06 - Divide and Conquer (part 1)

[KOMS119602] & [KOMS120403]

Design and Analysis of Algorithm (2021/2022)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 14-18 March 2022

Table of contents

Scheme of divide and conquer (DnC) algorithm

The principal of divide-and-conquer algorithm

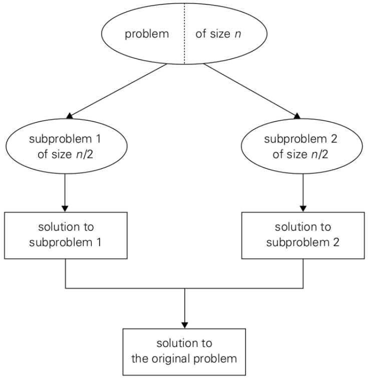
DIVIDE: breaking down the problem into two or more sub-problems that have the same or similar type, until these become simple enough to be solved directly. Ideally, the size of the sub-problems are equal.

CONQUER: solving each of the sub-problems, directly (if the size is small) or recursively (if the size is still big).

COMBINE: combining the solutions to the sub-problems to produce a solution to the original problem.

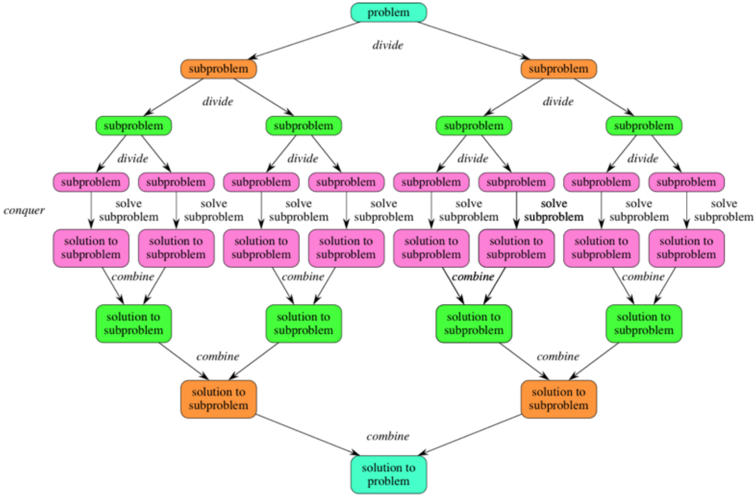
The principal of divide-and-conquer algorithm

In the most typical case of divide-and-conquer, a problem's instance of size n is divided into two instances of size $n/2$.



source: book of Anany Levitin

The principal of divide-and-conquer algorithm



source: `https://cdn.kastatic.org/ka-perseus-images/db9d172fc33b90e905c1213b8cce660c228bb99c.png`

Example of problems solvable by DnC algorithm

1. Binary search
2. Merge sort
3. Quick sort
4. Closest pair problem
5. *Convex hull problem* (haven't discussed yet)
6. Matrix multiplication
7. Strassen's algorithm
8. Karatsuba algorithm for fast multiplication
9. Multiplication of two polynomials

Divide and conquer vs Brute force

Study case: sum of array of integers

Problem

Given an array containing n integers $a_0, a_1, \ldots, a_{n-1}$.
Find $a_0 + a_1 + \cdots + a_{n-1}$.

Brute-force approach? add the element sequentially (one-by-one)

Divide-and-conquer:

- If $n = 1$, then return a_0;
- If $n > 1$, then recursively do the following: divide into two sub-arrays, then compute the sum of each sub-array.

$$a_0 + a_1 + \cdots + a_{n-1} = (a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1})$$

Which technique is more efficient?

Divide and conquer vs Brute force

Study case: sum of array of integers

Problem

Given an array containing n integers $a_0, a_1, \ldots, a_{n-1}$.
Find $a_0 + a_1 + \cdots + a_{n-1}$.

