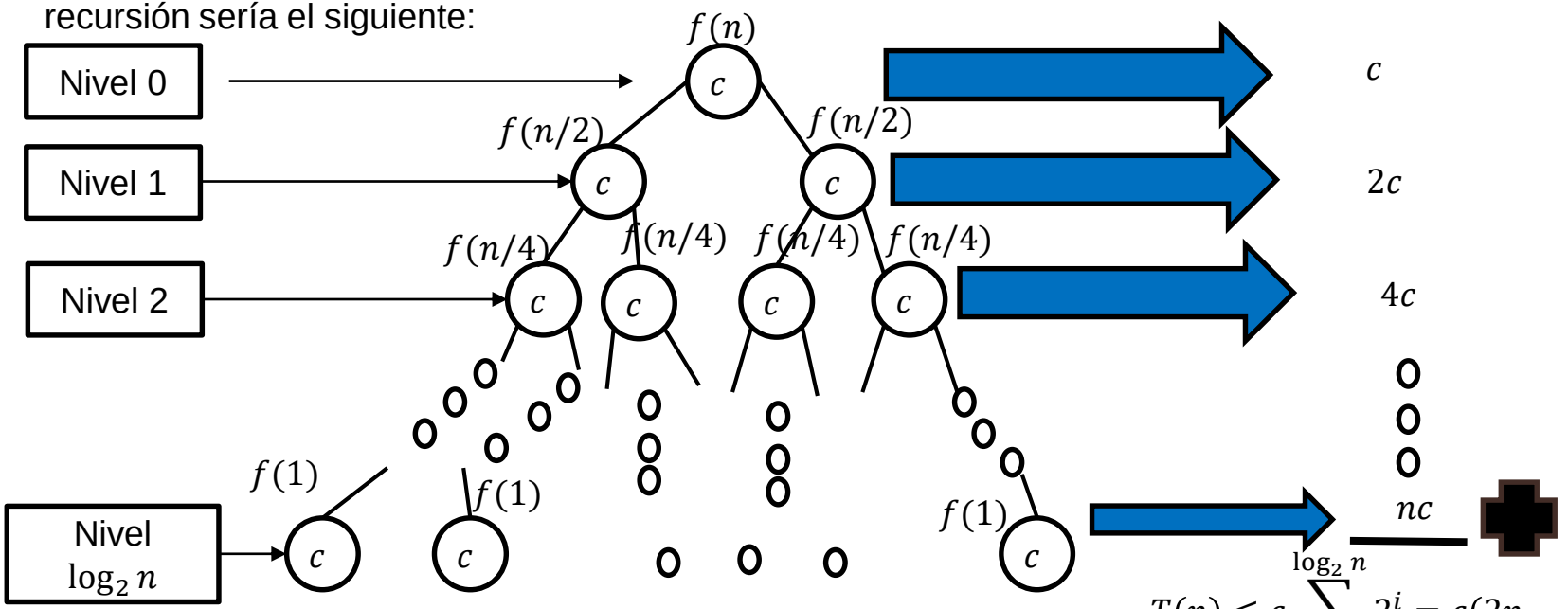
Por lo tanto, tenemos que $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c$.

Con este ejemplo explicaremos 2 métodos clásicos utilizados para estimar complejidad en *divide and conquer*:

