ADA
Divide & Conquer

Angel Napa

January 1, 2026

1 Divide & Conquer Method

Method that uses recursion to solve problems. The recursion consists in 3 steps

- **Divide** the problem in one or more subproblems

  (In MergeSort: divide the array of size n in two subsequences of size $n/2$.)

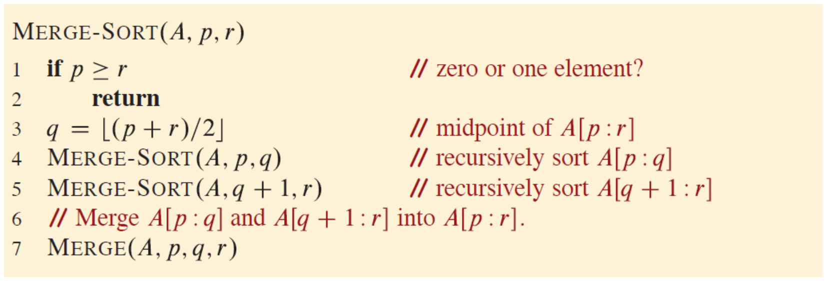
- **Conquer**: Solving the subproblems recursively.
  If the size is small enough, solve it directly

  (In MergeSort: sort the two subsequences using MergeSort.)

- **Combine** the subproblem solutions to form a solution to the original problem.

  (In MergeSort: Sorting the array with both halves already sorted)

Let's analyze the Merge-Sort algorithm. This algorithm uses Merge as a subtask, that receives $A[1 \dots n]$ and three indices p, q, r such that $A[p \dots q]$ and $A[q+1 \dots r]$ are already sorted, and sorts the subarray $A[p \dots r]$.



```
MERGE-SORT(A, p, r)
if p ≥ r                          // zero or one element?
    return
q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
MERGE-SORT(A, p, q)               // recursively sort A[p : q]
MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
// Merge A[p : q] and A[q + 1 : r] into A[p : r].
MERGE(A, p, q, r)
```

Figure 1: Cormen, Introduction to Algorithms

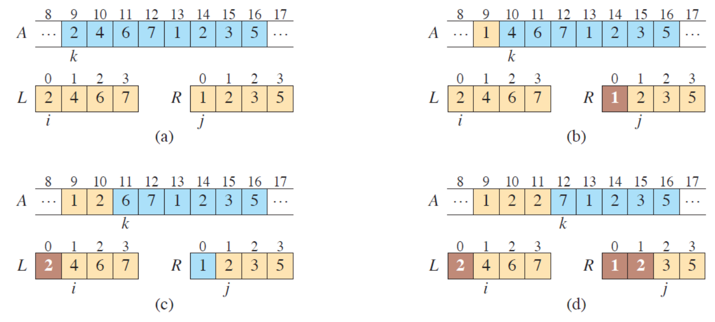


Figure 2: Cormen, Introduction to Algorithms

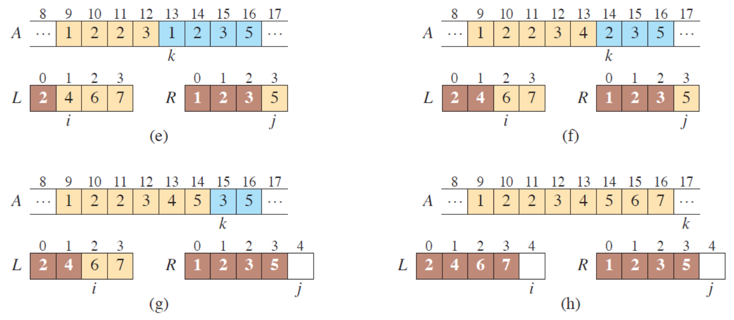
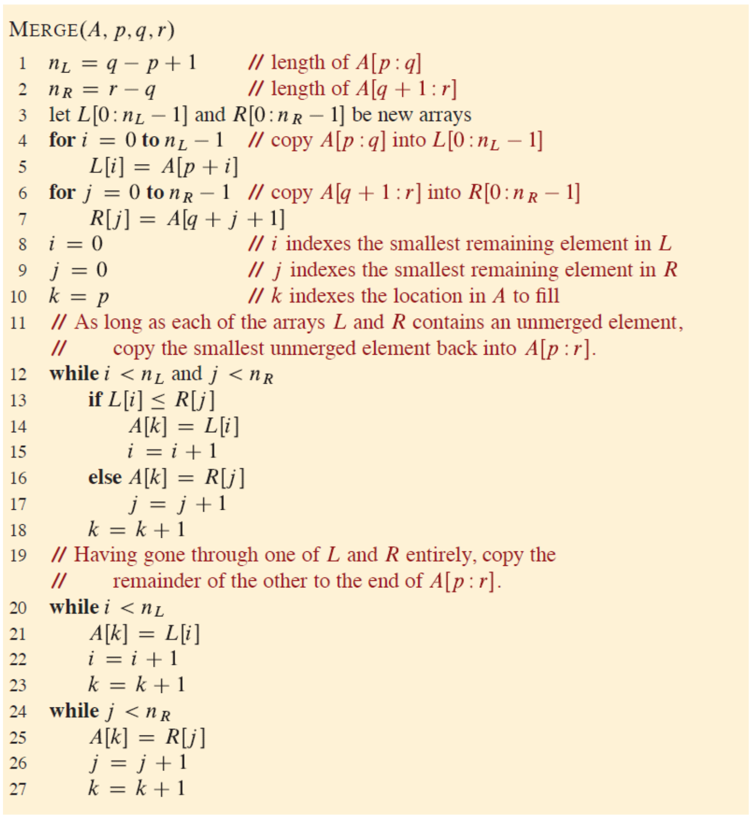


