CS-203 – LESSON 05 – DIVIDE-AND-CONQUER

GIUSEPPE TURINI

Kettering
UNIVERSITY

TABLE OF CONTENTS

STUDY GUIDE

Study Material:

- These slides.
- Your notes.
- * "Introduction to the Design and Analysis of Algorithms (3rd Ed.)", chap. 5, pp. 169-200.

Practice Exercises:

- Exercises from *: 5.1.1-3, 5.1.5-10, 5.2.1-8, 5.2.10, 5.3.1-8, 5.4.1-2, 5.4.8, 5.5.1-2, 5.5.4.

Additional Resources:

- "Introduction to Algorithms (3rd Ed.)", chap. 4, pp. 65-113.
- "Foundations of Algorithms (5th Ed.)", chap. 2, pp. 51-94.
- VisuAlgo

DIVIDE-AND-CONQUER

Divide-and-conquer is an algorithm design technique used by several efficient algorithms. Its general strategy works as follows:

Step 1: The original problem is divided into multiple subproblems of the same type but of smaller sizes (ideally all subproblems have equal size).

Step 2: All (or some) of the subproblems are solved (typically recursively, sometimes iteratively especially when size is very small).

Step 3: If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

Note: Divide-and-Conquer algorithms are usually implemented recursively; however, iterative implementations are possible and sometimes are also more efficient.

Note: Divide-and-Conquer algorithms are ideally suited for parallel implementations.

EXAMPLE: DIVIDE-BY-2

In Figure 1 we can see a flowchart describing visually a Divide-by-2 strategy.

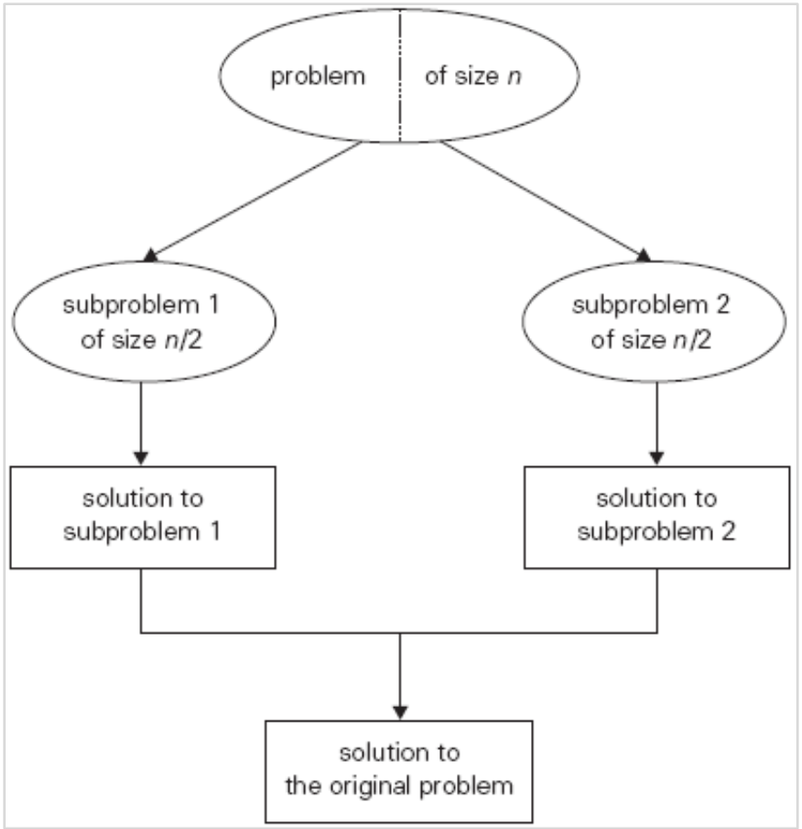

Figure 1. This flowchart illustrates the Divide-by-2 algorithm design.

GENERAL BASIC OPERATION COUNT

In general, in Divide-and-Conquer, a problem of size n can be divided into b subproblems (with b>1) of size ~n/b (usually integer size), with a of them (with $1 \leqslant a \leqslant b$) needing to be solved.

To simplify our analysis, let's assume that size n is a power of b (i.e., $n=b^k$).
Then, we get the following recurrence relation for the basic operation count C(n):

$$\textbf{general basic operation count recurrence in divide} - \textbf{and} - \textbf{conquer}$$
$$C(n) = a\, C(\, n/b\,) + f(n)$$

In the formula above, f(n) is the estimate of the work done to:

- Create the subproblems.
- Combine the subproblems solutions.