Brute-force approach? add the element sequentially (one-by-one)

Divide-and-conquer:

- If $n = 1$, then return a_0;
- If $n > 1$, then recursively do the following: divide into two sub-arrays, then compute the sum of each sub-array.

$$a_0+a_1+\cdots+a_{n-1} = (a_0+\cdots+a_{\lfloor n/2 \rfloor -1})+(a_{\lfloor n/2 \rfloor}+\cdots+a_{n-1})$$

Which technique is more efficient?
The brute force technique is better in this case.

Divide and conquer vs Brute force

- DnC is probably the best-known general algorithm design technique.
- Not every divide-and-conquer algorithm is necessarily more efficient than (even) a brute-force solution.
- Often, the time spent on executing the DnC algorithm is significantly smaller than solving a problem by a different method.
- The DnC approach yields some of the most important and efficient algorithms in CS.

Divide and conquer scheme

Algorithm 1 General scheme of divide-and-conquer

procedure DIVIDECONQUER(P: problem, n: integer)
    if $n \leq n_0$ **then** ▷ *P is small enough*
        SOLVE P
    else
        DIVIDE to r sub-problems $P_1, \ldots, P_r$ of size $n_1, \ldots, n_r$
        for each $P_1, \ldots, P_r$ **do**
            DIVIDECONQUER(P_i, n_i)
        end for
        COMBINE the solutions of $P_1, \ldots, P_r$ to solution of P
    end if
end procedure

DnC analysis of time complexity

Time complexity divide and conquer

$$T(n) = \begin{cases} g(n), & n \leq n_0 \\ T(n_1) + T(n_2) + \cdots + T(n_r) + f(n), & n \geq n_0 \end{cases}$$

- $T(n)$: the time complexity of problem P (of size n)
- $g(n)$: time complexity for SOLVE if n is small (i.e. $n \leq n_0$)
- $T(n_1) + T(n_2) + \cdots + T(n_r)$: time complexity to proceed each sub-problem
- $f(n)$: time complexity to DIVIDE the problem and COMBINE the solution of each sub-problem

An ideal situation is when the DIVIDE operation always produces two sub-problems of size half of the problem.

```
procedure DIVIDECONQUER(P: problem, n: integer)
    if n ≤ n_0 then                                  ▷ P is small enough
        SOLVE P
    else
        DIVIDE to 2 sub-problems P_1, P_2 of size n/2
        DIVIDECONQUER(P_1, n/2)
        DIVIDECONQUER(P_2, n/2)
        COMBINE the solutions of P_1, P_2 to solution of P
    end if
end procedure
```

Time complexity divide and conquer

If the instance is always divided into two sub-instances at each step, then:

$$T(n) = \begin{cases} g(n), & n \leq n_0 \\ 2T(n/2) + f(n), & n \geq n_0 \end{cases}$$

More generally, if the instance is always divided into $b \geq 1$ instances of equal size, where $a \geq 1$ instances need to be solved, then the complexity is given by:

$$T(n) = aT(n/b) + f(n)$$

The order of growth of its solution $T(n)$ depends on the values of the constants a and b and the order of growth of the function $f(n)$.

MinMax Problem:
An example of DnC algorithm

MinMax problem (1)

Problem

Given an array A of n integers. Find the min and max of the array simultaneously.

Example:

4	10	21	11	23	3	42	34	1

$$\min = 1$$
$$\max = 42$$

Figure: An array of integers, and the min & max of the array

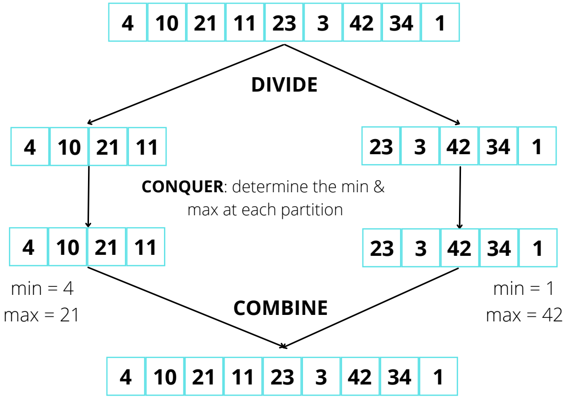
MinMax problem (2)

