Divide and Conquer

Carlos Cotta

Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga

http://www.lcc.uma.es/∼ccottap

Comput Eng, Softw Eng, Comput Sci & Math – 2023-2024

UNIVERSIDAD
DE MÁLAGA

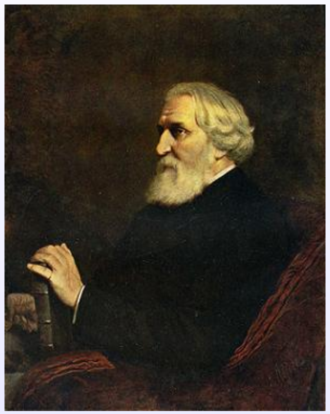
Divide and Conquer

Preliminaries

- Divide and Conquer is the simplest and best-known technique for algorithmic design.
- It is essential both from the point of view of recursion, and from the point of view of the top-down methodology.
- There exist several extremely efficient algorithms whose structure fits the Divide-and-Conquer approach.

"Divide et Impera"

The Whole and the Parts



Ivan Turgenev (1818–1883)
Russian writer

> Whatever man prays for, he prays for a miracle. Every prayer reduces itself to this: — *Great God, grant that twice two be not four.*
>
> *Prayer* (July 1881), Poems in Prose

An Illustrative (yet hardly practical) Example

We wish to compute the sum of the elements stored in a size-n array. A brute force approach would be as simple as:

Brute force sum

func Sum ($\downarrow A$: TArray):$\mathbb{N}$
variables i, s: $\mathbb{N}$
begin
  $s \leftarrow A[1]$
  for $i \leftarrow 2$ **to** n **do**
    $s \leftarrow s{+}A[i]$
  endfor
  return s
end

Can we think of another way of doing this?

An Illustrative (yet hardly practical) Example

If $n = 1$ computing the sum is trivial. If $n > 1$ we can divide the problem instance in two smaller instances of the same problem:

Divide-and-Conquer Sum

func Sum ($\downarrow A$: TArray, $\downarrow l, r$: $\mathbb{N}$):$\mathbb{N}$
variables m, s: $\mathbb{N}$
begin
  if $l{=}r$ **then** $s \leftarrow A[l]$
  else
    $m \leftarrow (l{+}r)/2$
    $s \leftarrow$ Sum(A, l, m) + Sum(A, $m{+}1$, r)
  endif
  return s
end

An Illustrative (yet hardly practical) Example

The complexity (measured as the number of sums) of the brute force algorithm is $t(n) = n - 1 \in \Theta(n)$. As to the Divide-and-Conquer algorithm we have:

$$t(n) = \begin{cases} 0 & n = 1 \\ 2t(n/2) + 3 & n > 1 \end{cases}$$

Using the Master Theorem, $\log_2 2 = 1$ and $f(n) \in \Theta(1)$, hence $t(n) \in \Theta(n)$ too, but the multiplicative constant is larger: $t(n) = 3n - 3$.

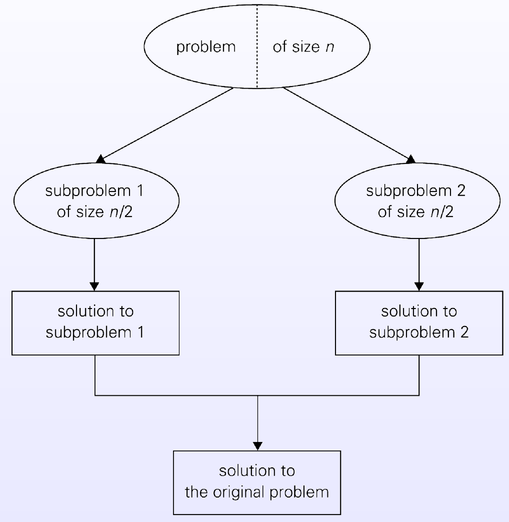
The Divide and Conquer approach has not provided any computational gain in this problem. It can provide huge improvements in other problems though.

Plan of Attack

Divide-and-Conquer algorithm approach the resolution of a problem as follows:

1. A problem instance is divided in several smaller instances (ideally these *chunks* should be about the same size).
2. These instances are recursively solved (or in some other way if they are small enough).
3. The solutions obtained for the smaller instances are combined to obtain the solution to the original problem instance.