Figure 3: Cormen, Introduction to Algorithms



```
MERGE($A, p, q, r$)
1   $n_L = q - p + 1$          // length of $A[p:q]$
2   $n_R = r - q$              // length of $A[q+1:r]$
3   let $L[0:n_L - 1]$ and $R[0:n_R - 1]$ be new arrays
4   for $i = 0$ to $n_L - 1$   // copy $A[p:q]$ into $L[0:n_L - 1]$
5       $L[i] = A[p+i]$
6   for $j = 0$ to $n_R - 1$   // copy $A[q+1:r]$ into $R[0:n_R - 1]$
7       $R[j] = A[q+j+1]$
8   $i = 0$                    // $i$ indexes the smallest remaining element in $L$
9   $j = 0$                    // $j$ indexes the smallest remaining element in $R$
10  $k = p$                    // $k$ indexes the location in $A$ to fill
11  // As long as each of the arrays $L$ and $R$ contains an unmerged element,
    //     copy the smallest unmerged element back into $A[p:r]$.
12  while $i < n_L$ and $j < n_R$
13      if $L[i] \le R[j]$
14          $A[k] = L[i]$
15          $i = i + 1$
16      else $A[k] = R[j]$
17          $j = j + 1$
18      $k = k + 1$
19  // Having gone through one of $L$ and $R$ entirely, copy the
    //     remainder of the other to the end of $A[p:r]$.
20  while $i < n_L$
21      $A[k] = L[i]$
22      $i = i + 1$
23      $k = k + 1$
24  while $j < n_R$
25      $A[k] = R[j]$
26      $j = j + 1$
27      $k = k + 1$
```

Figure 4: Cormen, Introduction to Algorithms

Let's analyze the MERGE subtask.

Invariant: At the start of each iteration of the first **while** loop (lines 12–18), the subarray $A[p \ldots k-1]$ contains the $k-p$ smallest elements between $L[0 \ldots n_L - 1]$ and $R[0 \ldots n_R - 1]$, sorted. Also, $L[i]$ y $R[j]$ are the smallest elements of each array that are not in that subarray.

Proof:

- **Inicialization** $k = p$, then $A[p \ldots k-1] = \emptyset$
- **Maintenance**

  Case 1: $L[i] \le R[j]$. Then, we execute line 14. Since $A[p \ldots k-1]$ was sorted with the smallest elements, then $A[p \ldots k]$ will have the $k-p+1$ smallest elements. Case 2: $L[i] > R[j]$: similar.
- **Termination**

  Let $k = pos + 1$. Then $A[p \ldots k-1] = A[p \ldots pos]$ contains the $k - p = pos - p + 1$ smallest elements of $L[0 \ldots n_L - 1]$ y $R[0 \ldots n_R - 1]$. If $pos = r$, we already finished sorting all of the elements of the subarray $A[p \dots r]$. There is nothing to prove. Otherwise, we are just adding the remaining numbers, that are already sorted, in the back of the subarray. (for more formal proof we should prove another two invariants). Either way, we sort the entire subarray.

Execution time of the subtask Merge:

- Lines 1–2, 8–11, 19: constant time
- Lines 3–7: time $\Theta(n_1 + n_2) = \Theta(n)$.
- Lines 12–27: time $\Theta(n)$.

Initial call: Merge-Sort$(A, 1, A.length)$.

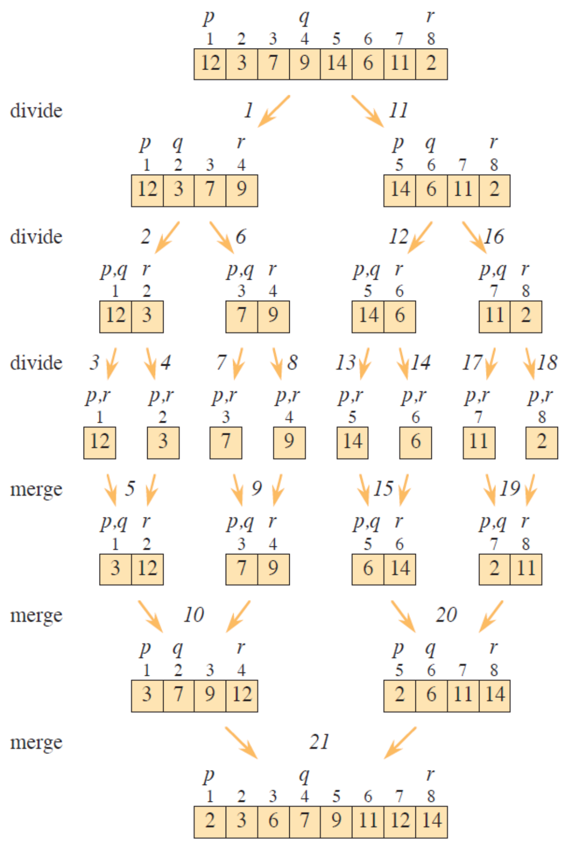


Figure 5: Cormen, Introduction to Algorithms.

2 Runtime Analysis

Generally speaking, when an algorithm call itself in its definition, we describe its execution time $T(n)$ by recurrence.

If the input size $n \leq n_0$, the solution takes a constant time: $T(n) = k$. If $n > n_0$ and a subproblems, each from size n/b, the solution takes:

$$T(n) = aT(n/b) + D(n) + C(n),$$

Where $D(n)$ is the amount of time used to divide the problems into subproblems and $C(n)$ is the time used to combine the subproblems.

Mergesort Analysis

Suppose for one moment that n is a power of 2. In this case

- $D(n)$ (divide). line 3: constant time k_1.
- $C(n)$ (combine): subtask Merge (line 7): $\Theta(n) = k_2 n$.

