Análisis y diseño de algoritmos

3. Divide y vencerás

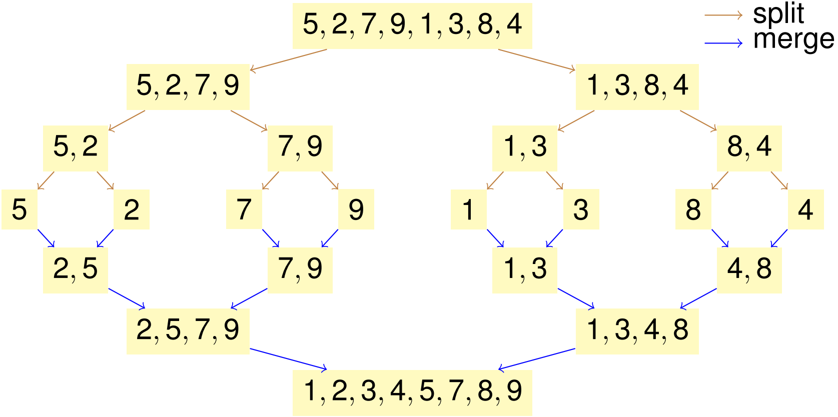
José Luis Verdú Mas, Jose Oncina,
Víctor M. Sánchez Cartagena

Dept. Lenguajes y Sistemas Informáticos
Universidad de Alicante

12 de enero de 2025

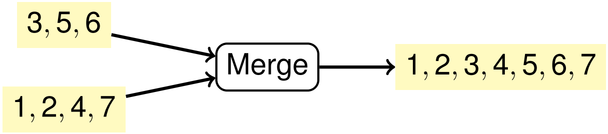
Algoritmo `Mergesort`: idea

- Ordenar de forma ascendente un vector V de n elementos.
- Solución usando el esquema “divide y vencerás”:



Algoritmo `Mergesort`: función `merge`

- El algoritmo `mergeSort` utiliza la función `merge` que obtiene un vector ordenado como fusión de dos vectores también ordenados



Algoritmo *Merge*

Merge

```
void merge( vector<int> &v, size_t first, size_t middel, size_t last ) {
    vector<int> v_merged;
    v_merged.reserve( last - first); // to make it faster

    size_t l = first, r = middel;
    while( l != middel && r != last ) {
        if( v[l] < v[r] ) {
            v_merged.push_back(v[l]); ++l;
        } else {
            v_merged.push_back(v[r]); ++r;
        }
    }
    for( ; l != middel; ++l ) v_merged.push_back(v[l]);
    for( ; r != last; ++r) v_merged.push_back(v[r]);
    // for( size_t i = first; i < last; ++i ) v[i] = v_merged[i-first];
    copy( begin(v_merged), end(v_merged), &v[first] );  // faster
}
```

¿Complejidad? $\Theta(n)$ (temporal y espacial); $n =$ `last - first`

Algoritmo *Merge*

Merge

```cpp
void merge( vector<int> &v, size_t first, size_t middel, size_t last ) {
    vector<int> v_merged;
    v_merged.reserve( last - first); // to make it faster

    size_t l = first, r = middel;
    while( l != middel && r != last ) {
        if( v[l] < v[r] ) {
            v_merged.push_back(v[l]); ++l;
        } else {
            v_merged.push_back(v[r]); ++r;
        }
    }
    for( ; l != middel; ++l ) v_merged.push_back(v[l]);
    for( ; r != last; ++r) v_merged.push_back(v[r]);
    // for( size_t i = first; i < last; ++i ) v[i] = v_merged[i-first];
    copy( begin(v_merged), end(v_merged), &v[first] );  // faster
}
```

¿Complejidad? $\Theta(n)$ (temporal y espacial); $n =$`last - first`

Algoritmo *Mergesort*

- Si la longitud de la lista es 0 ó 1, terminar. En otro caso
- Dividir la lista en dos sublistas de aproximadamente igual tamaño
- Ordenar cada sublista recursivamente aplicando `mergesort`
- Mezclar las dos sublistas ordenadas en una sola lista ordenada