- **Recursión-tree method**
- **Substitution method**

Exponenciación – Recursion Tree method

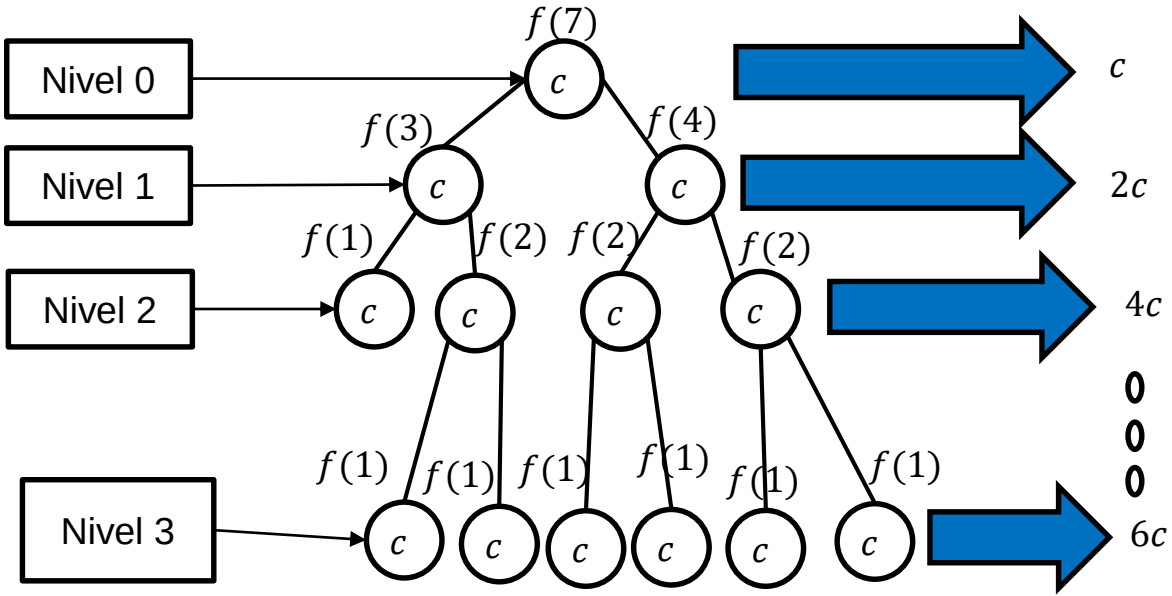
Para simplificar el análisis, supongamos que n es una potencia de 2. De tal forma que recursivamente siempre podemos dividir a la mitad este número y tenemos que $T(n) \leq 2T(n/2) + c$. Entonces el árbol de recursión sería el siguiente:



$$T(n) \leq c \sum_{i=0}^{\log_2 n} 2^i = c(2n-1) = O(n)$$

Exponenciación – Recursion Tree method

¿Cómo haríamos en los casos que no son potencias de 2? En realidad no es tan difícil. Veámoslo con un ejemplo particular para $n = 7$



En general, podemos deducir que

- El número de niveles es $\lceil \log_2 n \rceil + 1$
- Ya no se cumple que el la suma sea siempre igual a $2^i \times c$. Sin embargo, podemos decir que siempre es $\leq 2^i \times c$, debido a que el ultimo nivel puede tener menos de 2^i $nodos$

Con estas dos condiciones, podemos llegar a la misma complejdiad $O(n)$

Exponenciación – Substitution method

Para este método necesitamos primero “adivinar” una posible cota y luego demostrarla. Esta estimación inicial puede venir, por ejemplo, de un análisis rápido del recursion tree method.

Entonces, si estimemos una cota $T(n) \leq dn - c$, para cierto $d > 0$.
Por inducción,

- **Caso base:** $T(1) = c \leq d - c$ para todo $d \geq 2c$
- **Hipótesis inductiva:** supongamos que se cumple que $T(m) \leq dm - c$ para todo $m < n$.

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c \leq d \left\lfloor \frac{n}{2} \right\rfloor - c + d \left\lceil \frac{n}{2} \right\rceil - c + c = dn - c$$
$$\Longrightarrow T(n) \leq dn - c$$

Esto demuestra que $T(n) = O(n)$■

La desventaja de este método es que tienes que hacer una primera estimación. Es por eso que normalmente se prefiere utilizar el método de *recursión tree*.

Exponenciación Rápida

- **Problema: Calcular a^n con $a \in \mathbb{R}, n \in \mathbb{Z}^+ \cup \{0\}$.**

El método anterior nos dio la misma complejidad que una fuerzabruta, por lo que no tuvimos ninguna ganancia. Sin embargo, podemos hacer una optimización extra con la siguiente recurrencia:

$$a^n = \begin{cases} 1, & n = 0 \\ a^{n/2} \times a^{n/2}, & n\ par \\ a^{\lfloor n/2 \rfloor} \times a^{\lfloor n/2 \rfloor} \times a, & n\ impar \end{cases}$$