Algorithm 2 MinMax (brute-force)

procedure MinMax1($A[0..n-1]$: array, n: integer)
    min $\leftarrow A[0]$ ▷ *Assign the first element as the minimum*
    max $\leftarrow A[0]$ ▷ *Assign the first element as the maximum*
    for $i \leftarrow 1$ **to** $n-1$ **do**
        if $A[i] <$ min **then**
            min $\leftarrow A[i]$
        end if
        if $A[i] > max$ **then** max $\leftarrow A[i]$
        end if
    end for
end procedure

MinMax problem (3)

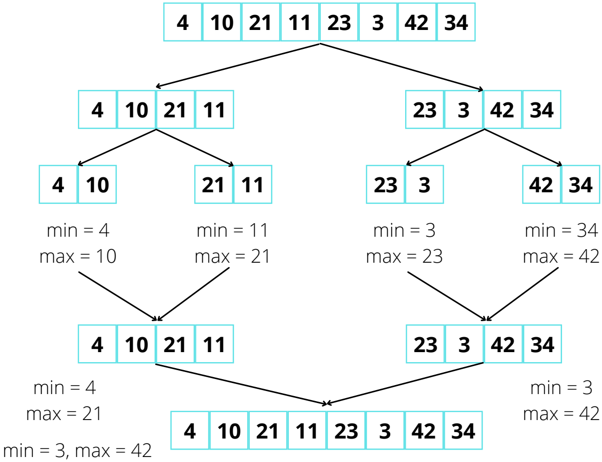
The scheme of Minmax with divide-and-conquer



Algorithm 3 MinMax (DnC)

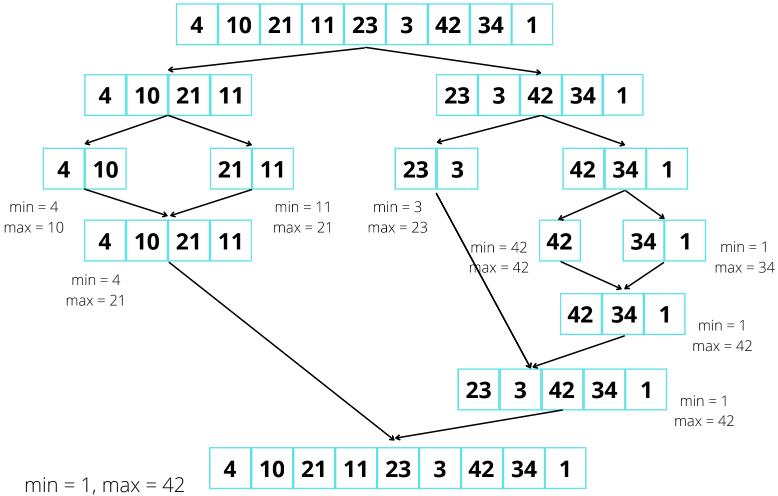
```
procedure MinMax2(input: A, i, j, output: min, max)
    if i = j then min ← A[i];  max ← A[i]
    else
        if i = j − 1 then                              ▷ The array has size 2
            if A[i] < A[j] then min ← A[i];  max ← A[j]
            else min ← A[j];  max ← A[i]
            end if
        else
            k ← (i + j) div 2                          ▷ Divide the array in the middle (position k)
            MinMax2(A, i, k, min1, max1)
            MinMax2(A, k + 1, j, min2, max2)
            if min1 < min2 then min ← min1
            else min ← min2
            end if
            if max1 < max2 then max ← max2
            else max ← max1
            end if
        end if
    end if
end procedure
```

Example:


4 10 21 11 23 3 42 34
4 10 21 11
23 3 42 34
4 10
21 11
23 3
42 34
min = 4
max = 10
min = 11
max = 21
min = 3
max = 23
min = 34
max = 42
4 10 21 11
23 3 42 34
min = 4
max = 21
min = 3
max = 42
4 10 21 11 23 3 42 34
min = 3, max = 42