Then, $T(n) = c$ if $n = 1$, otherwise, if $n > 1$,

$$T(n) = 2T(n/2) + k_2 n + k_1 = 2T(n/2) + cn.$$

For simplicity (just this time), we are assuming that $k_2 n + k_1 = cn$. We have the following tree:

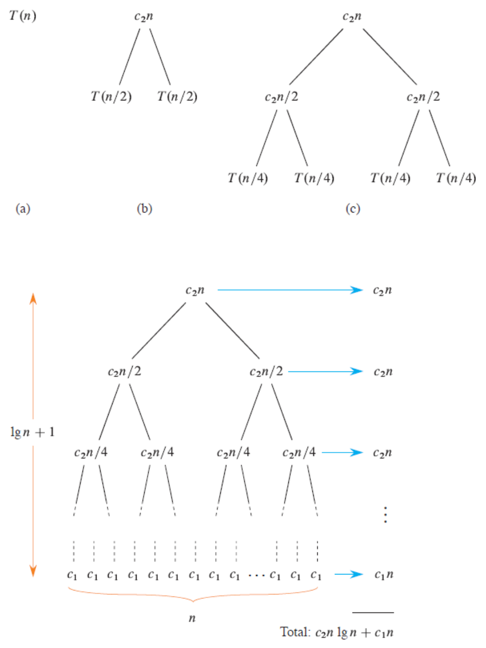


Figure 6: Tomada del libro Cormen, Introduction to Algorithms

Then,we can claim intuitively that $T(n) = cn \lg n + cn = \Theta(n \lg n)$.

This method gives us an intuition, but is not formal. In the next section we will see how to solve recurrences

3 Recurrences

A recurrence is a function that depends of itself in its definition. Stating and solving recurrences will help us find upper and lower bounds of the execution time of a divide & conquer algorithm

3.1 Explicit solution

We will see in this section a formal and more detailed way to solve recurrences.

Example 3.1. *Let $F : \mathbb{N} \to \mathbb{R}^+$ defined as*

$$F(n) = \begin{cases} 1 & n = 1 \\ 2F(n-1) + 1 & \textit{otherwise} \end{cases}$$

For example: $F(2) = 2F(1)+1 = 3, F(3) = 2F(2)+1 = 7, F(4) = 15$. What is the value of $F(n)$?

First solution

$$\begin{aligned} F(n) &= 2F(n-1) + 1 \\ &= 2(2F(n-2) + 1) + 1 \\ &= 4F(n-2) + 3 \\ &= 4(2F(n-3) + 1) + 3 \\ &= 8F(n-3) + 7 \\ &= \dots \\ &= 2^j F(n-j) + 2^j - 1 \\ &= 2^{n-1} F(1) + 2^{n-1} - 1 \\ &= 2 \cdot 2^{n-1} - 1 \\ &= 2^n - 1 \end{aligned}$$

Then $F(n) = 2^n - 1 = \Theta(2^n)$.

Second solution

For arbitrary k, we have:

$$F(k) - 2F(k-1) = 1$$

Thus, multiplying both sides by 2^{n-k} we have

$$\begin{aligned} 2^{n-k}F(k) - 2^{n-k+1}F(k-1) &= 2^{n-k} \quad \forall 2 \leq k \leq n \\ a_k - a_{k-1} &= 2^{n-k} \end{aligned}$$

Where a_k is defined as $2^{n-k}F(k)$. Adding up all possible k we have

$$\begin{aligned} F(n) - 2^{n-1} &= F(n) - 2^{n-1}F(1) \\ &= a_n - a_1 \\ &= \sum_{k=2}^{n} a_k - a_{k-1} \\ &= \sum_{k=2}^{n} 2^{n-k} \\ &= \sum_{j=0}^{n-2} 2^j \\ &= 2^{n-1} - 1 \end{aligned}$$

Adding both sides 2^{n-1} we also obtain here that $F(n) = 2^n - 1$.

Third solution

Let $G(n)$ be the function defined as:

$$G(n) = F(n) + 1$$

Using the recurrence of F, we can also find a recurrence for G:

$$G(n) = \begin{cases} 2 & n = 1 \\ 2G(n-1) & \text{otherwise} \end{cases}$$

This automatically tell us that G is a exponential form. so

$$G(n) = 2^n$$

We conclude that $F(n) = G(n) - 1 = 2^n - 1$.

Fourth solution

Using induction. This is left as an exercise to the reader.

Example 3.2. *Let $F : \mathbb{N} \to \mathbb{R}^+$ define as*

$$F(n) = \begin{cases} 1 & : n = 1 \\ F(n-1) + n & : \textit{otherwise} \end{cases}$$

For example $F(2) = F(1) + 2 = 3, F(3) = F(2) + 3 = 6$. What is the value of $F(n)$?

Solution

$$\begin{aligned} F(n) &= F(n-1) + n \\ &= F(n-2) + (n-1) + n \\ &= F(n-3) + (n-2) + (n-1) + n \\ &= \dots \\ &= F(n-j) + (n-j+1) + \cdots + n \\ &= F(1) + 2 + 3 + \cdots + n \\ &= 1 + 2 + 3 + \cdots + n \\ &= \frac{n(n+1)}{2} \end{aligned}$$

Then $F(n) = \Theta(n^2)$.

Example 3.3. *Let $F : \mathbb{N} \to \mathbb{R}^+$ defined as*

$$F(n) = \begin{cases} 1 & : n = 1 \\ 2F(\lfloor n/2 \rfloor) + n & : \textit{otherwise} \end{cases}$$

For example $F(2) = 2F(1) + 2 = 4, F(3) = 2F(1) + 3 = 5$. How can we find bounds of $F(n)$?

Solution First, suppose that n is a power of 2, this means $n = 2^j$ for some j. We have,

$$\begin{aligned} F(2^j) &= 2F(2^{j-1}) + 2^j \\ &= 2(2F(2^{j-2}) + 2^{j-1}) + 2^j \\ &= 2^2F(2^{j-2}) + 2^j + 2^j \\ &= 2^3F(2^{j-3}) + 2^j + 2^j + 2^j \\ &= 2^3F(2^{j-3}) + 3 \cdot 2^j \\ &= 2^iF(2^{j-i}) + i \cdot 2^j \\ &= 2^jF(1) + j \cdot 2^j \\ &= (j+1)2^j \\ &= (\lg n + 1)n. \end{aligned}$$

Now, suppose that n is an arbitrary positive integer. Let j such that $2^j \le n < 2^{j+1}$. Since F is increasing (see the end), we have that

$$F(n) < F(2^{j+1}) = (j+2) \cdot 2^{j+1} \le 3j \cdot 2^{j+1} = 6j \cdot 2^j \le 6n \lg n$$

(this last inequality holds because $j \leq \lg n$). furthermore,

$$F(n) \geq F(2^j) = (j+1)2^j = \frac{1}{2}(j+1)2^{j+1} > \frac{1}{2} n \lg n$$

(this last inequality holds because $j + 1 > \lg n$). We conclude that $F(n) = \Theta(n \lg n)$.