Typical Scheme



Considerations

In general, a size-n instance is divided in $a \geqslant 1$ instances with size n/b each.

It is often the case that $a = b$, i.e., the instance is divided in $b > 1$ equal chunks, and each of them is independently solved.

In some cases, it may be possible that not all subproblems have to be solved (i.e., $a < b$), or that subproblems overlap to some extent (i.e., $a > b$).

If the cost of dividing and combining is $f(n)$, the total cost of the Divide-and-Conquer algorithm is $t(n) = at(n/b) + f(n)$.

Another Example: Binary Search

Binary Search

```
func BinarySearch (↓e: ℕ, ↓A: ARRAY [1..N] OF ℕ, ↓l, r: ℕ): 𝔹
variables m: ℕ
begin
    if l>r then return false
    else
        m ← (l+r)/2
        if A[m]=e then return true
        else
            if A[m]>e then return BinarySearch(e,A,l,m − 1)
            else return BinarySearch(e,A,m + 1,r)
            endif
        endif
    endif
end
```


Binary Search Complexity

Binary search

Let $t(n) = t(n/2) + 2$ be the number of comparisons performed by the binary search algorithm in the worst case ($t(0) = 0$). With regard to the Master Theorem we have here that $a = 1$, $b = 2$, and $f(n) = 1$.

Since $f(n) \in \Theta(1) = \Theta(n^0)$, and $d = \log_2 1 = 0$, we are in the second case of the theorem with $k = 0$. Therefore, $t(n) \in \Theta(n^d \log^{k+1} n) = \Theta(\log n)$.

Mergesort

John von Neumann (1903–1957)
Hungarian-American polymath

Mergesort is a good example of successful application of the Divide-and-Conquer approach.

It was developed by John von Neumann in 1945. It allowed to sort large collections of data in early computers which had little RAM.

Sorting by Merging

We have to sort an array $A[1..n]$. To this end:

- We divide the array in two halves $A[1..\lfloor n/2 \rfloor]$ and $A[\lfloor n/2 \rfloor + 1..n]$.
- We recursively sort these two halves. If any of them has a single element or is empty, the process is trivial and does not require any recursive call.
- We merge the two sorted halves in order to come up with a completely sorted array.

Sorting by Merging

Mergesort

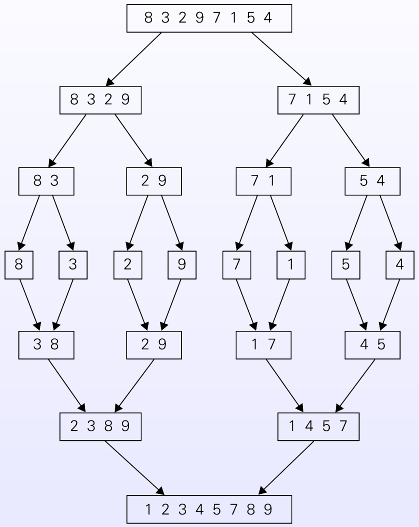
proc Mergesort ($\downarrow\uparrow A[1..n]$: TArray)
variables B, C: TArray
begin
  if $n>1$ **then**
    $B[1..\lfloor n/2 \rfloor] \leftarrow A[1..\lfloor n/2 \rfloor]$
    $C[1..\lceil n/2 \rceil] \leftarrow A[\lfloor n/2 \rfloor+1..n]$
    Mergesort(B)
    Mergesort(C)
    Merge(A, B, C)
  endif
end

Sorting by Merging

Mergesort

proc Merge ($\downarrow\uparrow A[1..n]$: TArray, $\downarrow B[1..p], C[1..q]$: TArray)
variables i, j, k: $\mathbb{N}$
begin
  $i \leftarrow 1$; $j \leftarrow 1$; $k \leftarrow 1$
  while $(j \leqslant p) \wedge (k \leqslant q)$ **do**
    if $B[j] \leqslant C[k]$ **then** $A[i] \leftarrow B[j]$; $j \leftarrow j+1$
    else $A[i] \leftarrow C[k]$; $k \leftarrow k+1$
    endif
    $i \leftarrow i+1$
  endwhile
  if $j>p$ **then** $A[i..n] \leftarrow C[k..q]$
  else $A[i..n] \leftarrow B[j..p]$
  endif
end

Mergesort at Work



(c) 2007 Pearson Addison-Wesley

Complexity of Mergesort

Let us initially focus on time complexity. There are two choices for the basic operation:

1. Comparisons between elements
2. Assignments of elements

Comparisons are exclusively performed in the procedure `Merge`.

