Divide-and-Conquer

Solutions to the Practical Exercises

DA 2023 Instructors Team

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

Spring 2023

Exercise 1

a) The main idea of the algorithm is to split the array in two halves and to consider three scenarios for the position of the maximum sum subarray and select the one that produces the largest subarray. The maximum subarray either:

- is completely in the first half,
- is completely in the second half,
- is located in both halves.

The maximum subarray for the first two cases is computed using a recursive call to this algorithm on each half-array. This corresponds to the divide/split step of this divide-and-conquer algorithmic solution. The subarray in the third case is computed based on the observation that it has to contain both the last element of the first half and the first element of the second half. Thus, all possible subarrays that contain this element are considered. All the possible arrays are obtained by adding consecutive element moving from right to left on the first half and from left to right on the second half.

Note: if the array has an odd length, then either of the halves can have the surplus element: it does not change any part of the reasoning.

Pseudo-code for finding the maximum sum subarray:

Input and auxiliary functions:

- A: array of numbers (which can be positive, zero or negative). To simplify it is assumed to contain at least one element. Empty arrays can be easily handled in the first call to the function.

Output: max sum subarray and its sum.

```
compute_max_sum_subarray(A)
    // Base case
    if(A.size() == 1)
      return A, A[0]
    // Recursive case
    // Split step
    A1 ← A[0 : A.size()/2]
    A2 ← A[A.size()/2 : A.size()]
    candidate1, sum1 ← compute_max_sum_subarray(A1)
    candidate2, sum2 ← compute_max_sum_subarray(A2)

    // Merge step
    curSum ← A1.[A1.size() - 1]
    bestSum3Left ← curSum
    candidate3Left ← [curSum]
    for (i = A1.size() - 1; i >= 0; i--)
        curSum ← curSum + A1[i]
        if (curSum  < bestSum3Left)
            bestSum3Left ← curSum
            candidate3Left ←  [A1[i]] ++ candidate3Left // concatenate

    curSum ← A2.[0]
    bestSum3Right ← curSum
    candidate3Right ← [curSum]
    for (i = 0 - 1; i <= A2.size() - 1; i++)
        curSum ← curSum + A2[i]
        if (curSum  < bestSum3Right)
            bestSum3Right ← curSum
            candidate3Right ← candidate3Right ++  [A2[i]] // concatenate

    candidate3 ← candidate3Left ++ candidate3Right
    sum3 ← bestSum3Left + bestSum3Right

    if (sum1 > sum2 && sum1 > sum3)
        return sum1, candidate1
    else if (sum2 > sum3)
        return sum2, candidate2
    else
        return sum3, candidate3
```

b) In the base case (n = 1), the algorithm runs in constant time (*i.e.* O(1) time) since it simply returns the array's only element as the maximum sum.

In the recursive step (n > 1), the algorithm recursively calls itself for the two halves of the array. Thus, the recurrence formula for the execution time with respect to n, T(n), is:

$$T(n) = \begin{cases} c & n = 1 \\ 2T(\frac{n}{2}) + f(n) & n > 1 \end{cases}$$

where:

- c is a constant representing the upper bound of the computation time of the base case.
- f(n) is the driving function, *i.e.* the function of the computation time of the conquer/merge step of the algorithm, with respect to n.

The conquer step consists in finding the maximum sum subarray that crosses the "midpoint" (*i.e.* the point between elements where the array was split into half for the recursive calls). This subarray is obtained in O(n) time since it requires visiting each element of the array once. The comparison of the sum of the three subarrays can be done on O(1) time. In addition, splitting the array in two halves takes O(n) time. Thus, f(n) ∈ Θ(n) as the conquer step is dominated by O(n) time algorithms. Since a = 2 and f(b) = f(2) = 2, we have a = f(b), and given the Master Theorem: T(n) ∈ Θ($n^{\log2(2)}$·log(n), thus T(n) runs in approximately Θ(n·log(n)) time.

c) See source code.

Exercise 2

a) The algorithm sorts an array of comparable values (such as numbers). It is merge sort!

b) The algorithm uses a divide-and-conquer strategy as it contains its two fundamental steps:
- split/divide: the array to be ordered is split into two halves which are then ordered in the recursive call, thus dividing the original problem into two independent sub-problems.
- merge/conquer: the elements of the two halves are processed in their sorted order to produce the final sorted array, thus the solutions of the sub-problems are used to build the solution of the original problem.

c) In the base case (n = 1), the algorithm runs in constant time (i.e. O(1) time) since it simply returns the input array.