MinMax problem (6)

Example:



min = 1, max = 42

MinMax problem (7): Time complexity

Compute the number of comparisons $T(n)$

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ 2 \cdot T(n/2) + 2 & \text{if } n > 2 \end{cases}$$

The explicit formula:

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + 2 \\ &= 2 \cdot (2 \cdot T(n/4) + 2) + 2 = 4 \cdot T(n/4) + (4 + 2) \\ &= 4 \cdot (2 \cdot T(n/8) + 2) + 4 + 2 = 8 \cdot T(n/8) + (8 + 4 + 2) \\ &\ \vdots \\ &= 2^{k-1} \cdot 1 + \sum_{i=1}^{k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 \\ &= n/2 + n - 2 \\ &= 3n/2 - 2 \ \in \mathcal{O}(n) \end{aligned}$$

MinMax problem (8): Time complexity

- Brute force $\textsc{MinMax1}$: $T(n) = 2n - 2$
- DnC $\textsc{MinMax2}$: $T(n) = 3n/2 - 2$

$$3n/2 - 2 < 2n - 2 \Leftrightarrow \text{ for } n \geq 2$$

The MinMax problem is more efficient to solve with DnC algorithm. But, asymptotically, both algorithms do not differ too much.

DnC-based sorting algorithms

Review

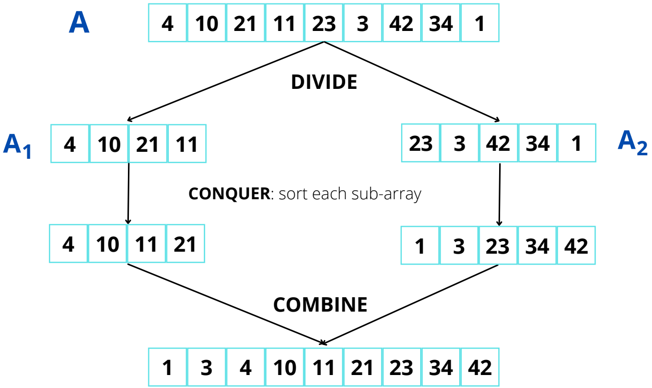
- Sorting problem: Given an ordorable array $A[0..n-1]$ (of size n). The array A is **sorted** if the elements in A is ordered in an ascending or descending order.
- Recall that the brute-force-based sorting algorithms such as *selection sort*, *bubble sort*, and *insertion sort* have time complexity $\mathcal{O}(n^2)$.
- Can we produce a sorting algorithm with a better time complexity using DnC approach?

DnC-based sorting (2)

Idea of DnC-based sorting procedures:

- If the array has size $n = 1$, then the array is sorted.
- If the array has size $n > 1$, then divide the array into two sub-arrays, then sort each sub-array.
- Merge the sorted sub-arrays into a sorted array. This is the result of the algorithm.

DnC-based sorting (3): scheme



Algorithm 4 DnC-based Sorting scheme

```
procedure DnCSort(A[0..n − 1]: array, n: integer)
    if size(A) = 1 then
        return A
    end if
    Divide(A, A_1, A_2) of size n_1 and n_2 resp.      ▷ n_2 = n − n_1
    DnCSort(A_1, n_1)                                   ▷ A_1 = A[0..n_1 − 1]
    DnCSort(A_2, n_2)                                   ▷ A_2 = A[n_1..n − 1]
    Combine(A_1, A_2, A)
end procedure
```

- Divide and Combine procedures depend on the problem.