Each iteration in the main loop of `Merge` implies a single comparison. In the worst case it performs $p+q-1=n-1$ iterations, and hence

$$t(n) = \begin{cases} 0 & n \leqslant 1 \\ 2t(n/2)+n-1 & n>1 \end{cases}$$

According to the Master Theorem ($\log_2 2 = 1$ and $f(n) \in \Theta(n)$) we have that $t(n) \in \Theta(n \log n)$.

Complexity of Mergesort

Focusing on element assignments, both procedure `Mergesort` and procedure `Merge` perform n operations each. Thus:

$$t(n) = \begin{cases} 0 & n \leqslant 1 \\ 2t(n/2)+2n & n>1 \end{cases}$$

Again, the Master Theorem indicates that $t(n) \in \Theta(n \log n)$.

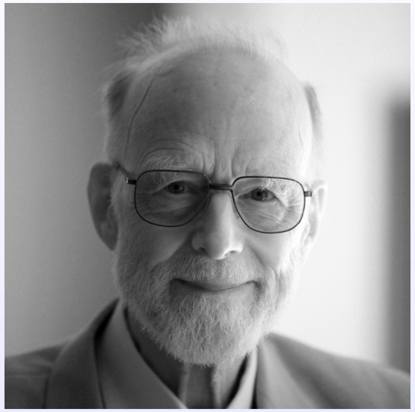
Complexity of Mergesort

Regarding space, procedure `Mergesort` uses two arrays whose combined size is n. The overall space consumption is therefore:

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

According to the Master Theorem we have $\log_2 1 = 0$, $f(n) \in \Theta(n) \in \Omega(n^\epsilon)$ for some $\epsilon > 0$ (for any $0 < \epsilon \leqslant 1$ actually), and $n/2 \leqslant cn$ for some $c < 1$ (for any $1/2 \leqslant c < 1$ actually). Hence, $t(n) \in \Theta(n)$.

Quicksort



Tony Hoare (1934)
British computer scientist

Quicksort is another successful example of the Divide-and-Conquer approach.

It is an extremely efficient sorting algorithm discovered by Sir C.A.R. "Tony" Hoare when working on a project for automatic translation between Russian and English.

Quicksort

We have to sort an array $A[1..n]$. To this end:

- We divide the array in two chunks $A[1..m-1]$ and $A[m+1..n]$, after having re-arranged the elements of the array so that
  1. $\forall i < m : A[i] \leqslant A[m]$
  2. $\forall i > m : A[i] > A[m]$
- We recursively sort these two chunks. The base case is trying to sort a chunk with one element (or none).

No further action is required after the recursive calls: the array is already sorted afterwards.

Quicksort

Quicksort

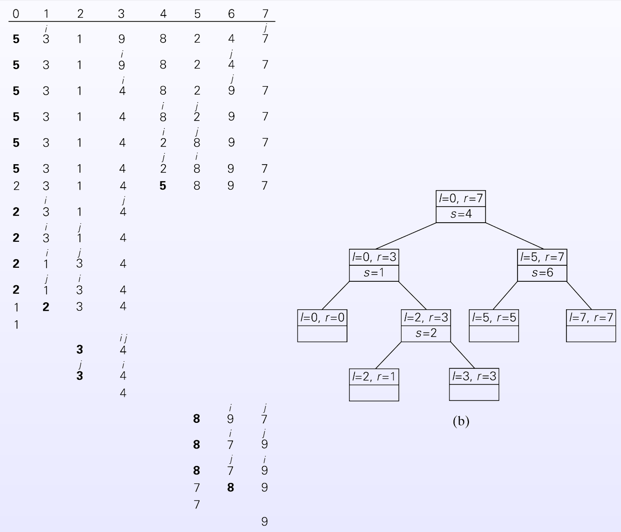
proc Quicksort ($\downarrow\uparrow A$: TArray, $\downarrow l, r$: $\mathbb{N}$)
variables m: $\mathbb{N}$
begin
    if $r > l$ **then**
        Divide(A, l, r, m)
        Quicksort(A, l, $m-1$)
        Quicksort(A, $m+1$, r)
    endif
end