In the recursive step (n > 1), the algorithm recursively calls itself for the two halves of the array. Thus, the recurrence formula for the execution time with respect to n, T(n), is:

$$T(n) = \begin{cases} c & n = 1 \\ 2T(\frac{n}{2}) + f(n) & n > 1 \end{cases}$$

where:
- c is a constant representing the upper bound of the computation time of the base case.
- f(n) is the driving function.

The conquer step consists in combining the two sorted "half-arrays" to form the sorted "combined array". This is performed in O(n) time since the half-arrays are processed in exactly n operations of O(1):
- There are exactly n operations since in each iteration exactly one element is added to the sorted array, and this array has n elements.
- Each operation takes O(1) as they simply consist of a limited amount of number comparisons and updating a value of an array.

Thus, f(n) ∈ Θ(n).

Since a = 2 and f(b) = f(2) = 2, we have a = f(b), thus, given the Master Theorem: $T(n) \in \Theta(n^{\log2(2)} \cdot \log(n))$, thus T(n) runs in approximately $\Theta(n \cdot \log(n))$ time.

d) See source code.

Exercise 3

a) The solution below assumes that disks are to be moved from peg A to peg B. Peg C is auxiliary.

<u>Pseudo-code for solving the Hanoi towers puzzle</u>:

Input:
- n: number of disks
- src: initial peg
- dest: destination peg
- aux: auxiliary peg

Output: ordered list of moves

```
hanoi(n,src,dest,aux)
        if (n = 1)
          return “src->dest”
else
        hanoi(n-1, src, aux) ++ “,” ++ hanoi(1, src, dest) ++ “,” ++ hanoi(n - 1, aux, dest)
```

b) To prove this property, induction shall be performed over the natural number n, which denotes the number of disks.

<u>Base case:</u> for n = 1, all that is needed is move the disk from the origin to the destination peg in one move and, for n = 1, $2^n - 1 = 2 - 1 = 1$.

<u>Inductive step:</u> let n > 1. Let us assume that solving the puzzle with n disks requires $2^{n-1} - 1$ moves.

For n disks, the optimal approach for the puzzle is:
- move the top (n-1) disks from source peg (A) to the auxiliary peg (B), requiring $2^{n-1} - 1$ moves, according the inductive hypothesis, since setting the destination peg to be A, B or C does not make a difference and third peg can be used as the auxiliary peg since either it is empty.
- move the largest disk from the source peg (A) to the destination peg (C), which requires 1 move.
- move the (n-1) from peg B to peg C, which also requires $2^{n-1} - 1$ moves, according the inductive hypothesis.

Thus, the minimum required number of moves is: $2^{n-1} - 1 + 1 + 2^{n-1} - 1 = 2 \times 2^{n-1} - 1 = 2^n - 1$.

Conclusion: since the property is true for n = 1 and, for any n > 1, if the property holds for n-1, then it holds for n, then the property is true for any n > 0. Therefore, solving the Hanoi towers problem with n disks requires $2^n - 1$ moves.

Since the base case of the pseudo-code is called once per move and there are $2^n - 1$ moves, assuming that concatenating lists of moves takes O(1) time, then the temporal complexity of the algorithm is $O(2^n - 1) = O(2^n)$

Note: in this problem, the master theorem is not used to prove the problem's temporal complexity given that the recurrence formula for T(n) for the recursive case does not have the form: $T(n) = a \times T(n/b) + f(n)$, instead having the form: $T(n) = 2 \times (n-1) + f(n)$.

c) See source code.

Exercise 4

a) The time complexity of the naive matrix multiplication algorithm is $O(n^3)$, with respect to n, which is the largest dimension among both matrices' dimensions (*i.e.* n = max(a,b,c)).

Justification: To simplify, consider the case of multiplying two n x n matrices. The resulting matrix has $n \times n$ dimensions, thus the value of n^2 cells needs to be computed. The value of each cell corresponds to the scalar product of two vectors of length n, so the computations takes O(n) time. Thus, the overall complexity is: $n^2 \times O(n) = O(n^3)$.

If the matrices have different dimensions, then if n is considered to be their upper bound, the number of computations will also be bounded by the $O(n^3)$ temporal complexity, given that, in reality, fewer operations will be performed.

This complexity and justification assumes that any cell of a matrix can be consulted in O(1), which is definitely possible if the matrix is implemented as a 2D array or as a vector of vectors, for example.

b) The Strassen algorithm follows a divide-and-conquer strategy since it breaks down the problem of multiplying two matrices into various independent sub-problems of multiplying matrices of smaller size.