Finally, we will show that $F(n)$ is increasing. We have to prove that $F(n) < F(n+1)$ for all $n \geq 1$. The proof will use induction on n. If $n = 1$, we have that $F(1) = 1 < 4 = F(n+1)$.

If $n > 1$, we have 2 cases:
If n is even, we have that

$$F(n) < F(n) + 1 = 2F\left(\frac{n}{2}\right) + n + 1 = 2F\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + n + 1 = F(n+1)$$

Else, if n is odd, we have:

$$F(n) = 2F\left(\frac{n-1}{2}\right) + n < 2F\left(\frac{n+1}{2}\right) + n < 2F\left(\frac{n+1}{2}\right) + n + 1 = F(n+1)$$

Practice more exercises in the sheet of exercises.

3.2 Proofs by induction

Example 3.4. *Let $T : \mathbb{N} \to \mathbb{R}^+$ defined as*

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(\lfloor \frac{n}{2} \rfloor) + n & \textit{otherwise} \end{cases}$$

Prove by induction that $T(n) = O(n^2)$. Prove by induction that $T(n) = \Omega(n \lg n)$.

First we will prove by induction on n that $T(n) \leq 2n^2$ for $n \geq 1$. If $n = 1$, we have $T(1) = 1 \leq 2 \leq 2n^2$. If $n > 1$, we have that

$$\begin{aligned} T(n) &= 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \\ &\leq 4\left(\left\lfloor \frac{n}{2} \right\rfloor\right)^2 + n \\ &\leq 4\left(\frac{n}{2}\right)^2 + n^2 \\ &= 2n^2. \end{aligned}$$

Now we will prove by induction on n that $T(n) \geq \frac{1}{4} n \lg n$ for $n \geq 4$. If $n = 4$, we have that $T(4) = 12 \geq \frac{1}{4}4\lg 4 = \frac{1}{4}n\lg n$. If $n > 4$, we have that

$$\begin{aligned}
T(n) &= 2T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + n \\
&\geq 2\cdot\frac{1}{4}\left\lfloor\frac{n}{2}\right\rfloor \lg\left\lfloor\frac{n}{2}\right\rfloor + n \\
&\geq \frac{1}{2}\left(\frac{n}{2} - 1\right)\lg\left(\frac{n}{2} - 1\right) + n \\
&\geq \frac{1}{2}\left(\frac{n}{2} - 1\right)\lg\left(\frac{n}{4}\right) + n \\
&= \frac{1}{2}\left(\frac{n}{2} - 1\right)(\lg n - 2) + n \\
&= \frac{1}{4}(n\lg n) + n/2 + 2 - \lg n/2 \\
&\geq \frac{1}{4}(n\lg n).
\end{aligned}$$

Example 3.5. *Let $T : \mathbb{N} \to \mathbb{R}^+$ defined as*

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(\lfloor\frac{n}{2}\rfloor) + 1 & \textit{otherwise} \end{cases}$$

Prove by induction that $T(n) = O(n)$.

We will prove by induction on n that $T(n) \leq 2n-1$ for each $n \geq 1$. If $n = 1$, we have $T(1) = 1 = 2 - 1 \leq 2n - 1$. If $n > 1$, we have

$$\begin{aligned}
T(n) &= 2T(\left\lfloor\frac{n}{2}\right\rfloor) + 1 \\
&\leq 2\left(2\left\lfloor\frac{n}{2}\right\rfloor - 1\right) + 1 \\
&= 4\left\lfloor\frac{n}{2}\right\rfloor - 1 \\
&\leq 4\cdot\frac{n}{2} - 1 \\
&= 2n - 1.
\end{aligned}$$

Thus, $T(n) \leq 2n - 1 \leq 2n$ para $n \geq 1$, therefore $T(n) = O(n)$.

3.3 Master Theorem

Let $a \geq 1$, $b \geq 2$, $k \geq 0$, $n_0 \geq 1 \in \mathbb{N}$; $c \in \mathbb{R}^+$. Let $F : \mathbb{N} \to \mathbb{R}^+$ a nondecreasing function such that

$$F(n) = aF(n/b) + cn^k$$

for $n = n_0b^1, n_0b^2, n_0b^3, \ldots$.

It holds that

- If $\lg a/\lg b > k$ then $F(n) = \Theta(n^{\lg a/\lg b})$.

- If $\lg a / \lg b = k$ then $F(n) = \Theta(n^k \lg n)$.
- If $\lg a / \lg b < k$ then $F(n) = \Theta(n^k)$.

In particular, when $b = 2$ we have

- If $\lg a > k$ then $F(n) = \Theta(n^{\lg a})$.
- If $\lg a = k$ then $F(n) = \Theta(n^k \lg n)$.
- If $\lg a < k$ then $F(n) = \Theta(n^k)$.

4 Maximum Subarray Problem

- Input: Array $A[1..n]$ of integers (not necessarily positive).
- Output: Indices i, j such that the sum of the elements in $A[i..j]$ is maximum possible.

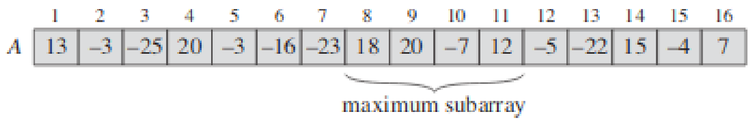


Figure 7: Taken from the book Cormen, Introduction to Algorithms

Brute force solution: Try all possible subsequences. As there are $\binom{n}{2}$ possibilities, the algorithm is $\Theta(n^2)$.

