Divide-and-Conquer

Practical Exercises

DA 2023 Instructors Team

Departamento de Engenharia Informática (DEI)
Faculdade de Engenharia da Universidade do Porto (FEUP)

Spring 2023

Exercise 1
Consider the same description for the **maximum sum subarray problem** as in the TP2 sheet.

a) Propose in pseudo-code a divide-and-conquer strategy for this problem.
b) Using the master theorem, indicate and justify the time complexity of the proposed algorithm.
c) Implement *maxSubsequenceDC* using the proposed algorithm.

```
int maxSubsequenceDC(int A[], unsigned int n , int &i, int &j)
```

Exercise 2
Consider the following pseudo-code:

```
mysteryFunc(A)
   if(A.size() ≤ 1)
     return A
   A1 ← A[0 : A.size()/2]
   A2 ← A[A.size()/2 : A.size()]
   B1 ← mysteryFunc(A1)
   B2 ← mysteryFunc(A2)
   i ← 0
   j ← 0
   k ← 0
   while(k < A.size())
      if (i ≥ B1.size())
         B[k++] ← B2[j++]
      else if (j ≥ B2.size())
         B[k++] ← B1[i++]
      else if (B1[i] < B2[j])
         B[k++] ← B1[i++]
      else
         B[k++] ← B2[j++]
return B
```

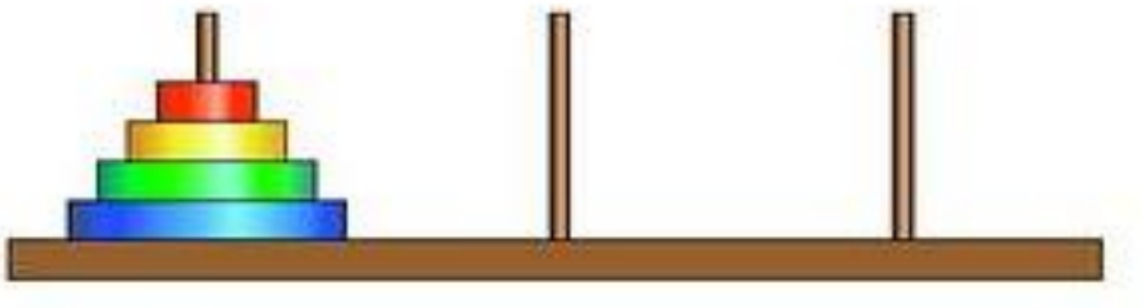
a) What does this algorithm in the pseudo-code do? Identify the name of the algorithm.
b) Explain why the algorithm follows a divide-and-conquer approach.
c) Using the master theorem, indicate and justify the temporal complexity of the algorithm, with respect to the array's size, n.
d) Implement the *mysteryFunc* in C++.

```
std::vector<int> mysteryFunc(const std::vector<int> &A)
```

Exercise 3

In the **Hanoi towers problem**, the goal is to move a stack of n disks of decreasing size from one peg to another, with the following constraints:

- Three pegs are available: A, B and C. The stack of n disks begins at one of the pegs.
- Only one disk can be moved at a time.
- At any time, the disk stack of any peg must be ordered in decreasing size order, with the largest disk at the bottom and the smallest one at the top.



a) Propose an algorithm in pseudo-code that solves this problem using a divide-and-conquer strategy. The solution must minimize the number of disk movements.
b) Using induction, prove that for n pegs, at most $(2^n - 1)$ moves are needed. Using this result indicate and justify the algorithm's time complexity, with respect to the number of disks, n.
c) Implement the *hanoiDC*, which implements the proposed algorithm.

```
std::string hanoiDC(unsigned int n, char src, char dest)
```

Input example: **n** = 4, **src** = 'A', **dest** = 'B'

Expected result:

"A→C,A→B,C→B,A→C,B→A,B→C,A→C,A→B,C→B,C→A,B→A,C→B,A→C,A→B,C→B"

Exercise 4

The goal of this exercise is to explore an efficient algorithm, called the Strassen algorithm, to multiply two matrices, with dimensions a × b and b × c.

a) Indicate and justify the time complexity of classic/naive matrix multiplication algorithm, commonly taught in algebra classes.

As an alternative to the classic algorithm, the Strassen algorithm proposes splitting the two matrices to be multiplied, X and Y, into four blocks and performing the operations detailed below, in order to produce the matrix Z. For now, to simplify, consider that X and Y are square matrices (i.e. same number of rows and columns) with a dimension that is a power of two (such as 1×1, 2×2, 4×4 …).