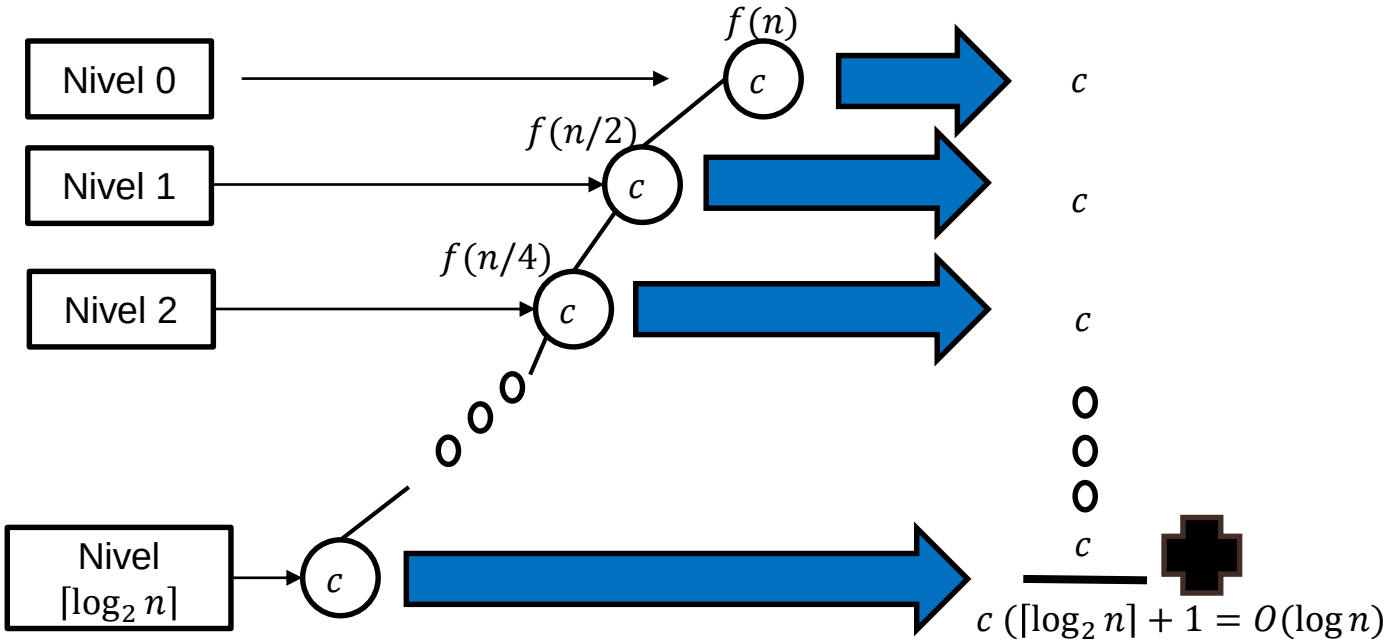
Nota: Si n es impar => $\lfloor \frac{n}{2} \rfloor = (\frac{n-1}{2})$

Podemos hacer una optimización adicional. Considerando que solo debemos calcular $a^{\lfloor n/2 \rfloor}$ una sola vez y luego podemos reusarlo.

```
Long fastPow(Long a, Long n) {
    if (n == 0) return 1;
    Long partial = fastPow(a, n / 2);
    Long ans = partial * partial;
    if (n % 2 == 1) ans *= a;
    return ans;
}
```

Exponenciación Rápida - Complejidad

Con esta optimización, la complejidad cambió a $T(n) = T(\lfloor n/2 \rfloor) + O(1)$.



Exponenciación Rápida Modular

- Como las potencias rápidamente sobrepasarán los 64 bits, causando overflow, normalmente en los concursos viene el siguiente **problema: Calcular $a^n \ mod \ c$ con $a, c \in \mathbb{Z}^+, n \in \mathbb{Z}^+ \cup \{0\}, c$.**

```
Long fastPow(Long a, Long n, Long c) { // O(log n)
    if (n == 0) return 1;
    Long partial = fastPow(a, n / 2, c);
    Long ans = (partial * partial) % c;
    if (n % 2 == 1) ans = (ans * a) % c;
    return ans;
}
```

Exponenciación Rápida - Generalizaciones

Podemos utilizar este mismo truco con cualquier operador que se aplique n veces y sea *asociativo*. Por ejemplo los operadores suma o xor son operadores asociativos (solo que el xor no es interesante para este caso).