Algoritmo *Mergesort*

Mergesort

```
void mergesort(
    vector<int> &v,
    size_t first,
    size_t last
) {

    if( last - first < 2 )
        return;

    size_t middel = first + ( last - first ) / 2;

    mergesort( v, first, middel);
    mergesort( v, middel, last);

    merge( v, first, middel, last );
}
```

Algoritmo *Mergesort*: Complejidad

- Talla: n $(n = \texttt{last} - \texttt{first})$
- Ecuación de recurrencia (coste exacto):

$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + 2T(\frac{n}{2}) & n > 1 \end{cases}$$

- Complejidad temporal: $\Theta(n \log n)$

Algoritmo *Mergesort*: Complejidad

- Talla: n $\qquad (n = \texttt{last} - \texttt{first})$
- Ecuación de recurrencia (coste exacto):

$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + 2T(\frac{n}{2}) & n > 1 \end{cases}$$

- Complejidad temporal: $\Theta(n \log n)$

¿Cuál es la complejidad espacial?

- Sin tener en cuenta la pila de recursión, viene dada por el mayor bloque de memoria auxiliar que necesita `merge`:

$$\max_{0 \leq i \leq \log_2 \frac{n}{2}} \frac{n}{2^i} = n$$

  - Notar que estos bloques de memoria auxiliar nunca coexisten.
- Si añadimos la pila, hay que añadir un término de orden $\log n$.
- **Complejidad espacial:** $\Theta(n)$

Algoritmo *Mergesort*: Complejidad

- Talla: n ($n = \texttt{last} - \texttt{first}$)
- Ecuación de recurrencia (coste exacto):

$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + 2T(\frac{n}{2}) & n > 1 \end{cases}$$

- Complejidad temporal: $\Theta(n \log n)$

¿Cuál es la complejidad espacial?

- Sin tener en cuenta la pila de recursión, viene dada por el mayor bloque de memoria auxiliar que necesita `merge`:

$$\max_{0 \leq i \leq \log_2 \frac{n}{2}} \frac{n}{2^i} = n$$

  - Notar que estos bloques de memoria auxiliar nunca coexisten.
- Si añadimos la pila, hay que añadir un término de orden $\log n$.
- **Complejidad espacial:** $\Theta(n)$

Algoritmo *Mergesort*: Complejidad

- Talla: n $\quad$ ($n = \texttt{last} - \texttt{first}$)
- Ecuación de recurrencia (coste exacto):

$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + 2T(\frac{n}{2}) & n > 1 \end{cases}$$

- Complejidad temporal: $\Theta(n \log n)$

¿Cuál es la complejidad espacial?

- Sin tener en cuenta la pila de recursión, viene dada por el mayor bloque de memoria auxiliar que necesita `merge`:

$$\max_{0 \leq i \leq \log_2 \frac{n}{2}} \frac{n}{2^i} = n$$

  - Notar que estos bloques de memoria auxiliar nunca coexisten.
- Si añadimos la pila, hay que añadir un término de orden $\log n$.
- **Complejidad espacial:** $\Theta(n)$

Algoritmo *Mergesort*: Complejidad

- Talla: n $(n = \texttt{last} - \texttt{first})$
- Ecuación de recurrencia (coste exacto):

$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + 2T(\frac{n}{2}) & n > 1 \end{cases}$$

- Complejidad temporal: $\Theta(n \log n)$

¿Cuál es la complejidad espacial?

- Sin tener en cuenta la pila de recursión, viene dada por el mayor bloque de memoria auxiliar que necesita `merge`:

$$\max_{0 \leq i \leq \log_2 \frac{n}{2}} \frac{n}{2^i} = n$$

  - Notar que estos bloques de memoria auxiliar nunca coexisten.
- Si añadimos la pila, hay que añadir un término de orden $\log n$.
- **Complejidad espacial:** $\Theta(n)$