Quicksort

Quicksort

proc Divide ($\downarrow\uparrow A$: TArray, $\downarrow l, r$: $\mathbb{N}$, $\uparrow m$: $\mathbb{N}$)
variables p: $\mathbb{N}$
begin
    $p \leftarrow l$
    repeat
        while $(l \leqslant r) \wedge (A[l] \leqslant A[p])$ **do** $l \leftarrow l+1$ **endwhile**
        while $(l \leqslant r) \wedge (A[r] > A[p])$ **do** $r \leftarrow r-1$ **endwhile**
        if $l < r$ **then** swap($A[l]$, $A[r]$) **endif**
    until $l \geqslant r$
    swap($A[p]$, $A[r]$)
    $m \leftarrow r$
end

Quicksort at Work



(c) 2007 Pearson Addison-Wesley

Complexity of Quicksort

Let us consider the time complexity of the algorithm. All element comparisons are done within procedure `Divide`.

Each call to `Divide` implies $\Theta(n)$ comparisons (n being the number of elements in the current chuck). In the best case, the outcome of `Divide` is two equal-size chunks:

$$t(n) = \begin{cases} 0 & n \leqslant 1 \\ 2t(n/2) + n & n > 1 \end{cases}$$

According to the Master Theorem ($\log_2 2 = 1$ and $f(n) \in \Theta(n)$) and hence $t(n) \in \Theta(n \log n)$.

Complexity of Quicksort

In the worst case the partition produces two extremely asymmetrical chunks, i.e., one with 0 elements and another one with $n-1$ elements:

$$t(n) = \begin{cases} 0 & n \leqslant 1 \\ t(n-1) + n & n > 1 \end{cases}$$

Solving the recurrence we have $t(n) \in \Theta(n^2)$ in the worst case.

If we consider element assignments as the basic operation, we come up with the same recurrence above, so the time complexity in that case is again $\Theta(n^2)$.

Complexity of Quicksort

In practice, The average complexity of Quicksort has the same order of growth than the best case, i.e., $\Theta(n \log n)$. More precisely, it can be shown to be

$$t_{avg}(n) \approx 2n \ln n \approx 1.39 n \log_2 n$$

There exist numerous strategies for minimizing the impact of the worst case: "smart choice" of the pivot, randomization, etc.

Integer Multiplication

Let us consider the problem of multiplying two large integers with a long number of digits each.

The sign can be easily dealt with, so we focus on positive integers.

Let us pick the following example: $A = 23$ and $B = 14$. To compute $A \cdot B$ by brute force:

$$
\begin{array}{ccc}
 & 2 & 3 \\
\times & 1 & 4 \\
\hline
 & 2 \cdot 4 & 3 \cdot 4 \\
2 \cdot 1 & 3 \cdot 1 & \\
\hline
2 & 8+3 & 12
\end{array}
$$

$$A \cdot B = 2 \cdot 10^2 + 11 \cdot 10^1 + 12 \cdot 10^0 = 200 + 110 + 12 = 322$$

If we consider that the product of two digits is the basic operation, we obviously perform n^2 operations.

A Different Approach

Anatoliy A. Karatsuba (1937-2008)
Russian mathematician

Karatsuba algorithm uses a different approach and results in asymptotically faster multiplication.

It was discovered by Anatoliy Karatsuba in 1960, during a seminar on cybernetics by Andreiy Kolmogorov.

A Different Approach

In more general terms, let $A = a_1 a_0$ and $B = b_1 b_0$. The product $C = AB$ can be expressed as:

$$C = c_2 10^2 + c_1 10^1 + c_0 10^0$$

where

$$
\begin{aligned}
c_2 &= a_1 b_1 \\
c_1 &= a_1 b_0 + a_0 b_1 \\
c_0 &= a_0 b_0
\end{aligned}
$$

Notice that

$$
\begin{aligned}
c_1 &= a_1 b_0 + a_0 b_1 = a_1 b_0 + a_0 b_1 + (a_1 b_1 + a_0 b_0) - (a_1 b_1 + a_0 b_0) = \\
&= (a_1 + a_0)(b_1 + b_0) - (a_1 b_1 + a_0 b_0) = \\
&= (a_1 + a_0)(b_1 + b_0) - (c_2 + c_0)
\end{aligned}
$$

We can try to exploit this result to speed-up the multiplication of larger integers.

