Disjoint sets and connected components; the Traveling Salesman Problem

Algoritmo y estructura de datos avanzadas, 2022-23

Lab 3

GROUP 02

Daniel Varela Sánchez

Guillermo Martín-Coello Juárez

I-C Questions on QuickSelect and QuickSort

1. **Argue that MergeSort sorts a table of 5 elements with at most 8 key comparisons**

   The MergeSort algorithm first divides elements until they are completely separated, to then add them together in the same order they were divided. For a 5 element array a merge sort will need 4 splits, using the following schema:

   ```
   5 -> 3, 2 -> 1,2,2 -> 1,1,1,2 -> 1,1,1,1,1
   ```

   Where each number is the amount of elements in each array.
   To merge the elements back together, the same schema will be followed, in reverse

   ```
   1,1,1,1,1 -> 1,1,1,2 -> 1,2,2 -> 3, 2 -> 5
   ```

   This could vary slightly depending on the implementation.
   As we can observe, this will produce two one-to-one comparisons, one two-to-one comparison and one three-to-two comparison.
   Merge sort comparisons between arrays compare the two first elements of each array and pull them into the new array. This is done for each element until one of the arrays is empty, and will cost at most the size of both arrays -1.

   ```
   Wcost = n1 + n2 - 1
   ```

   For each array comparison.
   The results will be for two one-to-one comparisons 2 * (1+1-1)
   one two-to-one comparison 1 * (2+1-1)
   and one three-to-two comparison 1 * (3+2-1)
   Adding a total of

   ```
   (2 * (1+1-1)) + (1 * (2+1-1)) + (1 * (3+2-1)) = 8 Comparisons
   ```

2. **Actually, in qsel_5 we are only looking for the median of 5 elements, not necessarily an ordering. Could we reduce the number of comparisons necessary?**

   As it was shown previously, the QuickSort algorithm uses 8 comparisons because it needs to order each of the elements of the array. On the other hand, to find the median we only need to compare enough elements to know which one is in the middle.
   For an array [a,b,c,d,e] we can compare #1 (a,b), #2 (c,d) and #3 (min(a,b),min(c,d)) (where the minimums are obtained using the previous comparisons) to find the smallest number of those four elements: min(a,b,c,d). This number can be discarded as it is clearly not the median, resulting in the following information obtained:

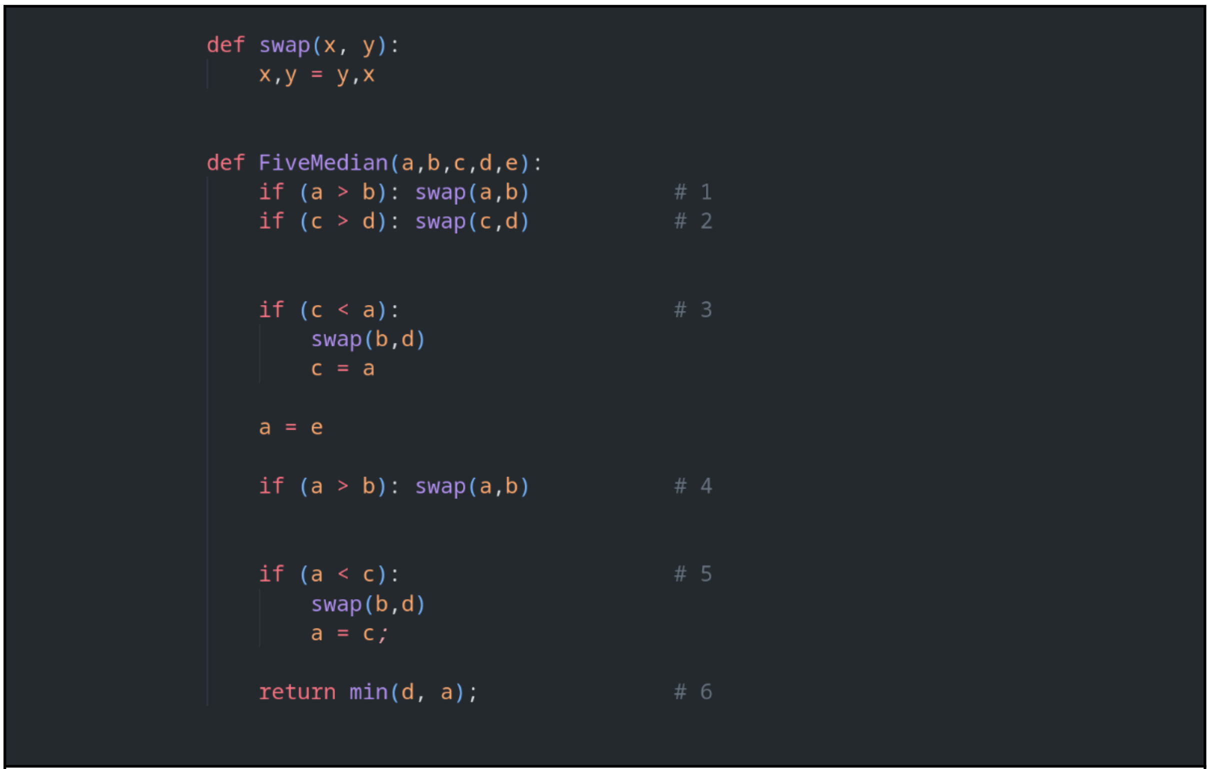
   ```
   min(a,b), max(a,b)
   min(c,d), max(c,d)
   min(a,b,c,d) (DISCARDED)
   ```