Técnica de divide y vencerás

- Técnica de diseño de algoritmos que consiste en:
  - Descomponer el problema en subproblemas de menor tamaño que el original
  - Resolver cada subproblema de forma individual e independiente
  - Combinar las soluciones de los subproblemas para obtener la solución del problema original
- Consideraciones:
  - No siempre un problema de talla menor es más fácil de resolver
  - La solución de los subproblemas no implica necesariamente que la solución del problema original se pueda obtener fácilmente
- Aplicable si encontramos:
  - Forma de descomponer un problema en subproblemas de talla menor
  - Forma directa de resolver problemas menores a un tamaño determinado
  - Forma de combinar las soluciones de los subproblemas que permita obtener la solución del problema original

Esquema divide y vencerás (DC)

```
Solution DC( Problem p ) {
  if( is_simple(p) )
     return trivial(p);

  list<Solution> s;
  for( Problem q : divide(p) )
    s.push_back( DC(q) );

  return combine(s);
}
```

`Mergesort` como divide y vencerás

Particularización (instanciación) del esquema general para el caso de Mergesort:

- `is_simple`: $(\texttt{last} - \texttt{first} < 2)$
- `trivial`: retorno sin hacer nada
- `divide`: $m = \texttt{first} + (\texttt{last} - \texttt{first})/2$
- `combine`: merge(...)

Quicksort

```
void quicksort(
  vector<int> &v,
  size_t first,
  size_t last
) {

  if( last - first < 2 ) return;

  size_t p = first, l = last;
  while(p+1 < l) {
    if (v[p+1] < v[p]) {
      swap( v[p+1], v[p] );
      p++;
    } else {
      l--;
      swap( v[p+1], v[l] );
    }
  }

  quicksort(v, first, p);
  quicksort(v, p+1, last);
}
```

`Quicksort` como divide y vencerás

Particularización (instanciación) del esquema general para el caso de quickSort:

- `is_simple`: (`last - first < 2`)
- `trivial`: retorno sin hacer nada
- `divide`: cálculo de la posición del pivote y reparto del resto
- `combine`: no es necesario

Análisis de eficiencia

- Eficiencia: costes de logarítmicos a exponenciales. Depende de:
  - Nº de subproblemas (h)
  - Tamaño de los subproblemas
  - Grado de intersección entre los subproblemas
- Ecuación de recurrencia:
  - $g(n)$ = tiempos de `divide` y `combine` para un tamaño n (sin llamadas recursivas)
  - b = constante de división del tamaño de problema

$$T(n) = hT\left(\frac{n}{b}\right) + g(n)$$

  - Solución general (master theorem): suponiendo la existencia de un entero k tal que: $g(n) \in \Theta(n^k)$

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } h < b^k \\ \Theta(n^k \log_b n) & \text{si } h = b^k \\ \Theta(n^{\log_b h}) & \text{si } h > b^k \end{cases}$$

Análisis de eficiencia (caso especial)

- **Teorema de reducción**: los mejores resultados en cuanto a coste se consiguen cuando los subproblemas son aproximadamente del mismo tamaño (y no contienen subproblemas comunes).
- Si se cumple la condición del teorema de reducción ($b = h = a$)

$$T(n) = aT\left(\frac{n}{a}\right) + g(n) \qquad\qquad g(n) \in \Theta(n^k)$$

$$T(n) = \begin{cases} \Theta(n^k) & k > 1 \\ \Theta(n \log n) & k = 1 \\ \Theta(n) & k < 1 \end{cases}$$

Demostración

La relación de recurrencia a resolver es:

$$T(n) = \begin{cases} 1 & n = 1 \\ a\,T(\frac{n}{a}) + n^k & n > 1 \end{cases} \tag{1}$$

donde:

- $a \in \mathbb{N}, a > 1$ (ya que el n debe reducirse en las llamadas recursivas)
- $k \in \mathbb{R}^{\geq 0}$