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \quad Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix} \quad Z = \begin{pmatrix} I & J \\ K & L \end{pmatrix}$$

$$M_1 = (A + D) \times (E + H)$$

$$M_2 = (C + D) \times E$$

$$M_3 = A \times (F - H)$$

$$M_4 = D \times (G - E)$$

$$M_5 = (A + B) \times H$$

$$M_6 = (C - A) \times (E + F)$$

$$M_7 = (B - D) \times (G + H)$$

$$I = M_1 + M_4 - M_5 + M_7$$

$$J = M_3 + M_5$$

$$K = M_2 + M_4$$

$$L = M_1 - M_2 + M_3 + M_6$$

The algorithm's base case corresponds to multiplying two 1×1 matrices, which yields a 1×1 matrix.

b) Explain why the Strassen algorithm follows a divide-and-conquer approach.
c) Using the master theorem, indicate and justify the temporal complexity of the Strassen algorithm.
d) Implement *strassen* which corresponds to the Strassen algorithm (for squared matrices with dimensions that are powers of two).

```
std::vector<std::vector<double>> strassen(const
    std::vector<std::vector<double>> &X, const
        std::vector<std::vector<double>> &Y)
```

e) Generalize *strassen* so that it works with non-square matrices, whose dimensions are not necessarily powers of two.

Suggestion: the Strassen algorithm itself does not need to be modified. Instead, pad the both input matrices with 0's so that they are both squared, have the same dimensions and have dimensions that are a power of 2. After performing the multiplication (you can rename the function developed in the previous question to *strassenAux* to execute the actual Strassen algorithm), cut the matrix so that it has the correct dimensions.

Exercise 5

Consider the *bisection* function below, which applies the bisection algorithm. This algorithm receives:

- two real numbers x1 and x2 (x1 <= x2) such that f(x1) and f(x2) have opposite signs;
- a unary function f, which represents a continuous function over the interval [x1, x2];
- a real number, err, that represents a maximum accepted error.

According to the intermediate value theorem (and its corollary, Bolzano's theorem), using these inputs, f has a root in [x1, x2], i.e. there is a value x in [x1, x2] such that f(x) = 0.

The bisection algorithm iteratively recomputes the midpoint between x1 and x2, xm = (x1 + x2)/2.0, and either continues searching for the root in [x1, xm] or [xm, x2] according to the sign of f(xm). The algorithm stops and returns xm when the following condition is met:

$$|x1 - x2| \leq err \wedge |f(x1) - f(x2)| \leq err$$

```
double bisection(double x1, double x2, double err, double(*f)(double))
```

Implement *bisection* using the previously described algorithm.

Note: The bisection method actually belongs to a simplified case of divide-and-conquer, which can be called, "reduce-and-conquer" where only one branch of the sub-problem solution tree is needed for the problem's solution. In fact, in each iteration, the algorithm only performs one recursive call, just like with performing binary search in a sorted array.

Exercise 6

The goal of the **closest pair problem** is to determine the two points that minimize the Euclidean distance between them, among a set of 2D points.

a) Implement the ***nearestPoints_DC*** function using the divide and conquer algorithm described in the appendix (except for the last part that requires a second list).

```
Result nearestPointsDC(std::vector<Point> &vp)
```

b) Implement the ***nearestPoints_DC_MT*** function, a multi-threaded version of the divide and conquer algorithm. Use the provided unit tests to compare the performances obtained for different sizes of input data and different numbers of threads.

```
Result nearestPointsDC_MT(std::vector<Point> &vp)
```

Exercise 7

Consider two n-bit integers x and y and assume for convenience that n is a power of 2 (the more general case is hardly any different) which they can be split into their left and right halves, which are n/2 bits long:

$$x = \boxed{x_L}\boxed{x_R} = 2^{n/2}x_L + x_R$$
$$y = \boxed{y_L}\boxed{y_R} = 2^{n/2}y_L + y_R.$$

For instance, if x = 10110110_2 then $x_L = 1011_2$, $x_R = 0110_2$, and x = $1011_2 \times 2^4 + 0110_2$. The product of x and y can then be rewritten as

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

and based on the observation that $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$, the overall operation (after some simplifications) only requires 3 multiplications rather than the original 4 multiplications.

With this "trick" due to Gauss, develop an efficient method to multiply the numbers x and y (which is rather immediate) and derive its asymptotic complexity by showing the associated divide-and-conquer recurrence and its solution using the Master Theorem.