- **Problema: Calcular $a \times n$ con $a, n \in \mathbb{Z}^+ \cup \{0\}$.**

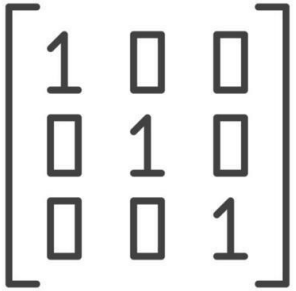
$$a \times n = \begin{cases} 0, & n = 0 \\ \left(a \times \frac{n}{2}\right) + \left(a \times \frac{n}{2}\right), & n\ par \\ \left(a \times \frac{n}{2}\right) + \left(a \times \frac{n}{2}\right) + a, & n\ impar \end{cases}$$

Podemos aplicar Divide and Conquer para hallar $(a \times n)$ en $O(\log n)$. Esto puede ser útil si estamos interesados en hallar $(a \times b)\ mod\ c$ pero los números a, b son muy cercanos al límite de 64 bits, por lo que la operación directa daría overflow.

Exponenciación de Matrices

La multiplicación de matrices también es asociativa, así que podemos hacer exponenciación de matrices usando el mismo método. Sin embargo la complejidad cambia debido a que, a diferencia de la multiplicación de dos números, que tiene complejidad $O(1)$; la multiplicación de 2 matrices cuadradas de dimensión n tiene complejidad $O(n^3)$. Por lo que si queremos elevar una matriz $n \times n$ a la potencia b , la complejidad sería

$$\boldsymbol{O(n^3 \log b)}$$



Merge sort

Problema: Ordenar un arreglo de n elementos

Algoritmo:
- ❑ **Divide:** Dividir el arreglo en dos mitades.
- ❑ **Conquer:** Ordena ambas mitades recursivamente. (Caso base cuando $n \leq 1$)
- ❑ **Combine:** Junta ambas mitades (ordenadas) de manera que el arreglo final se mantenga ordenado.

El paso de **dividir** es bastante simple y se puede hacer en $O(n)$.

El paso de **combine** (también llamado **merge**) es el más complicado. Se trata de combinar 2 arreglos A, B previamente ordenados, de manera que el resultado C siga estando ordenado.

Merge sort

Subproblema: Dados dos arreglos A, B ordenados, combinar sus elementos en un arreglo C también ordenado.

Podemos hacerlo con una técnica llamada *two pointers.* Con la cual vamos a tener dos “punteros” (posiciones) apuntando a los elementos más pequeños de ambas listas. En cada iteración, vamos añadiendo el mínimo de estos dos elementos a C y avancemos ese puntero en la lista, hasta que hayamos recorrido todos los elementos. Ejemplo:
Si tenemos $A = [1,5,9,15], B = [2,5,20]$. Esto haría el algoritmo:
$A = [\mathbf{\sout{1}}, 5,9,15], B = [2,5,20], C = [1]$
$A = [\sout{1},5,9,15], B = [\mathbf{\sout{2}}, 5,20], C = [1,2]$
$A = [\sout{1}, \mathbf{\sout{5}}, 9,15], B = [\sout{2},5,20], C = [1,2,5]$
$A = [\sout{1},\sout{5},9,15], B = [\sout{2}, \mathbf{\sout{5}}, 20], C = [1,2,5,5]$
$A = [\sout{1},\sout{5}, \mathbf{\sout{9}}, 15], B = [\sout{2},\sout{5},20], C = [1,2,5,5,9]$
$A = [\sout{1},\sout{5},\sout{9}, \mathbf{\sout{15}}], B = [\sout{2},\sout{5},20], C = [1,2,5,5,9,15]$
$A = [\sout{1},\sout{5},\sout{9},\sout{15}], B = [\sout{2},\sout{5}, \mathbf{\sout{20}}], C = [1,2,5,5,9,15,20]$