Group 02
Daniel Varela Sánchez
Guillermo Martín-Coello Juárez

After these operations, one of the elements of the first two comparisons will be discarded (a,b,c or d). The alone element left (for this example we will show as if the discarded element min(a,b,c,d) was “a”, so our alone number will be “b”, but it could be any other depending on the case) can be compared with the remaining number to form a new couple: #5 (b,e). Then we can compare this new couple to the left one min((b,e),(c,d)) to obtain a new minimum number to be discarded. Our information will be:

min(b,e), max(e,b)
min(c,d), max(c,d)
min(b,c,d,e) (DISCARDED)

We now have discarded 2 the two smallest numbers of the array, so the next smallest number will be the middle one (the median). The only comparison left to do is compare the remaining element of the couple with the recently discarded number (in this example we will discard “b” and use “c”) and compare it to the other couple #6 min(c,d,e), resulting in the Median with only 6 operations.
To better understand this explanation we made the following code:

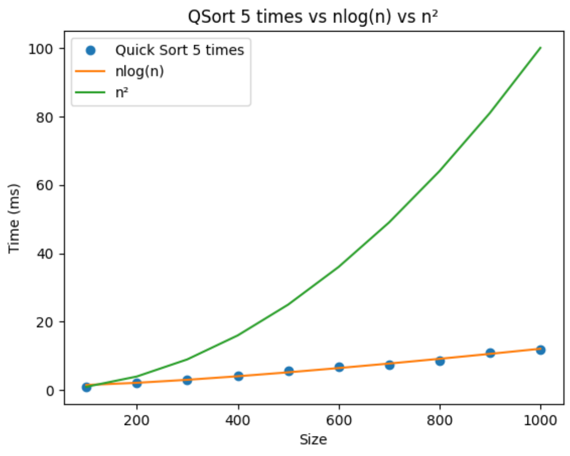


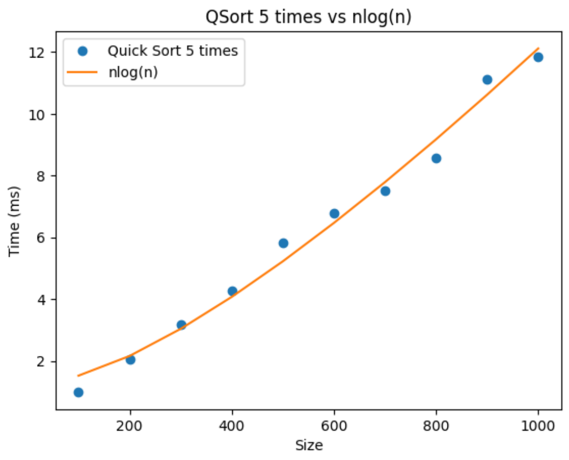
```
def swap(x, y):
    x,y = y,x


def FiveMedian(a,b,c,d,e):
    if (a > b): swap(a,b)           # 1
    if (c > d): swap(c,d)           # 2


    if (c < a):                     # 3
        swap(b,d)
        c = a

    a = e

    if (a > b): swap(a,b)           # 4


    if (a < c):                     # 5
        swap(b,d)
        a = c;

    return min(d, a);               # 6
```

Image I: Code used to calculate the median in 6 comparisons

3. **What kind of growth behavior vis-a-vis execution time can we expect in the worst case for the function qsort_5? Try justifying your answer experimentally and analytically.**
The worst case of QuickSort comes from the bad behavior over ordered tables of the pivot used in split. To improve pivot selection, we use the pivot 5 method. With this pivot, an O(N lg N) worst case for QuickSort can be achieved.