Two approaches of DnC sorting algorithms

1. Easy split/hard join
    - The **Divide** step of the array is computationally easy
    - The **Combine** step is computationally hard
    - Examples: *Merge Sort, Insertion Sort*
2. Hard split/easy join
    - The **Divide** step of the array is computationally hard
    - The **Combine** step is computationally easy
    - Examples: *Quick Sort, Selection Sort*

DnC-based sorting (5)

Example

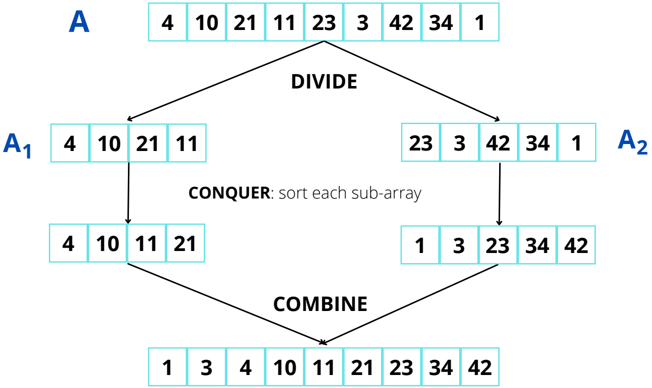
Given an array $A = [4, 12, 3, 9, 1, 21, 5, 1]$

1. **Easy split/hard join:** A is split based on the elements' positions
   - *Divide:* $A_1 = [4, 12, 3, 9]$ and $A_2 = [1, 21, 5, 2]$
   - *Sort:* $A_1 = [3, 4, 9, 12]$ and $A_2 = [1, 2, 5, 21]$
   - *Combine:* $A = [1, 2, 3, 4, 5, 9, 12, 21]$
2. **Hard split/easy join:** A is split based on the elements values
   - *Divide:* $A_1 = [4, 2, 3, 1]$ and $A_2 = [9, 21, 5, 12]$
   - *Sort:* $A_1 = [1, 2, 3, 4]$ and $A_2 = [5, 9, 12, 21]$
   - *Combine:* $A = [1, 2, 3, 4, 5, 9, 12, 21]$

Merge Sort

Merge Sort (1)

Basic idea:



Algorithm:

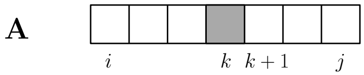
Input: array A, integer n
Output: array A sorted

1. If $n = 1$, then A is sorted
2. If $n > 1$, then
   - **Divide:** split A into two parts, each of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$
   - **Conquer:** recursively, implement $\textsc{MergeSort}$ in each sub-array
   - **Merge:** combine the sorted sub-arrays into the sorted array A

Merge Sort (3)

Algorithm 5 Merge Sort

procedure MERGESORT(A: ordorable array, i, j: integer) ▷ *i: starting index, j: last index, initialization: $i = 0, j = n - 1$ (i.e. the whole array A)*
    if $i = j$ **then** ▷ *length(A) $= 1$*
        return $A[i]$
    end if
    $k \leftarrow (i + j)$ **div** 2 ▷ *Divide the array into two*
    MERGESORT(A, i, k) ▷ *Sort the sub-array $A[i..k]$*
    MERGESORT($A, k + 1, j$) ▷ *Sort the sub-array $A[k + 1..j]$*
    MERGE(A, i, k, j) ▷ *Merge sorted $A[i..k]$ and $A[k + 1..j]$ into the sorted$A[i..j]$*
end procedure



Algorithm 6 “Merge” in MergeSort

procedure Merge(A, i, k, j) ▷ *$A[i..k]$ and $A[k+1..j]$ are sorted (ascending)*
output: Array $A[i..j]$ sorted (ascending)
declaration
  B : temporary array to store the merged values
end declaration

$p \leftarrow i;\ \ q \leftarrow k+1;\ \ r \leftarrow i$
while $p \leq k$ **and** $q \leq j$ **do** ▷ *while the left-array and the right-array are not finished*
  if $A[p] \leq A[q]$ **then**
    $B[r] \leftarrow A[p]$ ▷ *B is a temporary array to store the merged array; assign $A[p]$ (of left array) to B*
    $p \leftarrow p+1$
  else
    $B[r] \leftarrow A[q]$ ▷ *Assign $A[q]$ (of right array) to B*
    $q \leftarrow q+1$
  end if
  $r \leftarrow r+1$
end while ▷ *At this point, $p > k$ or $q > j$*