Note: Divide-and-Conquer algorithms are usually implemented recursively; however, iterative implementations are possible and sometimes are also more efficient.

MASTER THEOREM

To analyze Divide-and-Conquer algorithms we can use this theorem (Master Theorem).

Master Theorem

If $f(n) \in \Theta(n^d)$, where $d \geq 0$ in: $C(n) = a\,C(n/b) + f(n)$ then:

$$C(n) \in \begin{cases} \Theta(n^d), & \text{if } a < b^d. \\ \Theta(n^d \log n), & \text{if } a = b^d. \\ \Theta(n^{\log_b a}), & \text{if } a > b^d. \end{cases}$$

Note: analogous results hold for O and Ω notations.

Example: Suppose this is the basic operation count C(n) of a Divide-and-Conquer algorithm.

$$C(n) = \begin{cases} 1, & \text{if } n = 0. \\ 4\,C(n/2) + n^3, & \text{if } n > 0. \end{cases}$$

Then, by applying the Master Theorem, we can discover immediately that the basic operation count C(n) is in the efficiency class $\Theta(n^3)$.

ADD INTEGERS IN RANGE: ALGORITHM

Problem: Add all integers from a to b (with a ≤ b).

Add Integers in Range by Divide-by-2

To solve this problem using a Divide-and-Conquer strategy we have to consider 2 cases:

- **If a=b:** we simply return a.
- **If a>b:** we can divide the problem into 2 subproblems: subproblem 1, add integers from a to ⌊(a+b)/2⌋; and subproblem 2, add integers from ⌊(a+b)/2⌋+1 to b.

$$a + (a+1) + \cdots + (b-1) + b = \left(a + \cdots + \left\lfloor \frac{(a+b)}{2} \right\rfloor \right) + \left(\left\lfloor \frac{(a+b)}{2} \right\rfloor + 1 + \cdots + b \right)$$

Then, to solve each subproblem we can apply the same approach (recursively, top-down).

ADD INTEGERS IN RANGE: ALGORITHM 2

The recursive definition representing the computation done by this algorithm is this:

$$\text{Add}(a,b) = \begin{cases} a, & \text{if } a = b. \\ \text{Add}\left(\text{a}, \left\lfloor \frac{(a+b)}{2} \right\rfloor\right) + \text{Add}\left(\left\lfloor \frac{(a+b)}{2} \right\rfloor + 1, \text{b}\right), & \text{if } a > b. \end{cases}$$

Note: Obviously, this is not an an efficient way to solve this problem: because the splitting into subproblems does not decrease the amount of work done.

Note: Not every Divide-and-Conquer algorithm is necessarily efficient, but often the time spent in splitting into subproblems and merging the subproblems solutions turns out to be significantly smaller than solving the original problem by a different method.

ADD INTEGERS IN RANGE: IMPLEMENTATION

This is a Java implementation of the Add Integers In Range by Divide-by-2 algorithm (recursive implementation, top-down).

```
public static int AddIntegersInRangeRec( int a, int b ) {
   // BASE CASE: input range trivial.
   if( a == b ) { return a; }
   else {
      // Solving 2 subproblems.
      int resSP1 = AddIntegersInRangeRec( a, (a+b)/2 );
      int resSP2 = AddIntegersInRangeRec( ((a+b)/2)+1, b );
      // Merging subproblems solutions.
      return resSP1 + resSP2;
   }
}
```

ADD INTEGERS IN RANGE: ANALYSIS

The input size is the item-size of the input range: b-a+1. Let's call it n = b-a+1.

The basic operation is the addition (+) for the main stage of the algorithm, and the conditional (if) for the basic case check.

The basic operation count C(n) depends only on the input size n (no best-case, no worst-case).

The recursive definition representing the basic operation count C(n) is this:

$$\mathrm{C}(n) = \begin{cases} 1, & \text{if } n = 1 \text{ (or } a = b\text{).} \\ \mathrm{C}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathrm{C}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 + 1, & \text{if } n > 1 \text{ (or } a > b\text{).} \end{cases}$$

Note: The floor (round-down) and ceil (round-up) operators are used differently in the recursive definition of the algorithm computation and in the recursive definition of the basic operation count.

ADD INTEGERS IN RANGE: ANALYSIS 2

To simplify our analysis, initially we consider only input size (n) values allowing perfect splits, that is: we will consider only n values that are a power of 2 (i.e., $n=2^k$).