Can we do better?

Note that, given an array $A[low..high]$, a maximum subarray has three possibilities. It is

- Entirely in $A[low..mid]$
- Entirely in $A[mid + 1..high]$
- With a part in $A[low..mid]$ and the other part in $A[mid + 1..high]$

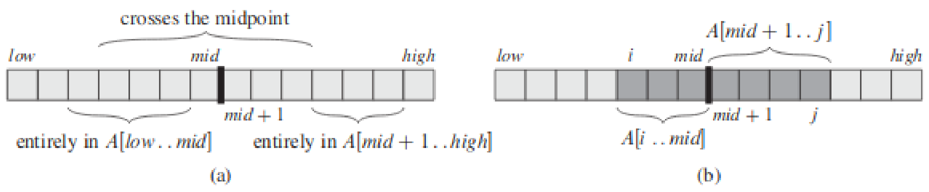


Figure 8: Taken from the book Cormen, Introduction to Algorithms

This observation allows us to design a divide and conquer algorithm for the problem.

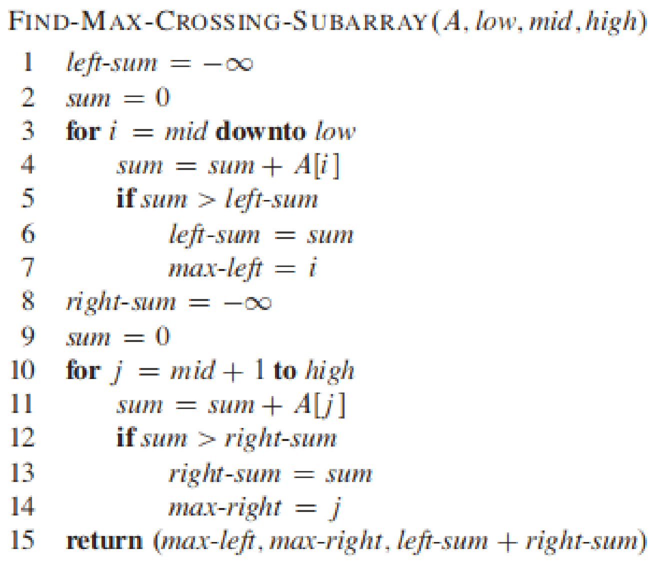
- **Divide** We split A into two subarrays of size $(high - low + 1)/2$
- **Conquer**: In each subarray, we find the corresponding maximum subarray.
- **Combine** We find which of the two subarrays from the recursive calls has the maximum sum.

  Then we need to compare this sum with a third candidate, which is the maximum subarray with a part in $A[low..mid]$ and the other part in $A[mid + 1..high]$

Next, we will see how to find this mentioned candidate in the "Combine" operation.

Input: An array A of integers, and three indices $low \leq mid < high$.

Output: Three integers $i, j, \sum_{k=i}^{j} A[k]$ such that $\sum_{k=i}^{j} A[k]$ is maximum for all i, j satisfying $i \leq mid < j$



```
FIND-MAX-CROSSING-SUBARRAY($A$, $low$, $mid$, $high$)
left-sum = $-\infty$
sum = 0
for $i$ = $mid$ downto $low$
    sum = sum + $A[i]$
    if sum > left-sum
        left-sum = sum
        max-left = $i$
right-sum = $-\infty$
sum = 0
for $j$ = $mid + 1$ to $high$
    sum = sum + $A[j]$
    if sum > right-sum
        right-sum = sum
        max-right = $j$
return (max-left, max-right, left-sum + right-sum)
```

Figure 9: Taken from the book Cormen, Introduction to Algorithms

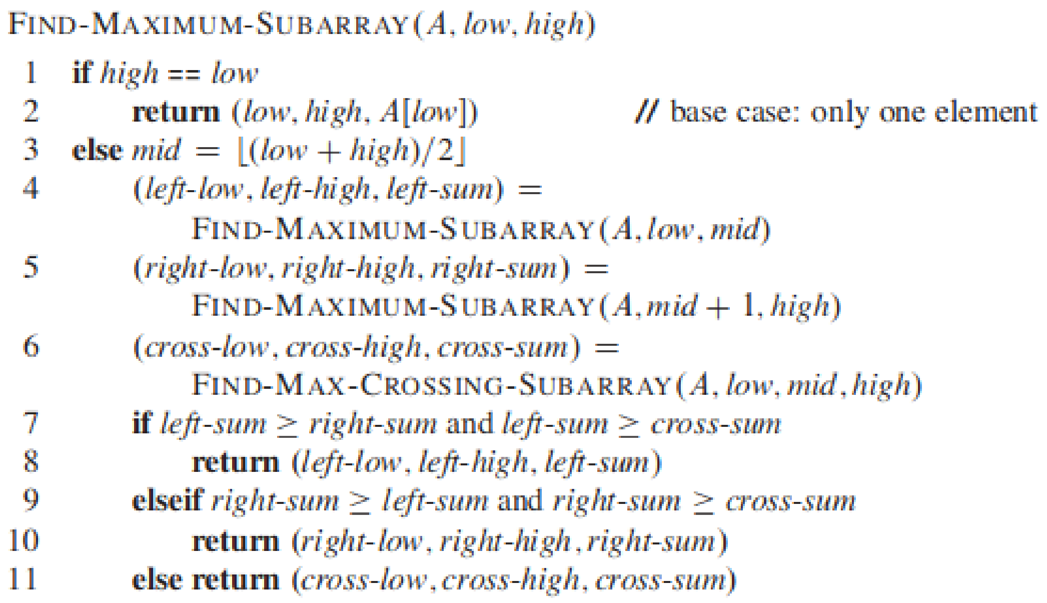
Execution time:

$T(n) = c_1 + (low - mid + 1)c_2 + (high - mid)c_3 = c_1 + c_4(low - mid + 1) = c_1 + 4n = \Theta(n)$

Now we will analyze the main algorithm.