1: **while** $p \leq k$ **do** ▷ *If the left-array is not finished, copy the rest of left-array A to B (if any)*
2: $\quad B[r] \leftarrow A[p]$
3: $\quad p \leftarrow p + 1$
4: $\quad r \leftarrow r + 1$
5: **end while**
6: **while** $q \leq j$ **do** ▷ *If the right-array is not finished, copy the rest of right-array A to B (if any)*
7: $\quad B[r] \leftarrow A[q]$
8: $\quad q \leftarrow q + 1$
9: $\quad r \leftarrow r + 1$
10: **end while**
11: **for** $r \leftarrow i$ **to** j **do** ▷ *Assign back all elements of B to A*
12: $\quad A[r] \leftarrow B[r]$
13: **end for**
14: **return** A ▷ *A is in ascending order*
15: **end procedure**

Remark. the line numbering of the code is continued from the previous slide: 17, 18, 19, ...

Merge Sort (4): Procedure MERGE example

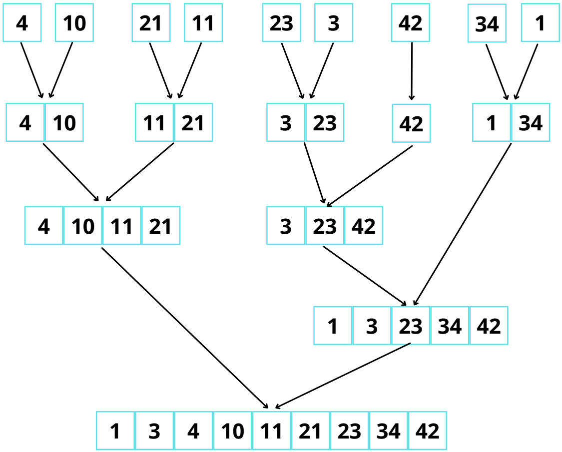


Figure: Example of MERGE procedure

Merge Sort (5): Procedure MergeSort example

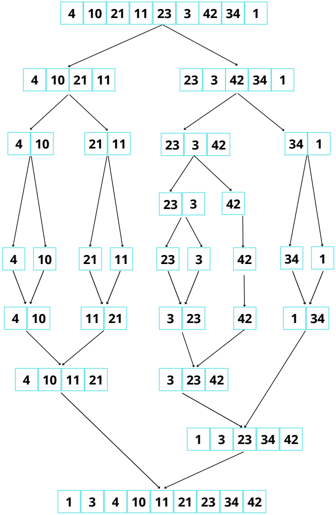


Figure: Example of MergeSort procedure

Merge Sort (4): Time complexity (TC)

Computing the TC of Merge Sort is similar to computing the TC of other recursive algorithms.

- The complexity of Merge Sort algorithm is measured from the number of comparisons of the elements in the array that is denoted by $T(n)$.
- The number of comparisons is in $\mathcal{O}(n)$, or cn for some constant c.
  (Here, we cannot compute exactly how many comparisons that we perform, because the MERGE *procedure involves many operations.)*
- So $T(n) = 2T(n/2) + cn$, for some constant c
- Hence:

$$T(n) = \begin{cases} 0, & n = 1 \\ 2T(n/2) + cn, & n > 1 \end{cases}$$

Merge Sort (4): Time complexity

- The explicit function can be computed by iteratively substituting the function. For simplification, we compute the special case, when $n = 2^k$ for some integer k.

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2(2T(n/4) + cn) + 3cn \\ &= 4(2T(n/8) + cn) + 3cn \\ &\vdots \\ &= 2^k T(n/2^k) + kcn \end{aligned}$$

Since $n = 2^k$, then $k = \log_2 n$. This yields:

$$T(n) = n \cdot T(1) + cn \cdot \log_2 n = 0 + cn \cdot \log_2 n \in \mathcal{O}(n \log n)$$

- This shows that Merge Sort has a better complexity ($\mathcal{O}(n \log n)$) than the brute-force-based sorting algorithms ($\mathcal{O}(n^2)$).