So, our basic operation count C(n) can be simplified into $C_{smp}(n)$.

$$\text{if } n = 2^k \text{ then: } \quad \mathrm{C_{smp}}(n) = \begin{cases} 1, & \text{if } n = 1 \ (\text{or } a = b). \\ 2\mathrm{C_{smp}}\left(\frac{n}{2}\right) + 2, & \text{if } n > 1 \ (\text{or } a > b). \end{cases}$$

Now, to this recurrence relation we can apply the Master Theorem using:

- **d=0:** because f(n) = 2, therefore f(n) is in $\Theta(1) = \Theta(n^0)$.
- **a=2, and b=2:** because we split into 2 subproblems and both of them have to be solved.

So, we apply the Master Theorem using the case $a > b^d$ (i.e., $2 > 2^0$), obtaining this result:

$$\mathrm{C_{smp}}(n) \in \Theta\left(n^{\log_b a} \right) = \Theta\left(n^{\log_2 2} \right) = \Theta(n)$$

ADD INTEGERS IN RANGE: ANALYSIS 3

Now we know the efficiency class ($\Theta(n)$) of the count $C_{smp}(n)$ when $n=2^k$.

This result allows us to use the Smoothness Rule (because the efficiency class function n is smooth), so we can extend the result obtained for $n=2^k$ to any value of n.

Note: By using the Master Theorem and then the Smoothness Rule, we were able to find the efficiency class of the basic operation count without actually converting it into a closed-form expression.

Note: Remember that, determining the efficiency class is different (weaker) than determining the exact equation of the basic operation count (stronger).

To improve the accuracy of our analysis and to double check our current result (the efficiency class just found) we can still convert the basic operation count $C_{smp}(n)$ into a closed-form expression by using the method of Backward Substitutions.

ADD INTEGERS IN RANGE: ANALYSIS 4

To improve the accuracy of our analysis and to double check our current result (the efficiency class just found) we can still convert the basic operation count $C_{smp}(n)$ into a closed-form expression by using the method of Backward Substitutions.

$$C_{smp}(n) = 2\,C_{smp}(n/2) + 2 =$$

apply recurrence

$$= 2\left[\,2\,C_{smp}(n/4) + 2\,\right] + 2 = 4\,C_{smp}(n/4) + 4 =$$

$$= 4\left[\,2\,C_{smp}(n/8) + 2\,\right] + 4 = 8\,C_{smp}(n/8) + 6 = \cdots$$

$$\Downarrow$$

create pattern $\quad C_{smp}(n) = 2^i\,C_{smp}\left(n/2^i\right) + 2i, \quad \text{with } i \in [0, \log_2 n]$

$$\Downarrow$$

solve pattern $\quad 2^i\,C_{smp}\left(n/2^i\right) + 2i = n\,C_{smp}(1) + 2\log_2 n = n + 2\log_2 n \in \Theta(n)$

Now we have a closed-form expression for the basic operation count $C_{smp}(n)$ and we have verified the result obtained by applying the Master Theorem.

COMPARISON: DIVIDE-BY-2 VS DECREASE-BY-FACTOR-2

In Figure 2 we can see a comparison between a Divide-by-2 strategy and a Decrease-by-Factor-2 strategy.