Input: An array of integers $A[low..high]$.

Output: Three integers $i, j, \sum_{k=i}^{j} A[k]$ such that $\sum_{k=i}^{j} A[k]$ is maximum for all i, j



```
FIND-MAXIMUM-SUBARRAY(A, low, high)
1  if high == low
2      return (low, high, A[low])          // base case: only one element
3  else mid = $\lfloor (low + high)/2 \rfloor$
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 $\geq$ right-sum and left-sum $\geq$ cross-sum
8          return (left-low, left-high, left-sum)
9      elseif right-sum $\geq$ left-sum and right-sum $\geq$ cross-sum
10         return (right-low, right-high, right-sum)
11     else return (cross-low, cross-high, cross-sum)
```

Figure 10: Taken from the book Cormen, Introduction to Algorithms

Execution time of FIND-MAXIMUM-SUBARRAY. When $n = 1$, lines 1 and 2 are executed: time $c_1 + c_2$. When $n > 1$, we have $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c_3 + c_7 + c_8 + c_9 + c_{10} + c_{11} + k_1 n$

$$T(n) = \begin{cases} c & n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + k_1 n + k_2 & \text{otherwise} \end{cases}$$

So $T(n) = \Theta(n \log n)$. (To see this, apply the master theorem, since when n is a power of 2, we have $T(n) = 2T(n/2) + kn$), we have $a = b = 2, k = 1$, second case of the master theorem)

5 Multiplication of Natural Numbers

- Input: Two natural numbers a and b with n digits each.
- Output: The product $a \cdot b$

The usual algorithm:

```
    9999    A
    7777    B
   -----
   69993    C
  69993     D
 69993      E
69993       F
--------
77762223    G
```

In the previous figure, $A \cdot B = C + D + E + F = G$. The execution time is proportional to $4 + 4 + 4 + 4 = 16 = 4^2$.

This is a simple algorithm that can be described as follows:

```
Require: Two integers represented by $a[1..n], b[1..n]$
Ensure: The product $a \cdot b$
BASIC-MULTIPLICATION($a, b, n$)                        cost     times
1: total = 0                                           $c_1$    $1$
2: for $j = 1$ to $n$                                  $c_2$    $n+1$
3:     sum = 0                                         $c_3$    $n$
4:     for $i = 1$ to $n$                              $c_4$    $(n+1) \cdot n$
5:         sum = sum $\cdot$ 10 + $b[j] \cdot a[i]$    $c_5$    $n \cdot n$
6:     total = total $\cdot$10 + sum                   $c_6$    $n$
7: return total                                        $c_7$    $1$
```

It's clear that the execution time is $\Theta(n^2)$. Now we'll see how to improve this algorithm.

5.1 A Divide and Conquer Algorithm

Note that, given two natural numbers a and b with n digits, we can express them as

$$a = a_1 \cdot 10^m + a_2,$$

$$b = b_1 \cdot 10^m + b_2.$$

Where $m = \lceil n/2 \rceil$.

For example, $3213209842 = 32132 \cdot 10^5 + 09842$ or $953421412 = 9534 \cdot 10^5 + 21412$.

Therefore

$$\begin{aligned} a \cdot b &= (a_1 \cdot 10^m + a_2) \cdot (b_1 \cdot 10^m + b_2) \\ &= (a_1 b_1) \cdot 10^{2m} + (a_1 b_2 + a_2 b_1) \cdot 10^m + (a_2 b_2) \end{aligned}$$

This way, we can divide our original problem into four subproblems: multiplying a_1 by b_1, multiplying a_1 by b_2, multiplying a_2 by b_1, and multiplying a_2 by b_2.

We obtain the following divide and conquer algorithm:

Require: Two integers a and b with n digits, where n is a power of two, and both a and b do not contain zeros.
Ensure: The product $a \cdot b$

```
MULTIPLICATION-DC (a, b, n)                              cost       times
 1: if $n = 1$                                           $\Theta(1)$    1
 2:     return $a \cdot b$                               $\Theta(n)$    1
 3: $a_1 = \lfloor a/10^{n/2} \rfloor$                   $\Theta(n)$    1
 4: $a_2 = a \mod 10^{n/2}$                              $\Theta(n)$    1
 5: $b_1 = \lfloor b/10^{n/2} \rfloor$                   $\Theta(n)$    1
 6: $b_2 = b \mod 10^{n/2}$                              $\Theta(n)$    1
 7: $p$ = MULTIPLICATION-DC($a_1, b_1, n/2$)             $T(n/2)$       1
 8: $q$ = MULTIPLICATION-DC($a_1, b_2, n/2$)             $T(n/2)$       1
 9: $r$ = MULTIPLICATION-DC($a_2, b_1, n/2$)             $T(n/2)$       1
10: $s$ = MULTIPLICATION-DC($a_2, b_2, n/2$)             $T(n/2)$       1
11: return $p \cdot 10^n + (q + r) \cdot 10^{n/2} + s$   $\Theta(n)$    1
```

Execution time:

$$T(n) = \begin{cases} \Theta(n) & n = 1 \\ 4T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

By the Master Theorem, $\lg 4/\lg 2 = 2 > 1$. Thus, $T(n) = \Theta(n^2)$.

We see that MULTIPLICATION-DC does not improve BASIC-MULTIPLICATION.

5.2 Karatsuba's Algorithm

This algorithm will improve the previous recurrence by making only 3 recursive calls. Therefore, we'll have an execution time of $\Theta(n^{\lg 3/\lg 2}) = \Theta(n^{1.59})$.

The main observation is as follows.

Given two natural numbers a and b with n digits, we express them as in the previous subsection:

$$a = a_1 \cdot 10^m + a_2,$$
$$b = b_1 \cdot 10^m + b_2.$$

Where $m = \lceil n/2 \rceil$.