Recursive Insertion Sort

Special case of Merge Sort

Insertion sort (1): Principal

- This is an easy split/hard join-sorting.
- We have seen an iterative version of Insertion Sort algorithm. We can also view it in a recursive way: it is a special case of Merge Sort.
- The array is split into two sub-arrays, where the first sub-array only consists of one element, and the second sub-array consists of $n - 1$ elements.

Insertion sort (2): Pseudocode

Algorithm 7 Recursive Insertion Sort

procedure INSERTIONSORT(A: ordorable array, i, j: integers)
    output: A in ascending order

    if $i < j$ **then** ▷ *size(A)* > 1
        $k \leftarrow i$ ▷ *A is split at position i (initialize as* $i = 0$
        INSERTIONSORT(A, i, k) ▷ *sort the sub-array* $A[i..k]$
        INSERTIONSORT($A, k+1, j$) ▷ *sort the sub-array* $A[k+1..j]$
        MERGE(A, i, k, j) ▷ *merge the sub-array* $A[i..k]$ *and* $A[k+1..j]$ *into* $A[i..j]$
    end if
end procedure

Insertion sort (3): Pseudocode

Remark. Since the left sub-array is of size 1, then we may remove the INSERTIONSORT procedure for the left sub-array.

Algorithm 8 Insertion Sort

procedure INSERTIONSORT(A: ordorable array, i, j: integers)
    output: A in ascending order
    initialization: $i \leftarrow 0$, $j \leftarrow n - 1$
    if $i < j$ **then** $\triangleright$ *size(A)* > 1
        $k \leftarrow i$ $\triangleright$ *A is split at position i (initialize as $i = 0$*
        INSERTIONSORT$(A, k + 1, j)$ $\triangleright$ *sort the sub-array $A[k + 1..j]$*
        MERGE(A, i, k, j) $\triangleright$ *merge the sub-array $A[i]$ and $A[k + 1..j]$ into $A[i..j]$*
    end if
end procedure

Remark. The MERGE procedure can be replaced with the 'Insertion method' used in the iterative version.

Insertion sort (4): Example

Example: Suppose that we want to sort the array
$A = [4, 10, 21, 11, 23, 3, 42, 34, 1]$.

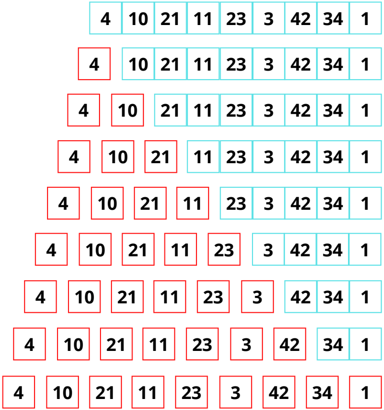


Figure: The 'Divide' and 'Conquer' steps

Insertion sort (5): Example

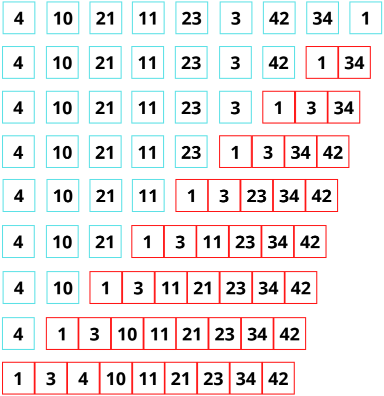


Figure: Applying the MERGE procedure

The recursive formula for the TC:

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

The explicit formula is obtained by recursive substitution:

$$\begin{aligned}
T(n) &= T(n-1) + cn \\
&= (T(n-2) + c(n-1)) + cn = T(n-2) + (cn + c(n-1)) \\
&= (T(n-3) + c(n-2)) + (cn + c(n-1)) = T(n-3) + \\
&\quad (cn + c(n-1) + c(n-2)) \\
&\vdots \\
&= cn + c(n-1) + c(n-2) + \cdots + 2c + a \\
&= c\left(\frac{1}{2} \cdot (n-1)(n+2)\right) \\
&= \frac{cn^2}{2} + \frac{cn}{2} + (a-c) \\
&= \mathcal{O}(n^2) \quad \text{(same as in the iterative version)}
\end{aligned}$$

Quick Sort

Click here

Recursive Selection Sort

Special case of Quick Sort

Selection sort (1): Principal

- This is a hard split/easy join-sorting.
- We have seen an iterative version of Selection Sort algorithm. We can also view it in a recursive way, as a special case of Quick Sort.
- The array is split into two sub-arrays, where the first sub-array only consists of one element, and the second sub-array consists of $n-1$ elements.