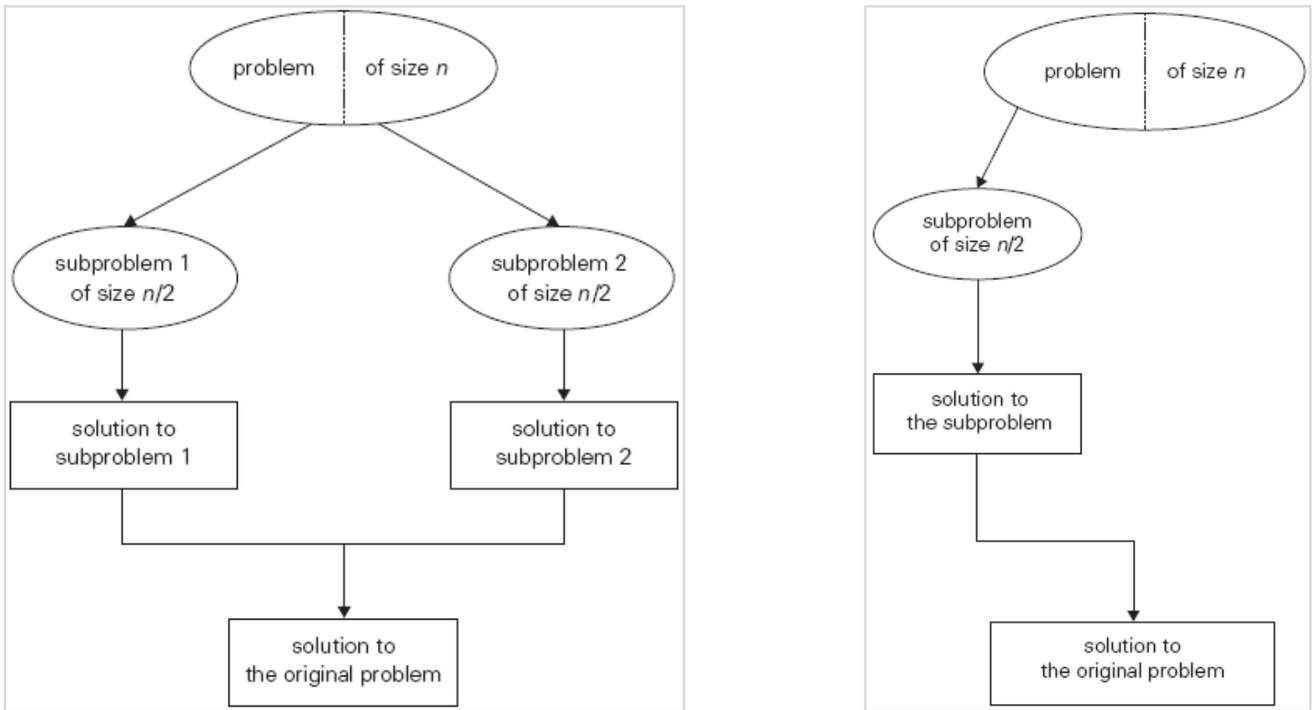

Figure 2. Comparison between divide-by-2 (left) and decrease-by-factor-2 (right).

COMPARISON: DIVIDE-AND-CONQUER VS DECREASE-BY-CONSTANT-FACTOR

A special case in a Divide-and-Conquer strategy is when a=1 (i.e., only 1 subproblem).

In this situation, the Divide-and-Conquer is equal to the Decrease-by-Constant-Factor.

In general, it is suggested to consider Decrease-by-Constant-Factor and Divide-and-Conquer as different algorithm designs (because in most cases the key ideas and the analysis tools used are different).

Note: Decrease-by-Constant-Factor algorithms as Binary Search can be considerate as degenerate cases of Divide-and-Conquer strategies, where just 1 of the subproblems needs to be solved.

MERGESORT: ALGORITHM

Problem: Sort an array of n orderable items, rearranging them in non-decreasing order.

Mergesort (Divide-by-2)
This algorithm sorts an input array using this strategy:

Step 1: Divide the input array (or subarray) into 2 halves.

Step 2: Sort each half of array/subarray recursively.

Step3: Merge the 2 smaller sorted halves into a single sorted array.
This merging is done by: (1) doing 2 parallel scans of the sorted halves, (2) comparing the current items in each scan, (3) inserting the smaller item to the final sorted array, and then (4) advancing only the scan used. Once a scan is completed, the other scan turns into a simple append of data in final array.

MERGESORT: ALGORITHM 2

Example: Figure 3 shows the operations of Mergesort in sorting an input unsorted array by recursively splitting it in halves, sorting the halves, and merging the results.

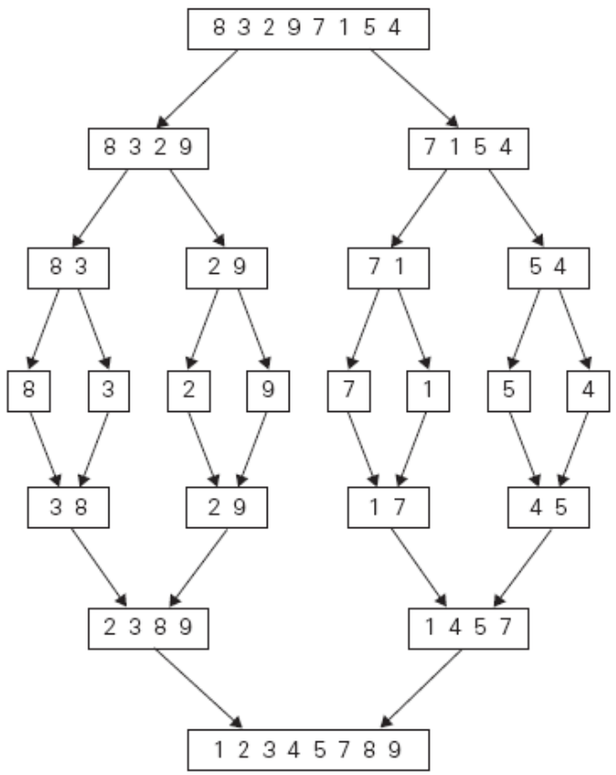

Figure 3. Recursive operations of Mergesort.