Exercise 8
Suppose you are choosing between the following three algorithms:

- Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
- Algorithm B solves problems of size n by recursively solving two subproblems of size n-1 and then combining the solutions in constant time.
- Algorithm C solves problems of size n by dividing them into nine subproblems of size n/3, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each of these algorithms (in big-O notation), and which would you choose if your problem instances exhibited large values of n?

Appendix: Divide and Conquer Algorithm to compute the closest pair of points

This appendix was adapted from M.A. Weiss, "Data Structures and Algorithms Analysis in C++", 3rd edition – chapter 10, pages 430-435.

Suppose *P* is a list of points on a plane. If *p1=(x1,y1)* and *p2=(x2,y2)*, the Euclidean distance between *p1* and *p2* is given by:

$$[(x_1 - x_2)^2 + (y_1 - y_2)^2]^{\frac{1}{2}}$$

The objective is to find the two closest points. If there are two points with the same coordinates, those are the two closest, with distance 0.

If there are N points, then there are *N(N-1)/2* pairs of distances. One could look through all of them with a very simple exhaustive search (brute force) algorithm. However, that algorithm would have complexity $O(N^2)$. With a divide and conquer algorithm like the one described below, one can guarantee *O(NlogN)* complexity.

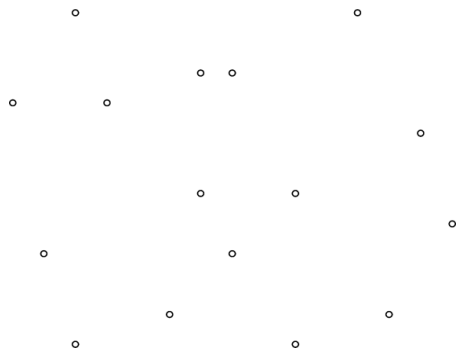


Figure 1 – A small set *P* of points

Figure 1 shows a small set of *P* points. If the points were sorted by their value in the abscissa (*x* axis), one could draw an imaginary vertical line which divides the set in two halves P_L and P_R. Given this division, either both points of the closest pair are in P_L, both are in P_R or one is in P_L and one is in P_R. We can call the distances between them d_L, d_R and d_C, as shown in figure 2.

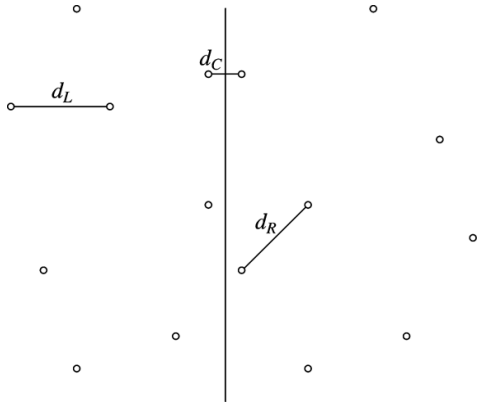


Figure 2 – Set *P* divided into P_L and P_R, with the minimum distances shown.

Computing d_L and d_R can be done recursively. O the problem is then to compute d_C. In order to guarantee a *O(NlogN)* complexity algorithm (needed to sort the values), it must be possible to compute d_C in *O(N)*.

Consider $\delta = (d_L, d_R)$. One only has to compute d_C if it is smaller than δ. With that in mind, it can be said that the two points which define d_C should be less than δ distance from the dividing line. We will name this area the **strip**.

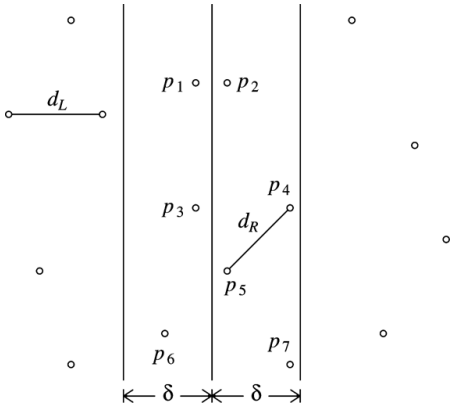


Figure 3 – Two bands containing all of the points considered for the strip d_C

There are two strategies to compute d_C. For a very large set of uniformly distributed points, the number of points expected to be contained in the strip is very small – on average there will be $O(\sqrt{N})$ points. In that case, the brute force algorithm can be used in this strip in time *O(N)*. The pseudo-code for this strategy is shown in figure 4.

```
// Points are all in the strip

for( i = 0; i < numPointsInStrip; i++ )
    for( j = i + 1; j < numPointsInStrip; j++ )
        if( dist(p_i, p_j) < δ )
            δ = dist(p_i, p_j);
```