We have

$$\begin{aligned} a \cdot b &= (a_1 \cdot 10^m + a_2) \cdot (b_1 \cdot 10^m + b_2) \\ &= (a_1 b_1) \cdot 10^{2m} + (a_1 b_2 + a_2 b_1) \cdot 10^m + (a_2 b_2) \\ &= (a_1 b_1) \cdot 10^{2m} + ((a_1 + a_2)(b_1 + b_2) - a_1 b_1 - a_2 b_2) \cdot 10^m + (a_2 b_2) \end{aligned}$$

This way, we only need to calculate three products: $a_1 b_1$, $a_2 b_2$, and $(a_1 + a_2)(b_1 + b_2)$. The product $a_1 b_2 + a_2 b_1$ can be calculated as $(a_1 + a_2)(b_1 + b_2) - a_1 b_1 - a_2 b_2$.

Require: Two integers a and b with n digits, where n is a power of 2 and both a and b do not contain zeros

Ensure: The product $a \cdot b$

```
KARATSUBA (a, b)                                   cost        times
 1: if $n \le 1$                                   $\Theta(1)$    1
 2:    return $a \cdot b$                          $\Theta(n)$    1
 3: $a_1 = \lfloor a/10^{n/2} \rfloor$             $\Theta(n)$    1
 4: $a_2 = a \mod 10^{n/2}$                        $\Theta(n)$    1
 5: $b_1 = \lfloor b/10^{n/2} \rfloor$             $\Theta(n)$    1
 6: $b_2 = b \mod 10^{n/2}$                        $\Theta(n)$    1
 7: $p = \text{KARATSUBA}(a_1, b_1)$               $T(n/2)$       1
 8: $q = \text{KARATSUBA}(a_1 + a_2, b_1 + b_2)$   $T(n/2)$       1
 9: $s = \text{KARATSUBA}(a_2, b_2)$               $T(n/2)$       1
10: return $p \cdot 10^n + (q - p - s) \cdot 10^n + s$   $\Theta(n)$    1
```

Execution time:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 3T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

By the Master Theorem, $\lg 3 / \lg 2 > 1$. Thus $T(n) = \Theta(n^{\lg 3/\lg 2}) = \Theta(n^{1.59\cdots})$.

Observation: In the above pseudocode, we have assumed for simplicity that both a and b have n digits and that n is a power of 2. Otherwise, a sufficient trick for a good implementation is to pad with leading zeros if necessary.

6 Matrix Multiplication

Problem: Given two matrices $A = (a_{ij})$ and $B = (b_{ij})$ of dimensions $n \times n$, calculate $C = A \cdot B$. Recall that $C = (c_{ij})$ is defined as

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

We have the following algorithm to compute C.

Input: Two matrices A and B of dimensions $n \times n$

Output: $A \cdot B$

```
MULTIPLY (A, B)
1: for $i = 1$ to $n$
2:    for $j = 1$ to $n$
3:       $c_{ij} = 0$
4:       for $k = 1$ to $n$
5:          $c_{ij} = c_{ij} + a_{ik} b_{kj}$
6: return $C$
```

A simple analysis of the algorithm tells us that its execution time is $\Theta(n^3)$.

6.1 Divide and Conquer Algorithm

For simplicity, let's assume that n is a power of 2. We can partition each of the matrices into four parts. That is,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},$$

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},$$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Therefore,

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \quad (1)$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \quad (2)$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \quad (3)$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \quad (4)$$

From these equations, we can formulate the following divide and conquer algorithm.

Input: Two matrices A and B of dimensions $n \times n$ Output: $A \cdot B$

```
MULTIPLY-DC ($A, B$)
 1: if $n = 1$
 2:     $c_{11} = a_{11} \cdot b_{11}$
 3: else
 4:     Create auxiliary matrices
 5:     $C_{11}$=MULTIPLY-DC($A_{11}, B_{11}$)+ MULTIPLY-DC($A_{12}, B_{21}$)
 6:     $C_{12}$=MULTIPLY-DC($A_{11}, B_{12}$)+ MULTIPLY-DC($A_{12}, B_{22}$)
 7:     $C_{21}$=MULTIPLY-DC($A_{21}, B_{11}$)+ MULTIPLY-DC($A_{22}, B_{21}$)
 8:     $C_{22}$=MULTIPLY-DC($A_{21}, B_{12}$)+ MULTIPLY-DC($A_{22}, B_{22}$)
 9:     Fill $C$ from its parts
10: return $C$
```

We have the following recurrence.

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 8T(n/2) + \Theta(n^2) & \text{otherwise} \end{cases}$$

The Master Theorem tells us that $T(n) = \Theta(n^3)$. It has not improved the basic algorithm.

6.2 Strassen's Algorithm

Analogously to the Karatsuba algorithm, Strassen's algorithm attempts to perform only 7 recursive operations. Before the calls, we will create the following matrices:

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

Input: Two matrices A and B of dimensions $n \times n$ Output: $A \cdot B$

```
STRASSEN $(A, B)$
if $n = 1$
    $c_{11} = a_{11} \cdot b_{11}$
else
    Create auxiliary matrices
    $P_1$=MULTIPLY-DC$(A_{11}, S_1)$
    $P_2$=MULTIPLY-DC$(S_2, B_{22})$
    $P_3$=MULTIPLY-DC$(S_3, B_{11})$
    $P_4$=MULTIPLY-DC$(A_{22}, S_4)$
    $P_5$=MULTIPLY-DC$(S_5, S_6)$
    $P_6$=MULTIPLY-DC$(S_7, S_8)$
    $P_7$=MULTIPLY-DC$(S_9, S_{10})$
    $C_{11}$=$P_5 + P_4 - P_2 + P_6$
    $C_{12}$=$P_1 + P_2$
    $C_{21}$=$P_3 + P_4$
    $C_{22}$=$P_5 + P_1 - P_3 - P_7$
    Fill $C$ from its parts
return $C$
```

We have the following recurrence

$$
T(n) = \begin{cases} \Theta(1) & n = 1 \\ 7T(n/2) + \Theta(n^2) & \text{otherwise} \end{cases}
$$

The Master Theorem tells us that $T(n) = \Theta(n^{\lg 7})$.