MERGESORT: IMPLEMENTATION

This is a Java implementation of Mergesort by Divide-by-2 (recursive, top-down).

```
public static void MergeSortedArrays( int[] B, int[] C, int[] A ) {
   int i = 0; int j = 0; int k = 0; // Init temp variables.
   // Scanning sorted arrays B and C, while inserting in A.
   while( ( i < B.length ) && ( j < C.length ) ) {
      if( B[i] <= C[j] ) { A[k] = B[i]; i++; }
      else { A[k] = C[j]; j++; }
      k++; }
   // One scan has terminated, transfer remaining sorted data in A.
   if( i == B.length ) { System.arraycopy( C, j, A, k, C.length - j ); }
   else { System.arraycopy( B, i, A, k, B.length - i ); }
}
public static void Mergesort( int[] A ) {
   if( A.length > 1 ) { // Check if sorting is really necessary.
      int h = (int) Math.floor( A.length / 2 ); // Determine halves size.
      // Init half 1 and 2.
      int B[] = new int[h]; System.arraycopy(A, 0, B, 0, h);
      int C[] = new int[A.length - h]; System.arraycopy(A, h, C, 0, A.length - h);
      // Sort (recursively) halves 1 and 2.
      Mergesort(B); Mergesort(C);
      // Merge sorted halves (arrays B and C) into final sorted array (A).
      MergeSortedArrays( B, C, A ); }
}
```

MERGESORT: ANALYSIS

The input size is the item-size of the input array (n).

The basic operation is:

- **Stage 1 (check array size):** conditional command (if).
- **Stage 2 (subarray initialization):** memory allocation (array-copy in constant time).
- **Stage 3 (recursive sorting):** recursive calls.
- **Stage 4 (final merging):** comparison/array-copy.

The basic operation count C(n) depends on input content: because the final merging requires a variable number of comparisons and the array-copy is usually executed in constant time. So, we have to study best-case and worst-case.

MERGESORT: ANALYSIS 2

- The **best-case** inputs include all arrays already sorted. So, in these scenarios, the final merging performs n/2 comparisons and 1 array-copy.

$$\mathrm{C}_{\mathrm{best}}(n) = \begin{cases} 1, & \text{if } n = 1. \\ 2 + \mathrm{C}_{\mathrm{best}}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathrm{C}_{\mathrm{best}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \frac{n}{2} + 1, & \text{if } n > 1. \end{cases}$$

- The **worst-case** inputs include arrays unsorted for which the final merging requires the maximum numbers of comparisons (n-1) and no array-copy.

$$\mathrm{C}_{\mathrm{worst}}(n) = \begin{cases} 1, & \text{if } n = 1. \\ 2 + \mathrm{C}_{\mathrm{worst}}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathrm{C}_{\mathrm{worst}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + (\mathrm{n} - 1), & \text{if } n > 1. \end{cases}$$

MERGESORT: ANALYSIS 3

To simplify our analysis, initially we consider only input size (n) values allowing perfect splits, that is: we will consider only n values that are a power of 2 (i.e., $n=2^k$). So, the basic operation count C(n) can be simplified in both the best-case ($C_{best\text{-}smp}(n)$) and worst-case ($C_{worst\text{-}smp}(n)$).

$$\text{if } n = 2^k \text{ then: } \quad C_{best-smp}(n) = \begin{cases} 1, & \text{if } n = 1. \\ 2\, C_{best-smp}\left(\frac{n}{2}\right) + \left(\frac{n}{2} + 3\right), & \text{if } n > 1. \end{cases}$$

$$\text{if } n = 2^k \text{ then: } \quad C_{worst-smp}(n) = \begin{cases} 1, & \text{if } n = 1. \\ 2\, C_{worst-smp}\left(\frac{n}{2}\right) + (n+1), & \text{if } n > 1. \end{cases}$$

Note: In both best-case and worst-case, the recursive definitions of the basic operation count are very similar. They only differ in their non-recursive part, but both of these terms are in the same efficiency class ($\Theta(n)$). So, we should expect the same results for best-case and worst-case if we apply the Master Theorem.