Es fácil notar que la complejidad es $O(|A| + |B|)$

Merge sort

- **Combine:** Junta ambas mitades (ordenadas) de manera que el arreglo final se mantenga ordenado.

```cpp
vector<int> merge(vector<int> &L, vector<int> &R) { // O(L + R)
    vector<int> ans;
    int indL = 0;
    int indR = 0;
    int n = L.size() + R.size();
    for (int i = 0; i < n; i++) {
        if (indL == L.size()) {
            // we already finished with L
            ans.push_back(R[indR]);
            indR++;
        } else if (indR == R.size()) {
            // we already finished with R
            ans.push_back(L[indL]);
            indL++;
        } else {
            if (L[indL] <= R[indR]) {
                ans.push_back(L[indL]);
                indL++;
            } else {
                ans.push_back(R[indR]);
                indR++;
            }
        }
    }
    return ans;
}
```

Merge sort

Ahora ya podemos completar el Divide & Conquer.

```
void sort(vector<int> &v) {
    int n = v.size();
    if (n <= 1) return; // base case, already sorted

    // Divide
    vector<int> L, R;
    for (int i = 0; i < n; i++) {
        if (i < n / 2) L.push_back(v[i]);
        else R.push_back(v[i]);
    }

    // Conquer
    sort(L);
    sort(R);

    // Combine
    v = merge(L, R); // O(n)
}
```

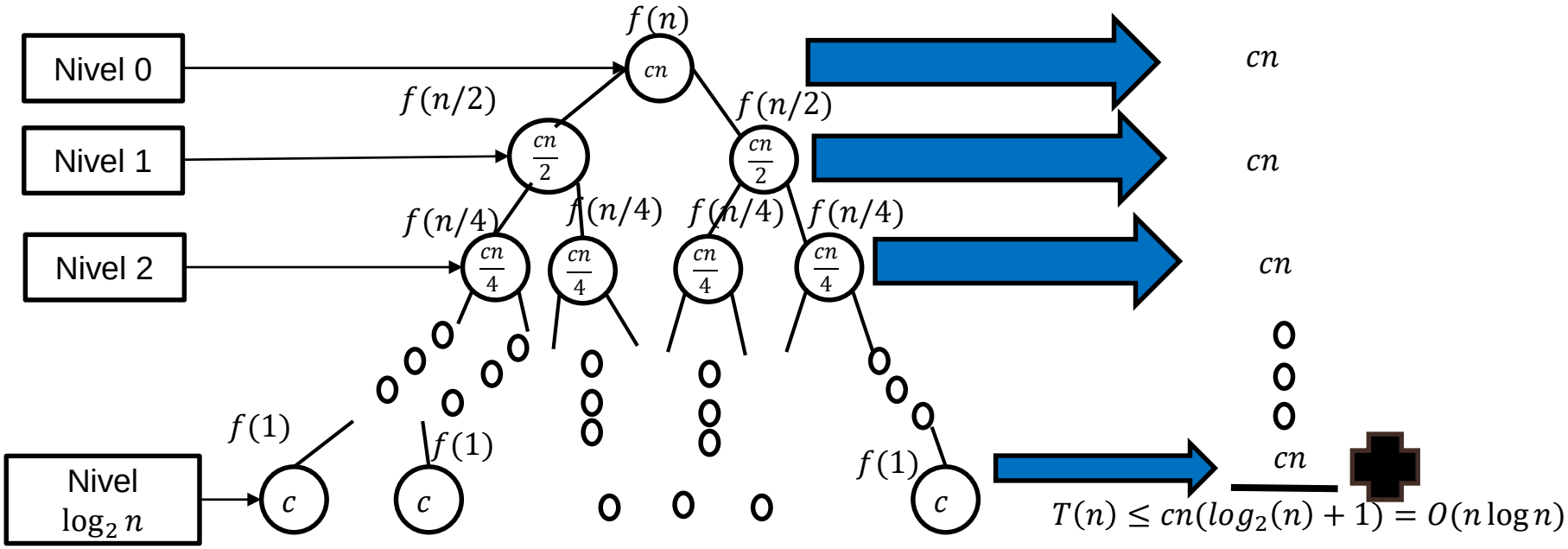
Merge sort

Para visualizarlo mejor, esto pasaría si tuviéramos el arreglo $[6, 5, 3, 1, 8, 7, 2, 4]$

Fuente: Wikipedia

Merge Sort – Time Complexity

Podemos ver que $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$. Es decir $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$.
Construyamos el árbol de recursión asumiendo potencias de 2.



