Lucas Hasting — CS 430, SP25

Homework #0 — June 19, 2025

**Question #1.** Searching in a $n \times n$ matrix: You are given a square matrix of size $n$ of integers. The matrix is stored in a 2D array. The matrix is stored so that every row is in increasing order and every column is in increasing order. For example:

| | | |
|---|---|---|
| 1 | 4 | 5 |
| 10 | 11 | 16 |
| 23 | 25 | 60 |

Devise a brute force (exhaustive) algorithm to search the array for an integer $k$. Return true if $k$ is in the matrix and false otherwise. Show (provide a convincing argument) your algorithm has a worst-case complexity of $O(n^2)$.

**Answer.** The algorithm is shown below:

```
    //Input: A 2D array A with n x n elements,
        and an item k to be searched in the array
    //Output: true (if the item is found),
        false (if the item is not found)
    procedure search(A[1...n][1...n], k)   //n > 0
1        for i := 1 to n do
2            for j := 1 to n do
3                if (A[i][j] == k)
4                    return (true)
5        return (false)
    end procedure
```

The cost table for the search procedure is shown below:

| line | cost |
|---|---|
| 1 | $n+1$ |
| 2 | $n(n+1)$ |
| 3 | $n^2$ |
| 4/5 | 1 |

Taking the sum of all the costs gives:

$$T(n) = n + 1 + n(n+1) + n^2 + 1 = n + 2 + n^2 + n + n^2$$

which simplifies to:

$$T(n) = 2(1 + n + n^2).$$

Since we only care about $T(n)$ as $n$ gets larger, we remove the constants and keep only the highest degree of n which results in:

$$T(n) = n^2 \in O(n^2).$$

**Question #2.** Devise an improved algorithm for problem #1 to search the matrix with a worst-case complexity of $O(n)$. Show (provide a convincing argument) your improved algorithm has a worst-case complexity of $O(n)$.

**Answer.** The algorithm is shown below:

```
     //Input: A 2D array A with n x n elements,
         and an item k to be searched in the array
     //Output: true (if the item is found),
         false (if the item is not found)
     procedure binary_search(A[0...n-1], k): // n > 0
1        low = 0
2        high = n^2 - 1
3        while(low <= high):
4          m = floor((low + high) / 2)
5           if(k < A[floor(m / n)][m % n]):
6                   high = m - 1
7           else if(k > A[floor(m / n)][m % n]):
8                   low = m + 1
9           else:
10                   return (true)
11       return (false)
     end procedure
```

The cost table for the binary_search procedure is shown below:

| line | cost |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 3 | $t_1$ |
| 4 | $t_1 - 1$ |
| 5 | $t_1 - 1$ |
| 6 | $t_2$ |
| 7 | $t_1 - t_2 - 1$ |
| 8 | $t_3$ |
| 9 | 1 |
| 10/11 | 1 |

First, we will look at $t_1$ closer to give a function in terms of $n$ for how many times the while loop will execute. $t_1$ is executed at most, until the size of the array is less than 1. While $t_1$ executes, the array will be split in half using floor division, over time the size becomes $\lfloor \frac{\frac{\lfloor \frac{n^2}{2} \rfloor}{2}}{\dots} \rfloor$

$t_1$ times. Using the nested division property of floor [1], it can be re-written to $\lfloor(\frac{(\frac{(\frac{n^2}{2})}{2})}{\cdots})\rfloor$ $t_1$ times, or just $\lfloor\frac{n^2}{2^{t_1}}\rfloor$.

So, $t_1$ can be expressed as the number of divisions until the size is 1 (we will let that be $t_0$), plus one more operation to give a size of 0 (which occurs whenever $k$ is not found). That is:

$$t_1 = t_0 + 1.$$

Below we show that the statements

$$\lfloor\frac{n^2}{2^{t_0}}\rfloor = 1,$$

and

$$t_0 = \lfloor log_2(n^2)\rfloor$$

are equivalent:

$$\begin{array}{lcc} & log_2(n^2) & = t_0 \\ \equiv & 2^{log_2(n^2)} & = 2^{t_0} \\ \equiv & n^2 & = 2^{t_0} \\ \equiv & \frac{n^2}{2^{t_0}} & = 1 \end{array}$$