7 Counting Inversions

- Input: Array $A[1..n]$ of distinct integers
- Output: Number of inversions. Where an *inversion* is a pair (i, j) such that $i < j$ and $A[i] > A[j]$

For example, let $A = [2, 4, 1, 3, 5]$. The inversions are $(1, 3), (2, 3), (2, 4)$

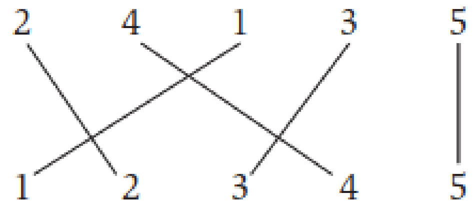


Figure 11: Taken from the book Kleinberg-Tardos, Algorithm Design

Naive Algorithm: Evaluate all possible ordered pairs and check if they are inversions.

Input: An array of distinct integers $A[1..n]$

Output: The number of inversions in A.

```
NAIVE-INVERSIONS(A, n)                          cost    times
1: total = 0                                    $c_1$   1
2: for $i = 1$ to $n - 1$                       $c_2$   $n$
3:     for $j = i + 1$ to $n$                   $c_3$   $\sum_{i=1}^{n-1} n - i + 1$
4:         if $A[i] > A[j]$                     $c_4$   $\sum_{i=1}^{n-1} n - i$
5:             total = total + 1                $c_5$   $\sum_{i=1}^{n-1} t_i$
6: return total                                 $c_6$   1
```

Where t_i is the number of times we execute line 5 during the i-th iteration. Clearly we have $0 \leq t_i \leq n - i$.

This algorithm, both in the worst and best case, consumes $\Theta(n^2)$ time. (An example of the worst case occurs if the array is sorted in decreasing order). Next, we will see a divide and conquer algorithm for this problem.

7.1 Divide and Conquer

We observe that, given the array $A[1..n]$, an inversion (i, j) can be in exactly one of these possibilities:

- $i, j \in \{1, \ldots, \lfloor n/2 \rfloor\}$
- $i, j \in \{\lfloor n/2 \rfloor + 1, \ldots, n\}$
- $i \in \{1, \ldots, \lfloor n/2 \rfloor\}$, $j \in \{\lfloor n/2 \rfloor + 1, \ldots, n\}$

That gives us the idea of a divide and conquer algorithm.

- **Divide** We divide A into two subarrays of sizes $\lfloor n/2 \rfloor, \lceil n/2 \rceil$
- **Conquer**: In each subarray, we find the number of inversions
- **Combine** We add this number with the number of inversions that have one element before or equal to $\lfloor n/2 \rfloor$ and the other after $\lfloor n/2 \rfloor$. This would give us the total number of inversions.

So, informally, we see that $T(n) = 2T(n/2) + C(n)$, where $C(n)$ is the time to combine. If we want the recurrence to be $\Theta(n \lg n)$ then $C(n)$ must be $\Theta(n)$.

But if we try to count the (i, j) such that $1 \leq i \leq \lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor + 1 \leq j \leq n$ with brute force, we can spend time proportional to $n/2 \cdot n/2 = \Theta(n^2)$. We need a more efficient algorithm.

Note that if the subarrays $A[1..\lfloor n/2 \rfloor]$ and $A[\lfloor n/2 \rfloor + 1 \ldots n]$ were already sorted, then we would only spend time $\Theta(n)$ due to the following observation:

If (i, j) is an inversion with $i \in \{1, \ldots, \lfloor n/2 \rfloor\}$ and $j \in \{\lfloor n/2 \rfloor + 1, \ldots, n\}$
then (i', j) is also an inversion for all $i' > i$ (5)

To prove (5), it suffices to observe that $A[i] > A[j]$ because (i, j) is an inversion and that $A[i'] > A[i]$ because A is sorted. The following subroutine takes care of this counting. It reminds us of the Merge function of Mergesort. Then, the following subroutine would do what is asked.

Input: An array of distinct integers $A[1..n]$ and three indices p, q, r such that $A[p..q]$ and $A[q+1..r]$ are sorted.

Output: The number of inversions (i, j) in A such that $i \in \{p, \ldots, q\}$, $j \in \{q+1, \ldots, r\}$. Also, it sorts the array $A[p..r]$.

```
CENTRAL-INVERSIONS($A, p, q, r$)
$n_1 = q - p + 1$
$n_2 = r - q$
Let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
for $i = 1$ to $n_1$
    $L[i] = A[p + i - 1]$
for $j = 1$ to $n_2$
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$L[n_2 + 1] = \infty$
$i = 1$
$j = 1$
$total = 0$
for $k = p$ to $r$
    if $L[i] > R[j]$
        $A[k] = R[j]$
        $total = total + (n_1 + 1 - i)$
        $j = j + 1$
    else
        $A[k] = L[i]$
        $i = i + 1$
return total
```

Note that CENTRAL-INVERSIONS is essentially the Merge subroutine of Mergesort, which consumes time $\Theta(n)$.

Finally, with the help of this subroutine, we design our main algorithm.

Input: An array of distinct integers $A[p..r]$

Output: The number of inversions in A

```
INVERSIONS-DC($A, p, r$)                                   cost                         times
1: if ($p == r$)                                          $c_1$                        1
2:     return 0                                           $c_2$                        0
3: $q = \left\lfloor \frac{r-p+1}{2} \right\rfloor$       $c_3$                        1
4: $total_1$ = INVERSIONS-DC($A, p, q$)                   $T(\lfloor n/2 \rfloor)$     1
5: $total_2$ = INVERSIONS-DC($A, q+1, r$)                 $T(\lceil n/2 \rceil)$       1
6: $total_3$ = CENTRAL-INVERSIONS($A, p, q, r$)           $kn$                         1
7: return $total_1 + total_2 + total_3$                   $c_5$                        1
```

Note that $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + kn$. Then, by the Master Theorem, $T(n) = \Theta(n \lg n)$.