MERGESORT: ANALYSIS 4

Now, we can apply the Master Theorem to the worst-case basic operation count $C_{worst\text{-}smp}(n)$:

$$\text{if } n = 2^k \text{ then: } \quad C_{worst-smp}(n) = \begin{cases} 1, & \text{if } n = 1. \\ 2\, C_{worst-smp}\left(\frac{n}{2}\right) + (n+1), & \text{if } n > 1. \end{cases}$$

In this recurrence relation (considering the Master Theorem) we have:

- **d=1:** because f(n) = n+1, therefore f(n) is in $\Theta(n) = \Theta(n^1)$.
- **a=2, and b=2:** because we split into 2 subproblems and both of them have to be solved.

So, we apply the Master Theorem using the case $a = b^d$ (i.e., $2 = 2^1$), obtaining this result:

$$C_{worst-smp}(n) \in \Theta(\, n^d \log n\,) = \Theta(\, n^1 \log n\,) = \Theta(n \log n)$$

MERGESORT: ANALYSIS 5

Analyzing the simplified counts (for $n=2^k$) by applying the Master Theorem we obtain:

- All results of the simplified counts (best-case, and worst-case) can be extended to the general case (any n) thanks to the Smoothness Rule (n log n is a smooth function).
- The count in both best-case and worst-case is in the same efficiency class $\Theta(n \log n)$.

Note: Mergesort is stable (Quicksort and Heapsort are more efficient but unstable).

Note: A disadvantage of Mergesort is the linear amount of extra storage required. The space requirement (in its memory-optimized version) is $\Theta(n)$, and Mergesort is not in-place.

Note: Mergesort can be implemented iteratively (bottom-up).

Note: The Multiway Mergesort variation splits the array to be sorted in more than 2 parts.

QUICKSORT: ALGORITHM

Problem: Sort an array of n orderable items, rearranging them in non-decreasing order.

Quicksort (Divide-by-2)

This sorting algorithm is based on the array partitioning (see Figure 4):

Step 1: Partition the input array, placing the pivot is in its final sorted position.

Step 2: Then, can continue sorting the 2 subarrays (left and right of pivot) recursively.

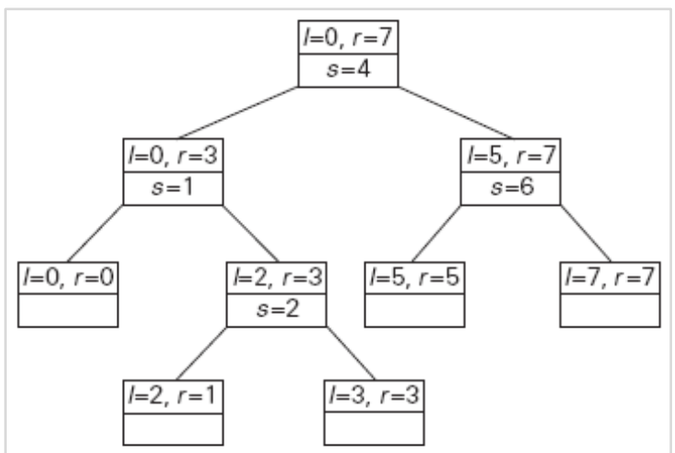

Figure 4. Example of recursive calls to sort an array using Quicksort (l/r are the subarray boundary-indices, s is the partitioning split-index).

QUICKSORT: IMPLEMENTATION

This is a Java implementation of Quicksort by Divide-by-2 (recursive, top-down).

```
public static void Quicksort( int[] A, int left, int right ) {
   // Check if any sorting is really necessary.
   if( left < right ) {
      // Array partitioning (any partitioning algorithm can be used here).
      int pivot = DecreaseAndConquer.Partitioning.HoarePartitioning(A,left,right);
      // Recursive application of Quicksort to left-part and right-part.
      Quicksort( A, left, pivot-1 );
      Quicksort( A, pivot+1, right );
   }
}
```

Note: This implementation can be improved in terms of efficiency by: (1) performing a better pivot selection (instead of using 1st item), (2) switching to Insertion Sort when subarray size is small, and (3) eliminating recursion by implementing Quicksort iteratively.

QUICKSORT: ANALYSIS

The input size is the item-size of the input array (n).

The basic operation is:

- **Stage 1 (check subarray size):** conditional command (if).
- **Stage 2 (subarray partitioning):** comparison (performed by partitioning algorithm).
- **Stage 3 (recursive sorting):** recursive calls.

The basic operation count C(n) depends on input content: because the splitting generated by the partitioning can have different shapes (skewed or balanced), affecting the numbers of recursive calls/partitionings executed.
So, we have to study best-case and worst-case.