Since

$$\frac{n^2}{2^{t_0}} = 1 \equiv t_0 = log_2(n^2),$$

then

$$\lfloor\frac{n^2}{2^{t_0}} = 1\rfloor \equiv \lfloor t_0 = log_2(n^2)\rfloor$$

which is the same as:

$$\lfloor\frac{n^2}{2^{t_0}}\rfloor = \lfloor 1\rfloor \equiv \lfloor t_0\rfloor = \lfloor log_2(n^2)\rfloor.$$

Since 1 and $t_0$ are integers, this is the same as:

$$\lfloor\frac{n^2}{2^{t_1}}\rfloor = 1 \equiv t_0 = \lfloor log_2(n^2)\rfloor.$$

Substituting the value of $t_0$ into $t_1$, we get the updated cost table as shown below:

| line | cost |
|---|---|
| 1 | $1$ |
| 2 | $1$ |
| 3 | $\lfloor log_2(n^2) \rfloor + 1$ |
| 4 | $\lfloor log_2(n^2) \rfloor$ |
| 5 | $\lfloor log_2(n^2) \rfloor$ |
| 6 | $t_2$ |
| 7 | $\lfloor log_2(n^2) \rfloor - t_2$ |
| 8 | $t_3$ |
| 9 | $1$ |
| 10/11 | $1$ |

$T(n)$ becomes the sum of the costs, which is:

$$T(n) = 4\lfloor log_2(n^2) \rfloor + 5 + t_2 + t_3 - t_2.$$

Assuming $t_2$ and $t_3$ are some constants, we only care about when $n$ gets larger, so removing all constants gives us:

$$T(n) = log_2(n^2) \in O(n).$$

**Question #3.** Identify the loop invariant for the selection sort algorithm shown below and describe how it is established and maintained. Finally, argue the algorithm is correct using the loop invariant.

```
    procedure select(A[1...n]) // n > 0
1       for i := 1 to n-1 do
2           min := i
3               for j := i+1 to n do
4                   if (A[j] < A[min]
5                   min := j
6           temp := A[i]
7           A[i] := A[min]
8           A[min] := temp
    end procedure
```

**Answer.** I will choose the loop invariant to be the partition of the array guaranteed to be sorted as $A[1...i-1]$. An unsorted array will be represented as $A[1...0]$.

First (initialization), when the loop is initialized, $i = 1$, so the partition of the array guaranteed to be sorted is $A[1...0]$ which means the array is unsorted. This is true as the array has not yet entered the loop to sort the array.

Next (maintenance), we will assume $i$ is increased at the end of line 8, and show the loop invariant is true at the beginning of the loop and after a repetition of the loop. We will assume that $n > 1$, as the guard is $i <= n - 1$, which is false only when $n = 1$. Since $i = 1$ at the beginning of the first iteration, the guaranteed sorted partition is $A[1...0]$ is true (no elements have been swapped), so both the guard and the invariant are true before the first iteration of the loop.

During each iteration, the minimum element's index is found in the partition $A[i+1...n]$ and the item in that index is swapped with the index at $i$, meaning that the guaranteed sorted partition after each iteration is $A[1...i]$. So, after the first iteration the guaranteed sorted partition is $A[1...1]$, which means the first item is guaranteed to be in the right position. Then $i$ is increased by 1 (it is assumed to be after line 8 withing the loop) which shows the loop invariant as being true as $A[1...i-1]$ becomes $A[1...2-1] = A[1...1]$.

Third (goal), we will check to see if we reached our goal. The guard statement is false whenever $i > n-1$, so when $i = n$ (the last increment of $i$ in the loop), $n > n-1$, and $A[1...n]$ is the guaranteed sorted partition, this is true as at the end of the loop $A[1...n-1]$ is guaranteed to be sorted, so the last element $A[n]$ must be in the correct position. Therefore, the negation of the guard and the loop invariant are both true.

Finally (termination), since $i$ increases, it will eventually reach $n-1$ as $1 <= n-1$ which is the initialization of $i$, and $i$ increases every iteration of the loop, so the guard will start true and eventually become false. This is under the assumption $n > 1$, because if $n = 1$, the loop will not execute. So, when $n > 1$ the loop will terminate.

**Sources**:

[1]: https://en.wikipedia.org/wiki/Floor_and_ceiling_functions

- Graham, Knuth, & Patashnik, p. 71, apply theorem 3.10 with x/m as input and the division by n as function