Merge Sort – Time Complexity

Si n no es potencia de 2, habrán $\lceil \log_2 n \rceil + 1$ niveles y, en cada nivel, la suma de tiempos será menor o igual que cn por lo que llegaremos a la misma cota $O(n \log n)$.

En C++ ya existe una función sort que funciona en la misma complejidad. Sin embargo, el análisis de complejidad utilizado para merge sort es muy útil para problemas que tienen la misma estructura.

Maximum subarray sum

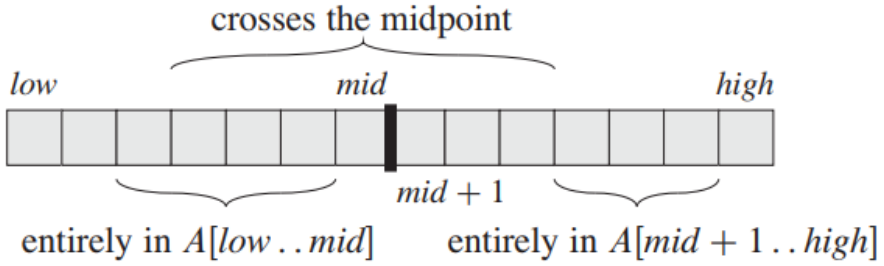
Problema: Dado un arreglo A de n elementos, encontrar la máxima suma posible de elementos consecutivos (subarray). El arreglo vacío no está permitido como posible solución.

Solución simple: ¡Utilizar fuerzabruta! en $O(n^2)$ podemos probar todos los subarreglos.
Solución Divide & Conquer:

- **Divide:** Dividir el arreglo en dos mitades.
- **Conquer:** Retornamos la máxima suma consecutiva recursivamente. (Caso base cuando $n = 1$ trivial)
- **Combine:** Para combinar la respuesta debemos identificar que hay 3 casos, la respuesta puede ubicarse:

1. Completamente a la izquierda
2. Completamente a la derecha
3. Cruzando el punto medio

La respuesta sería el máximo de los 3 casos. El 3er caso es el más complicado.



Maximum subarray sum

Subproblema (maximum crossing subarray): Dado dos arreglos A, B, encontrar la máxima suma que se obtendría de juntar un arreglo sufijo de A con un arreglo prefijo de B.

La solución es sencilla. Simplemente debemos encontrar la máxima suma sufijo de A en $O(n)$ y la máxima suma sufijo de B en $O(n)$ y luego dar como resultado la suma de ambos.

Maximum subarray sum

Optimización del Divide: En el caso del merge sort, a la hora de hacer la operación de dividir el arreglo, tuvimos que hacer un bucle en $O(n)$ para construir estos arreglos. Para este problema en particular, realmente no necesitamos hacer eso. Podemos mantener el rango $[l, r]$ de las posiciones del arreglo con las que estamos trabajando.

Aunque esto no reducirá la complejidad, mejorará la “constante escondida”.

Maximum Subarray Sum

```
const int MX = 1e5;
Long A[MX];

Long maxCrossingSubarraySum(int l, int mid, int r) {
    // Two arrays A[l...mid] and A[mid + 1...r]
    Long maxSuffix = A[mid];
    Long acum = 0;
    for (int i = mid; i >= l; i--) {
        acum += A[i];
        maxSuffix = max(maxSuffix, acum);
    }
    Long maxPrefix = A[mid + 1];
    acum = 0;
    for (int i = mid + 1; i <= r; i++) {
        acum += A[i];
        maxPrefix = max(maxPrefix, acum);
    }
    return maxPrefix + maxSuffix;
}
```

```
Long maxSubarraySum(int l, int r) {
    if (l == r) return A[l];
    int mid = (l + r) / 2;
    Long leftSum = maxSubarraySum(l, mid);
    Long rightSum = maxSubarraySum(mid + 1, r);
    Long cross = maxCrossingSubarraySum(l, mid, r);
    return max({leftSum, rightSum, cross});
}
```

Maximum subarray sum – Time Complexity

Como la función del **combine** se hace en $O(n)$ tenemos $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$.