$$
\begin{aligned}
T(n) &\stackrel{1}{=} a\,T\left(\frac{n}{a}\right) + n^k \\
&\stackrel{2}{=} a\left[a\,T\left(\frac{n}{a^2}\right) + \left(\frac{n}{a}\right)^k\right] + n^k \\
&\stackrel{2}{=} a^2\,T\left(\frac{n}{a^2}\right) + a\left(\frac{n}{a}\right)^k + n^k \\
&\stackrel{3}{=} a^2\left[a\,T\left(\frac{n}{a^3}\right) + \left(\frac{n}{a^2}\right)^k\right] + a\left(\frac{n}{a}\right)^k + n^k \\
&\stackrel{3}{=} a^3\,T\left(\frac{n}{a^3}\right) + a^2\left(\frac{n}{a^2}\right)^k + a\left(\frac{n}{a}\right)^k + n^k \\
&\dots \\
&\stackrel{j}{=} a^j\,T\left(\frac{n}{a^j}\right) + n^k\sum_{i=0}^{j-1}\left(\frac{1}{a^{k-1}}\right)^i
\end{aligned}
$$

La recursión termina cuando $\frac{n}{a^j} = 1$; por lo tanto, tomaremos $j = \log_a n$

Tomando $j = \log_a n$ se tiene:

$$T(n) = n + n^k \sum_{i=0}^{\log_a(n)-1} \left(\frac{1}{a^{k-1}}\right)^i$$

Para resolver el sumatorio hemos de distinguir tres casos en función del valor que tome $\frac{1}{a^{k-1}}$, al que llamaremos r:

1. $r = 1$, es decir, cuando se cumple $k = 1$:
   Se trata de una serie constante (o serie aritmética de diferencia 0)
2. $r < 1$, es decir, $k > 1$:
   Serie geométrica que converge
3. $r > 1$, es decir, $k < 1$:
   Serie geométrica que no converge

Recordemos que, en el caso de series geométricas, se tiene: $\sum_{i=0}^{M} r^i = \frac{r^{M+1}-1}{r-1}$

Resolviendo cada caso:

1. $k = 1$ $(r = 1)$
   Tenemos $\sum_{i=0}^{\log_a(n)-1} 1 = \log_a n$. Por lo tanto:

$$T(n) = n + n \log_a n \in \Theta(n \log n)$$

2. $k > 1$ $(r < 1)$
   Para valores grandes de j, $r^j \sim 0$, el sumatorio sería $\sim \frac{1}{1-r}$

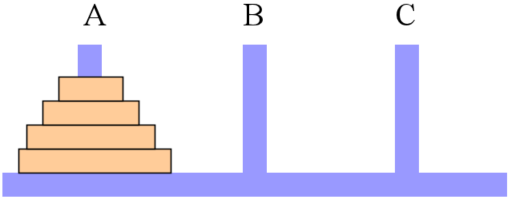
$$T(n) = n + n^k \frac{1}{1 - r} \in \Theta(n^k)$$

3. $k < 1$ $(r > 1)$
   Para valores grandes de j, el sumatorio sería $\sim \frac{r^j}{r-1}$,
   y como: $r^j = \left(\frac{1}{a^{k-1}}\right)^{\log_a n} = n^{\log_a\left(\frac{1}{a^{k-1}}\right)} = n^{\log_a a^{-(k-1)}} = n^{1-k}$

$$T(n) = n + n^k \frac{r^j}{r - 1} = n + n^k \frac{n^{1-k}}{r - 1} \in \Theta(n)$$

Las torres de Hanoi

- Colocar los discos de la torre A en la C empleando como ayuda la torre B
- Los discos han de moverse uno a uno y sin colocar nunca un disco sobre otro más pequeño



- ¿Cómo se afrontaría el problema?
- ¿Cuál sería la complejidad del algoritmo resultante?

Las torres de Hanoi: solución

- hanoi($n, A \overset{B}{\rightarrow} C$) es la solución del problema: mover los n discos superiores del pivote A al pivote C.
- Supongamos que sabemos mover $n-1$ discos: sabemos cómo resolver hanoi($n-1, X \overset{Y}{\rightarrow} Z$).
- También sabemos como mover 1 disco del pivote X al Z: hanoi($1, X \overset{Y}{\rightarrow} Z$), que es el caso trivial. Lo llamaremos mover($X \rightarrow Z$).
- Resolver hanoi($n, A \overset{B}{\rightarrow} C$) equivale a ejecutar:
  - hanoi($n-1, A \overset{C}{\rightarrow} B$)
  - mover($A \rightarrow C$)
  - hanoi($n-1, B \overset{A}{\rightarrow} C$)