Figure 4 – Brute-Force Algorithm to compute *min(δ, dC)*

In the worst-case scenario, every point can be in the strip. In this case, the brute force strategy does not run in linear time. It is necessary to look closely at the problem in order to improve the algorithm: the *y* coordinates of the two points which define d_C should differ, at most, δ; otherwise, $d_C > \delta$. Suppose the points in the strip are sorted by their *y* coordinate. If the y coordinates of points p_i and p_j differ more than δ, the algorithm skips to point p_{i+1}. This simple modification is implemented in the algorithm shown in figure 5.

```
// Points are all in the strip and sorted by y-coordinate

for( i = 0; i < numPointsInStrip; i++ )
    for( j = i + 1; j < numPointsInStrip; j++ )
        if( p_i and p_j's y-coordinates differ by more than δ )
            break;          // Go to next p_i.
        else
        if( dist(p_i, p_j) < δ )
            δ = dist(p_i, p_j);
```

Figure 5 – Improved computation of *min(δ, d_C)*

This simple additional test has a very significant effect on the algorithm's behavior, because for each point p_i, very few points p_j are examined (if their coordinates differ by more than δ, the internal *for* loop is terminated). Figure 6 shows, for example, that for point *p3*, only points *p4* and *p5* are less than δ distance away on the vertical axis.

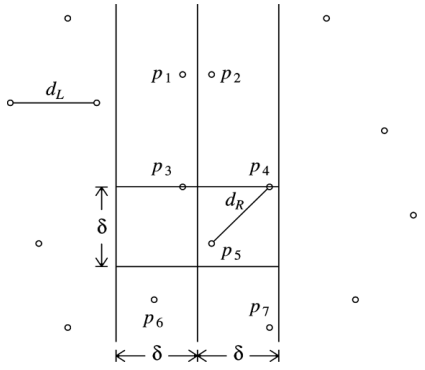


Figure 6 – Only points *p4* e *p5* are considered in the second *for* loop

In the worst-case scenario, for any point p_i at most seven points p_j will be considered. The reason for this is simple: these points must be contained in a δxδ square on the left half of the strip or a δxδ square

on the right half of the strip. On the other hand, all points in each δxδ square are at least δ distance from each other. Worst case scenario, each square contains four points, one in each corner. One of those points is p_i, leaving, therefore, at most seven points to be considered. This case is illustrated in figure 7. Even in points p_{L2} and p_{R1} have the same coordinates they can be different points. In this analysis, the important thing to notice is that the number of points in the λ by 2λ rectangle is $O(1)$.

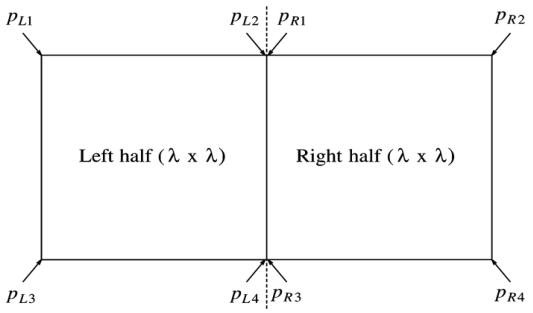


Figure 7 - There at most eight points in the rectangle, each sharing two coordinates with other points.

Given that there are at most seven points to be considered for each p_i, the time needed to compute a d_C better than δ is O(N). It seems, then, that a O(NlogN) solution for the closest pair problem has been found, based on the recursive call of the left and right halves plus the linear time to combine the results.

However, this solution is not effectively O(NlogN). It has been assumed that the list of points is sorted. If this operation is done in each recursive call, then there is work of complexity O(NlogN) to consider, which results in an overall complexity of O(Nlog^{2}N). This is not too bad when compared to brute force's O(N^2). It is, however, relatively easy to reduce the complexity of each recursive call to O(N), therefore guaranteeing overall complexity O(NlogN).

The idea is to maintain two lists: one contains the points sorted by the x axis, while the other contains the points sorted by the y axis. This implies a first step where the points are sorted, with complexity O(NlogN). Let these lists be names P and Q, respectively. P_L and Q_L are the lists passed into the left half recursive call, while P_R e Q_R are the lists passed into the right half recursive call. List P is easily split in half. Once the dividing line is known, Q is traversed sequentially, placing each element in Q_L or Q_R as appropriate. It is easy to verify that both Q_L and Q_R are automatically sorted by y as they are filled in. Once the recursive call returns, all points in Q whose x coordinate does not belong within the strip are removed. That way, Q contains all of the points which are within the strip, already sorted by y.

This strategy guarantees that the overall algorithm has complexity O(NlogN), because the only extra processing which is done is done with complexity O(N).