Remark. This method follows the Levitin's version of SELECTIONSORT (by looking for the min element). In the other version (if we look for the max element), the right sub-array has size one and the left sub-array has size $n-1$.

Selection sort (2): Pseudocode

Remark. Since the left sub-array is of size 1, then we do not need to recursive call InsertionSort for the *left* sub-array.

Algorithm 9 Recursive Selection Sort

procedure SelectionSort(A: ordorable array, i, j: integers)
    input: array $A[i..j]$
    output: $A[i..j]$ in ascending order

    initialization: $i \leftarrow 0$, $j \leftarrow n - 1$
    if $i < j$ **then** $\triangleright$ *size(A) > 1*
        Partition(A, i, j) $\triangleright$ *Partition the array into sub-arrays of size 1 and $n - 1$*
        SelectionSort$(A, i + 1, j)$ $\triangleright$ *Sort only the right sub-array*
    end if
end procedure

Selection sort (3): Pseudocode

Remark. Since the left sub-array is of size 1, then we do not need to recursive call InsertionSort for the *left* sub-array.

Algorithm 10 Partition procedure

procedure Partition(A: ordorable array, i, j: integers) ▷ *Partition $A[i..j]$ by looking for the minimum element and assign it to $A[i]$*
    idxMin $\leftarrow i$
    for $k \leftarrow i + 1$ **do to** j
        if $A[k] < A[\text{idxMin}]$ **then**
            idxMin $\leftarrow k$
        end if
    end for
    Swap($A[i], A[\text{idxMin}]$) ▷ *Exchange $A[i]$ and $A[idxMin]$*
end procedure

Selection sort (4): Example

Suppose that we want to sort the array:
$A = [4, 10, 21, 11, 23, 3, 42, 34, 1]$

4	10	21	11	23	3	42	34	1
1	10	21	11	23	3	42	34	4
1	3	21	11	23	10	42	34	4
1	3	4	11	23	10	42	34	21
1	3	4	10	23	11	42	34	21
1	3	4	10	11	23	42	34	21
1	3	4	10	11	21	42	34	23
1	3	4	10	11	21	23	34	42
1	3	4	10	11	21	23	34	42
1	3	4	10	11	21	23	34	42
1	3	4	10	11	21	23	34	42

- X Unsorted
- X Sorted
- X Current left sub-array

The recursive formula for the TC:

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

The explicit formula is obtained by substitution (*as in Insertion Sort*):

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= (T(n-2) + c(n-1)) + cn = T(n-2) + (cn + c(n-1)) \\ &= (T(n-3) + c(n-2)) + (cn + c(n-1)) = T(n-3)+ \\ &\quad (cn + c(n-1) + c(n-2)) \\ &\vdots \\ &= cn + c(n-1) + c(n-2) + \cdots + 2c + a \\ &= c\left(\frac{1}{2} \cdot (n-1)(n+2)\right) \\ &= \frac{cn^2}{2} + \frac{cn}{2} + (a-c) \\ &= \mathcal{O}(n^2) \quad \text{(same as in the iterative version)} \end{aligned}$$

Conclusion

What can we conclude from the four sorting algorithms?

Splitting the array into two **balanced** arrays (of size $n/2$ each) will result in the best algorithm performance (in the case of Merge Sort and Quick Sort, namely $\mathcal{O}(n \log n)$).

While the **unbalanced** split (into 1 element and $n-1$ elements) results in poor algorithm performance (in the case of Insertion sort and Selection sort, namely $\mathcal{O}(n^2)$).