QUICKSORT: ANALYSIS 2

- The **best-case** inputs include arrays in which all splits happen in the middle (balanced). So, in these scenarios, the partitioning requires n comparisons.

$$C_{best}(n) = \begin{cases} 1, & \text{if } n = 1. \\ 2\, C_{best}\left(\frac{n}{2}\right) + n, & \text{if } n > 1. \end{cases}$$

In this recurrence relation (considering the Master Theorem) we have:

- **d=1:** because f(n) = n, therefore f(n) is in $\Theta(n) = \Theta(n^1)$.
- **a=2, and b=2:** because we split into 2 subproblems and both have to be solved.

So, we apply the Master Theorem using the case $a = b^d$ (i.e., $2 = 2^1$), obtaining:

$$C_{best}(n) \in \Theta\left(n^d \log n\right) = \Theta\left(n^1 \log n\right) = \Theta(n \log n)$$

QUICKSORT: ANALYSIS 3

- The **worst-case** inputs include arrays in which all splits all the splits will be skewed to the extreme (a subarray will be empty and other will have a size of n-1).
  For example, this situation happens for arrays already sorted (increasingly).
  So, in these scenarios, the total number of comparisons performed by all the partitionings executed is the following.

$$C_{worst}(n) = n + (n-1) + \cdots + 3 + 2 = \sum_{i=2}^{n} i = (n+2)\frac{(n-2+1)}{2} \in \Theta(n^2)$$

QUICKSORT: ANALYSIS 4

Summarizing the results of our analysis of Quicksort we obtain:

- Time efficiency in best-case is $\Theta(n \log n)$ and in worst-case is $\Theta(n^2)$.
- Time efficiency in average-case is $\Theta(n \log n)$ (slightly better than Mergesort).

Note: Quicksort is considered the method of choice for sorting large files ($n \geq 10000$).

Note: Quicksort is in-place.

Note: Quicksort can be implemented iteratively (top-down).

COMPARISON: QUICKSORT VS MERGESORT

Note: In Mergesort, the items are divided according to their position.
In Quicksort, the items are divided according to their value (pivot value in partitioning).

Note: In Mergesort, the split into subproblems is immediate and most work happens to combine subproblem solutions.
In Quicksort, entire work happens in split stage (partitioning), with no work done to combine subproblems solutions.

Note: Quicksort is in-place, but Mergesort is not in-place (requires extra memory).

Note: Quicksort has better time efficiency than Mergesort (but equal efficiency class).

Note: Mergesort is more efficient to sort linked lists than Quicksort.

CLOSEST-PAIR 2D BY DIVIDE-BY-2: ALGORITHM

Problem: Find the closest pair of 2D points in a set of n 2D points (distinct points).

Closest-Pair 2D by Divide-by-2
We start by assuming that:

- The 2D points (stored in container P) are ordered by their x coordinate.
- The same points (stored also in container Q) are ordered by their y coordinate.

Then, using a Divide-and-Conquer strategy, we start solving the problem using both P and Q:

- **If $2 \leq n \leq 3$:** The problem is solved by using a Brute Force approach.
- **If $n > 3$:** We divide the points into 2 subsets P_{left} and P_{right} of $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ points, respectively. This split is done by drawing a vertical line ($x=m$) through the median m of x coordinates of the points.

CLOSEST-PAIR 2D BY DIVIDE-BY-2: ALGORITHM 2

Example: Figure 5 shows the Closest-Pair 2D problem by Divide-by-2.