Las torres de Hanoi: Complejidad (1)

Nótese que aquí la talla de los subproblemas no es $\frac{n}{a}$ sino $n - 1$:

- No se pueden aplicar las fórmulas generales de las transparencias anteriores
- El problema tiene una complejidad intrínseca peor que las descritas en las transparencias anteriores

Las torres de Hanoi: Complejidad (2)

- Ecuación de recurrencia para el coste exacto (asumiendo coste 1 para todas las operaciones de 1 disco):

$$T(n) = \begin{cases} 1 & n = 1 \\ 1 + 2T(n-1) & n > 1 \end{cases}$$

- Solución:

$$\begin{aligned} T(n) &\overset{1}{=} 1 + 2T(n-1) \\ &\overset{2}{=} 1 + 2(1 + 2T(n-2)) = 1 + 2 + 2^2 T(n-2) \\ &\overset{3}{=} 1 + 2 + 2^2(1 + 2T(n-3)) = 2^0 + 2^1 + 2^2 + 2^3 T(n-3) = \ldots \\ &\overset{k}{=} \sum_{i=0}^{k-1} 2^i + 2^k T(n-k) = 2^k - 1 + 2^k T(n-k) \end{aligned}$$

- Paramos cuando $n - k = 1$, o sea, en el paso $k = n - 1$,

$$T(n) = 2^n - 1 \in \Theta(2^n)$$

Ejercicios

1. Selección del k-ésimo mínimo
   - Dado un vector A de n números enteros diferentes, diseñar un algoritmo que encuentre el k-ésimo valor mínimo.
2. Búsqueda binaria o dicotómica
   - Dado un vector X de n elementos ordenado de forma ascendente y un elemento e, diseñar un algoritmo que devuelva la posición del elemento e en el vector X.
3. Cálculo recursivo de la potencia n-ésima.

Selección del k-ésimo mínimo

Formas de abordarlo (nótese que k **no es constante**):

1. Seleccionar k veces el elemento más pequeño del vector (sin volver a escoger elementos previamente seleccionados)
   - complejidad $\Theta(k \cdot n)$ (cuadrática);
   - ineficiente.
2. Construir un `heap` sobre el vector; realizar $k - 1$ operaciones `pop` y una operación `top`
   - complejidad $O(n + k \log n)$;
   - mucho mejor que lo anterior, pero aún mejorable.
3. El algoritmo `Quickselect` (basado en `Quicksort`)
   - complejidad $\Omega(n)$ y $O(n^2)$;
   - pero en promedio es $\Theta(n)$ y estadísticamente, **la más eficiente.**

¿Qué complejidad obtendríamos si k fuera una constante predeterminada?

El algoritmo `Quickselect`

- Notar que se trata de determinar el elemento que ocuparía la posición *k* si el vector estuviera ordenado.
  - Para ello, se realiza lo que se llama una `ordenación parcial` del vector basada en la posición *k*:
    - Esto es, recolocar los elementos del vector de forma que a la izquierda del elemento que ocupa la posición *k* solo estén los menores a este y, a la derecha solo los mayores (o iguales).
    - El que está en la posición *k* será el elemento buscado.
- Aprovechando la idea de Quicksort, donde cada pivote siempre queda en su posición definitiva, esa ordenación parcial consiste en realizar particionados sucesivos hasta que un pivote quede en la posición *k*.
  - Tras cada particionado, y en el caso de no haber tenido éxito, se vuelve a particionar el fragmento del vector donde está el elemento buscado.
    - Se sabe cuál es comparando *k* con la pos. donde queda el pivote.
    - El otro fragmento se descarta, dando lugar a la ordenación parcial.
  - Si nunca se tiene éxito, al llegar al caso base se habrá localizado el elemento buscado.