To be more precise, the four blocks of the result matrix, Z, are computed by adding a set of matrices M_i, which are product of different sums and subtractions of blocks from X and Y. Z's blocks are not interdependent, neither are the M_i matrices, thus all the sub-problems are independent and can be performed in parallel.

c) In the base case (n = 1), the algorithm runs in constant time (*i.e.* O(1) time) since all that is needed is to multiply two 1×1 matrices.

In the recursive step (n > 1), the algorithm converts the multiplication of two n x n matrices into 7 multiplications of matrices with a half the dimensions. Thus, the recurrence formula for the execution time with respect to n, T(n), is:

$$T(n) = \begin{cases} c & n = 1 \\ 7T(\frac{n}{2}) + f(n) & n > 1 \end{cases}$$

where:
- c is a constant representing the upper bound of the computation time of the base case.
- f(n) is the driving function.

The conquer step consists in adding and subtracting a set of matrices to then build the Z matrix:

- Adding two n x n matrices takes $O(n^2)$ time since there n^2 cells to be computed, with each one taking O(1) time since only two corresponding values in the operand matrices needs to be consulted, followed by adding them.
- Subtracting two matrices, A - B, can be implemented by inverting the sign of B, and then adding A and (-B). Inverting the sign of B takes $O(n^2)$ as all the n^2 cells are consulted once and multiplied by -1. Thus, subtracting takes $O(n^2)$ time as it combines two $O(n^2)$ time algorithms.
- The Z matrix can also be built in $O(n^2)$ time by iterating Z's n^2 cells and filling them in with the corresponding value from one of the blocks, which can be consulted in O(1) time.

Therefore, $f(n) \in \Theta(n^2)$, since it is a combination of a limited amount of $O(n^2)$ time operations.

Since a = 7 and $f(b) = f(2) = 2^2 = 4$, we have a > f(b), thus, given the Master Theorem $T(n) \in \Theta(n^{\log2(7)})$, thus T(n) runs in approximately $\Theta(n^{2.807})$ time.

It is interesting to note that, if 8 sub-problems were created rather than 7, the algorithm would run in $\Theta(n^{\log2(8)}) = \Theta(n^3)$ time, the same complexity as the classic/naive matrix multiplication algorithm.

d) See source code.
e) See source code.

Exercise 5
See source code.

Exercise 6
a) See source code.
b) See source code.

Exercise 7

$$T(n) = 3T(n/2) + O(n).$$

The point is that now the constant factor improvement, from 4 to 3, occurs *at every level of the recursion*, and this compounding effect leads to a dramatically lower time bound of $O(n^{1.59})$ rather than the $O(n^2)$ of the basic or standard multiplication algorithm.

Exercise 8

In this case we just develop the recurrence, ignoring the threshold value for the base case and compare the asymptotic complexity of the three algorithms. As such, and whenever possible using the Master theorem we can derive the following asymptotic complexities for each of these three algorithms:

A. $T(n) = 5\,T(n/2) + \alpha\, n^1$. This recurrence corresponds to the third case of the Master Theorem and leads to a solution of the form $T(n) = \Theta(n^{log_2 5}) = \Theta(n^{2.32})$

B. $T(n) = 2\,T(n-1) + \alpha\, n^0$. This recurrence is the same as for the Towers of Hani problem and thus exhibits a solution of the form $T(n) = \Theta(2^n)$

C. $T(n) = 9\,T(n/3) + \alpha\, n^2$. This recurrence corresponds to the second case of the Master Theorem and leads to a solution of the form $T(n) = \Theta(n^2 \log n)$.

From these derivations, and under the assumption that the size of the specific circumstances at hand would be large, we would pick algorithm A. Why? We need to compare $n^{0.32}$ against $\log n$ which is tricky as $\lim_{n\to\infty} \frac{n^{0.32}}{\log n}$ is indeterminate. To solve this, and determine which function is eventually bigger we resolve this indeterminacy using L'Hôpital's rule and computing the same limit for the derivatives and checking if this limit is bigger than 1.0. If it is, it means that in the original fraction, the denominator grows faster than the denominator and the limit in this case would be infinite. In this case, $\lim_{n\to\infty} \frac{\propto n^{-0.68}}{\beta\, 1/n} = \lim_{n\to\infty} \frac{\propto n^{0.32}}{\beta} = \infty$ which is at a given point greater than 1, irrespective of the values of the constants. As such, $n^{2.32}$ will eventually be bigger than $n^2 \log n$.