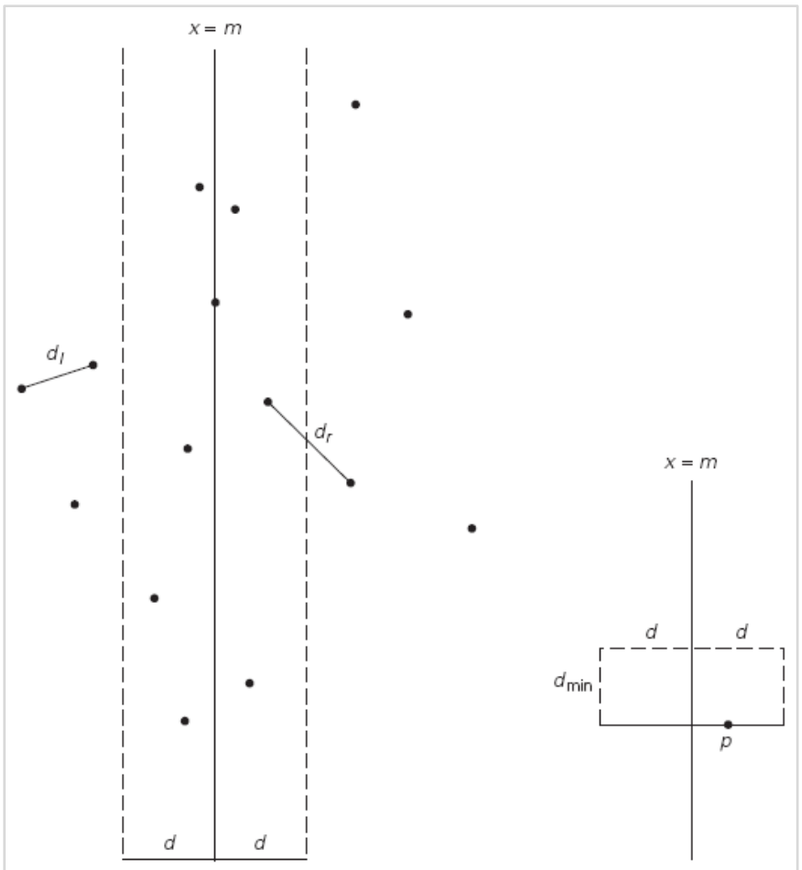

Figure 5. The Divide-by-2 strategy used to solve the Closest-Pair 2D.

CLOSEST-PAIR 2D BY DIVIDE-BY-2: ALGORITHM 3

Closest-Pair 2D by Divide-by-2 (Continued)

After drawing the median vertical line (x=m), we can solve the problem recursively for subsets P_{left} and P_{right} (Q_{left} and Q_{right} can be generated efficiently).

Then, let d_{left} and d_{right} be the distances of the closest-pairs solving P_{left} and P_{right}, respectively. Now consider $d=\min(d_{left}, d_{right})$: this value may not be the distance of the closest-pair in P.

For example: the closest-pair can lie on the opposite sides of the median vertical line.

So, to combine the subproblem solutions (closest-pairs solving P_{left} and P_{right}), we need to examine such pairs (lying on opposite sides of median vertical line).
To do this, we can limit our attention to the 2D points inside the symmetric vertical strip of width 2d around the median vertical line (x=m).
Any pair lying on opposite sides of the median vertical line, with the chance of being closer than d, must be in this symmetric vertical strip.

See Figure 5.

CLOSEST-PAIR 2D BY DIVIDE-BY-2: IMPLEMENTATION

This is the pseudocode of the Closest-Pair 2D by Divide-by-2 (recursive, top-down).

ALGORITHM *EfficientClosestPair*(P, Q)

```
//Solves the closest-pair problem by divide-and-conquer
//Input: An array P of n ≥ 2 points in the Cartesian plane sorted in
//       nondecreasing order of their x coordinates and an array Q of the
//       same points sorted in nondecreasing order of the y coordinates
//Output: Euclidean distance between the closest pair of points
if n ≤ 3
    return the minimal distance found by the brute-force algorithm
else
    copy the first ⌈n/2⌉ points of P to array P_l
    copy the same ⌈n/2⌉ points from Q to array Q_l
    copy the remaining ⌊n/2⌋ points of P to array P_r
    copy the same ⌊n/2⌋ points from Q to array Q_r
    d_l ← EfficientClosestPair(P_l, Q_l)
    d_r ← EfficientClosestPair(P_r, Q_r)
    d ← min{d_l, d_r}
    m ← P[⌈n/2⌉ − 1].x
    copy all the points of Q for which |x − m| < d into array S[0..num − 1]
    dminsq ← d^2
    for i ← 0 to num − 2 do
        k ← i + 1
        while k ≤ num − 1 and (S[k].y − S[i].y)^2 < dminsq
            dminsq ← min((S[k].x − S[i].x)^2 + (S[k].y − S[i].y)^2, dminsq)
            k ← k + 1
return sqrt(dminsq)
```

Figure 6. Pseudocode of the Closest-Pair 2D by Divide-by-2.

CLOSEST-PAIR 2D BY DIVIDE-BY-2: ANALYSIS

The input size is the number of 2D points in input (n).

The basic operation is:

- **Stage 1 (n≤3):** see count for Closest-Pair by Brute Force with size 3.
- **Stage 2 (subarray initialization):** assignment (array-copy in constant time).
- **Stage 3 (recursive calls):** recursive calls.
- **Stage 4 (final merging):** distance comparison in double loop (for+while).

The basic operation count C(n) depends on input content: because the final merging requires a variable amount of work depending on how many points are located in the vertical strip represented by container S.
So, we have to study best-case and worst-case.