The reason is that a first element pivot will be in the average near the middle of the table, but it may also be at the beginning or the end, causing $W_{QS} = n^2$. With pivot5, the pivot will be in the median, and the cost in pair with pair with the average, $A_{QS} = n*\log(n)$, $W_{QS5} = n*\log(n)$.
To prove this hypothesis, a graph of the worst case (using an ordered list) execution of QSort5 was done compared to the QuickSort worst case (n^2) and to the expected worst case for QSort5.





As we can observe, the algorithm worst case cost clearly follows the expected result.

II-C Questions about dynamic programming

1. **The problem of finding the maximum non-consecutive common substring is often confused with that of finding the maximal consecutive one. See, as an example, the entry Longest common substring problem in Wikipedia. Describe a dynamic programming algorithm that finds the longest consecutive common substring between two strings S and T and apply it by hand to find the longest common substring between bajamas and bananas.**

Finding the maximum consecutive substring works by only using a variable to save the length of the longest substring yet to be discovered, and only changing to a new one if this variable is surpassed.

```
function LCSubstr(S[1..r], T[1..n])
    L := array(1..r, 1..n)
    z := 0
    ret := {}

    for i := 1..r
        for j := 1..n
            if S[i] = T[j]
                if i = 1 or j = 1
                    L[i, j] := 1
                else
                    L[i, j] := L[i − 1, j − 1] + 1
                if L[i, j] > z
                    z := L[i, j]
                    ret := {S[i − z + 1..i]}
                else if L[i, j] = z
                    ret := ret ∪ {S[i − z + 1..i]}
            else
                L[i, j] := 0
    return ret
```

Maximum Consecutive Substring algorithm. Source: "Wikipedia"

Group 02
Daniel Varela Sánchez
Guillermo Martín-Coello Juárez

A manual understanding of the algorithm shown will be the following one:

	0	b	a	n	a	n	a	s
0	1	0	0	0	0	0	0	0
b	0	2	0	0	0	0	0	0
a	0	0	3	0	1	0	1	0
j	0	0	0	0	0	0	0	0
a	0	0	1	0	1	0	1	0
m	0	0	0	0	0	0	0	0
a	0	0	1	0	1	0	1	0
s	0	0	0	0	0	0	0	2

Evolution of Z and Ret:

Z = 1, Ret = 0

Z = 2, Ret = 0b

Z = 3, Ret = 0ba

Z = 3, Ret = 0ba

Z = 3, Ret = 0ba

Z = 3, Ret = 0ba

Z = 3, Ret = 0ba

Z = 3, Ret = 0ba

Ret = ba

2. We know that finding the smallest number of multiplications necessary to multiply a list of n matrices is $O(n^3)$, but now we want to estimate that number, $v(n)$ more precisely. Estimate such a number using an expression $v(n) = f(n) + O(g(n))$ with f and g such that $|v(n) - f(n)| = O(g(n))$.

Since we have to compute $O(N^2)$ values m_{ij} for $2 \leq j \leq N$, and $1 \leq i < j$ at a cost of $O(j - i) = O(N)$ for each term, the overall cost will be O(N3). To exactly find the precise cost we need to look into the code used:

```
    for l in range(2, len(l_dims)):
        for i in range(1, len(l_dims)-l+1):
            j = i+l-1
            matrix[i][j] = float("inf")
            for k in range(i, j):
                q =
matrix[i][k]+matrix[k+1][j]+l_dims[i-1]*l_dims[k]*l_dims[j]
                if q < matrix[i][j]:
                    matrix[i][j] = q
```

Minimum Multiplication Matrix function (core part)

As we can observe the cost is approximately $((2*n)+1)*n^2$, or:

$$\boxed{v(n) = 2n^3 + n^2 + O(n^3)}$$

Group 02
Daniel Varela Sánchez
Guillermo Martín-Coello Juárez