Quickselect: ordenación parcial basada en la pos. *k*

```
Elem quickselect( Elem v[], size_t first, size_t last, size_t k) {

  if( last == first )
      return v[k];

  size_t p = first; // pivote
  size_t = last;
  while(p < j) {
    if (v[p+1] < v[p]) {
      swap( v[p+1], v[p] );
      p++;
    } else {
      swap( v[p+1], v[j] );
      j--;
    }
  }

  if( k == p ) return v[k];
  else if ( k < p ) return quickselect(v,first,p-1,k);
  else return quickselect(v,p+1,last,k);
}
```

Quickselect

- Notar que, además de devolver el k-ésimo elemento más pequeño, quickselect realiza una ordenación parcial del vector según se ha descrito.
- En la `STL` de `C++` tenemos un algoritmo similar: `nth_element(...)` que además, permite cambiar la función de comparación predeterminada ($<$).

  `https://en.cppreference.com/w/cpp/algorithm/nth_element`

Complejidad temporal `Quickselect`

- Tamaño del problema: $n = \texttt{last} - \texttt{first}$
- Mejor caso
  - cuando $\texttt{k} = \texttt{p}$ tras el primer particionado;
  - complejidad: $\Omega(n)$
- Peor caso
  - cuando siempre ocurre $\texttt{k} \neq \texttt{p}$ (encuentra el k-ésimo elemento al llegar al caso base) y el pivote siempre queda en un extremo;
  - complejidad:
$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + T(n-1) & n > 1 \end{cases} \in O(n^2)$$
- Otro caso interesante
  - cuando siempre ocurre $\texttt{k} \neq \texttt{p}$ (encuentra el k-ésimo elemento al llegar al caso base) y el pivote siempre queda en el medio;
  - complejidad:
$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + T(\frac{n}{2}) & n > 1 \end{cases} \in \Theta(n)$$
  - coincide con el promedio estadístico cuando $n \to \infty$

Búsqueda binaria

```
size_t lower_bound( const auto &v, auto val, size_t first, size_t count
    ) {
  if( count == 0 ) return first;
  size_t step = count/2;
  size_t med = first + step;
  if( v[med] < val )
    return lower_bound( v, val, med+1, count - (step + 1));
  else
    return lower_bound( v, val, first, step );
}

size_t binary_search( const auto &v, auto val ) {
  size_t first = lower_bound( v, val, 0, v.size() );
  if( first < v.size() && v[first] == val )
    return first;
  else
    return NOT_FOUND;
}
```

- ¿Reconocéis en el algoritmo los componentes del *divide y vencerás*?

Búsqueda binaria

- Esta solución se puede ver como un *divide y vencerás* en el que
  - La operación `divide` viene representada por `med = first + step;`
  - El problema `is_simple` corresponde al caso `count == 0`
  - Sólo se resuelve uno de los dos subproblemas
    - divide y vencerás → reduce y vencerás
  - No es necesaria la operación `combine`

Búsqueda binaria: complejidad

- Ecuación de recurrencia para el caso peor:

$$T(n) = \begin{cases} 1 & n = 1 \\ 1 + T(\frac{n}{2}) & n > 1 \end{cases}$$

(agrupamos $n = 0$ en $n = 1$ porque no se produce división).

- Solución:

$$\begin{aligned} T(n) &\overset{1}{=} 1 + T(\frac{n}{2}) \\ &\overset{2}{=} 1 + 1 + T(\frac{n}{2^2}) = 2 + T(\frac{n}{2^2}) \\ &\dots \\ &\overset{k}{=} k + T(\frac{n}{2^k}) \end{aligned}$$

- La recursión termina cuando $\frac{n}{2^k} = 1$ por lo que tomamos $k = \log_2 n$.

$$T(n) \in O(\log n)$$

Cálculo de la potencia n-ésima

Si asumimos que multiplicar dos elementos de un determinado tipo tiene un coste constante, es posible calcular el valor de x^n en tiempo sublineal usando la siguiente recursión:

$$x^n = \begin{cases} 1 & n = 0 \\ x^{\frac{n}{2}} x^{\frac{n}{2}} & n \text{ es par} \\ x^{\frac{n-1}{2}} x^{\frac{n-1}{2}} x & n \text{ es impar} \end{cases}$$

Escribid un algoritmo para calcular eficientemente x^n.

- ¿Se puede evitar repetir operaciones?
- ¿Cuál es el coste asintótico del algoritmo resultante?