Esto es exactamente igual que en merge sort, por lo que el resultado es $O(n \log n)$.

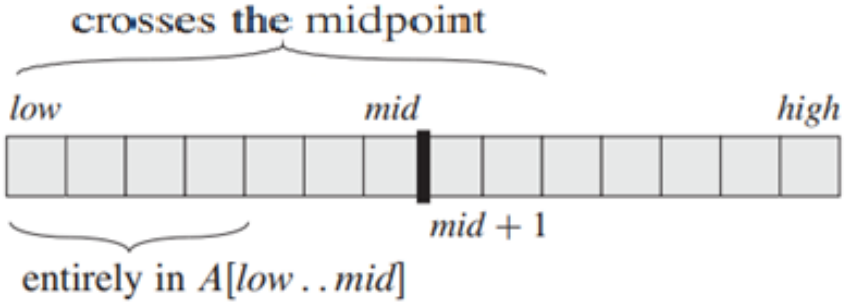
Pero… ¿y si pudiésemos hacer el **combine** en $O(1)$? Obtendríamos una complejidad similar a la que analizamos en la exponenciación de enteros dando un total de $O(n)$.

En el **combine** estamos hallando la suma prefijo máxima y la suma sufijo máxima. ¿Qué pasaría si estos valores lo calculásemos en el **conquer** (recursión)? Analicémoslo de manera separada.

Máxima suma prefijo

Key observation: Si queremos calcular recursivamente la respuesta respecto a las dos mitades, podemos notar que la respuesta solo puede ser de dos tipos:

- El máximo prefijo está solo en la parte izquierda
- El máximo prefijo está cruzando el punto medio

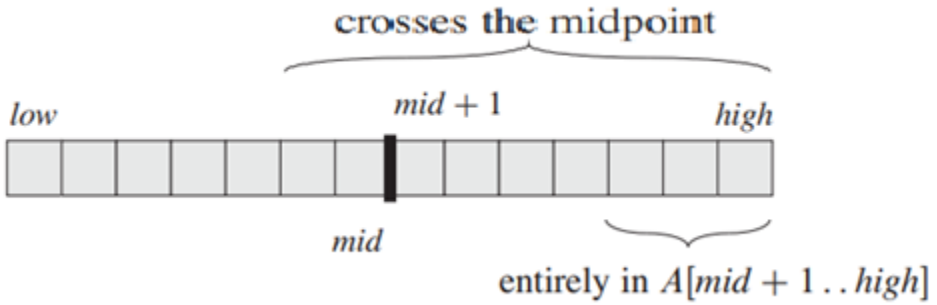


$$maxPrefix(l,r) = \max\left(maxPrefix(l,mid), sum(l,mid) + maxPrefix(mid+1\ ,r)\right)\ \blacksquare$$

Máxima suma sufijo

Aquí también hay dos casos

- El máximo sufijo está solo en la parte derecha
- El máximo sufijo está cruzando el punto medio



$$maxSuffix(l,r) = \max\big(maxSuffix(mid+1,r), maxSuffix(l\,,mid) + sum(mid+1,r)\big)\ \blacksquare$$

Maximum subarray sum - Optimization

Volviendo a nuestro problema, tendríamos la siguiente solución optimizada.

- **Divide:** Dividir el arreglo en dos mitades.
- **Conquer:** Recursivamente debemos retornar 4 valores $maxSubarraySum, maxPrefixSum, maxSuffixSum, sum$**.** En el caso base de $n = 1$, las 4 variables son iguales a el único elemento.
- **Combine:** Actualizamos las 4 variables con las fórmulas obtenidas.

Complejidad total: $O(n)$

Referencias

- [1] CLRS. Introduction to Algorithms. Chapter 4: Divide-and-Conquer
- [2] Jeff Erickson. Algorithms. Chapter 1 – Recursion. https://jeffe.cs.illinois.edu/teaching/algorithms/book/01-recursion.pdf

¡Gracias por su atención!