The General Case: Karatsuba's Algorithm

Let $A = a_{n-1} a_{n-2} \cdots a_0$ and $B = b_{n-1} b_{n-2} \cdots b_0$. Now, let us divide A in half:

$$A = \overbrace{a_{n-1} a_{n-2} \cdots a_{n/2}}^{A_1} \; \overbrace{a_{n/2-1} \cdots a_1 a_0}^{A_0}$$

Analogously, B_1 is the left half of B and B_0 is its right half. Clearly:

$$A = A_1 10^{n/2} + A_0, \quad B = B_1 10^{n/2} + B_0$$

The same relationship seen before holds in this case:

$$
\begin{aligned}
C &= AB = \left(A_1 10^{n/2} + A_0\right)\left(B_1 10^{n/2} + B_0\right) = \\
&= (A_1 B_1)\, 10^n + (A_1 B_0 + A_0 B_1)\, 10^{n/2} + (A_0 B_0) = \\
&= C_2 10^n + C_1 10^{n/2} + C_0
\end{aligned}
$$

The General Case: Karatsuba's Algorithm

Once we have C_2 and C_0, we can compute C_1 as

$$C_1 = (A_1 + A_0)(B_1 + B_0) - (C_2 + C_0)$$

All calculations are done by means of sums and multiplications of integers with $n/2$ digits.

These multiplications can be done using the very same algorithm in a recursive way.

Recursion ends when the numbers have a single digit (or when they are small enough for a direct multiplication to be efficient).

The General Case: Knuth's Variant

When computing C_1 in Karatsuba's algorithm:

$$C_1 = (A_1 + A_0)(B_1 + B_0) - (C_2 + C_0)$$

we can end up with some irregularity, because even when A_0, A_1 have $n/2$ digits each, their sum can have $n/2 + 1$ digits (and the same applies to B_0, B_1).

Knuth proposed a variant to tackle this issue:

$$C_1 = C_0 + C_2 - (A_0 - A_1)(B_0 - B_1)$$

The subtraction $A_0 - A_1$ has exactly $n/2$ digits (but can be negative, so we have to take this into account in the procedure).

Complexity of the Algorithm

If we measure complexity in terms of the number of one-digit multiplications we have

$$t(n) = \begin{cases} 1 & n = 1 \\ 3t(n/2) & n > 1 \end{cases}$$

Applying the Master Theorem we have that $a = 3$, $b = 2$, $d = \log_b a = \log_2 3 \backsimeq 1.585$, and $f(n) = 0 \in \Theta(1)$. Since there exists $\epsilon > 0$ such that $f(n) \in O(n^{d-\epsilon})$ (any $\epsilon \in (0, d]$ would do), we have that $t(n) \in \Theta(n^{1.585})$.

This is a notable improvement with respect to the brute force approach which has complexity $\Theta(n^2)$.

Complementary Bibliography

A. Mohammed and M. Othman
"Comparative Analysis of Some Pivot Selection Schemes for Quicksort Algorithm",
Information Technology Journal 6:424-427, 2007
http://dx.doi.org/10.3923/itj.2007.424.427

A.A. Karatsuba
"The complexity of computations",
Proc. Steklov Institute of Mathematics 211:169–183, 1995

Image Credits

- Cartoon of Julius Cæsar: © Goscinny and Uderzo.
- Picture of Ivan Turgenev: Vasily Perov, frontispiece of one of the volumes in the 1978–Russian Academy edition of Turgenev's *Complete Works*
- Picture of John von Neumann: LANL (public domain)
- Picture of Sir C.A.R. Hoare: Rama, CC-BY-SA
- Picture of Anatoliy Karatsuba: AliceNovak, CC-BY-SA 3.0
- D&C scheme and Sorting algorithms: A. Levitin, © 2007 Pearson Addison-Wesley