CS345 : Algorithms II Fall 2012

On designing efficient algorithms and proving correctness

There are two objectives of this lecture.

- To state the importance and non-triviality of the proof of correctness of an algorithm. For this purpose, we revisit the well known algorithm of binary search.
- To give glimpse of the journey we take to design efficient algorithm for a given problem. For this purpose, we consider two simple problems:
  1. finding *2-majority element* in an array A.
  2. finding a *local minima* in a $n \times n$ grid storing distinct values

1 Proof of correctness of algorithm

Consider the pseudocode given in **Algorithm 1** for binary search. The input is a sorted array A storing n elements and an element x to be searched in A. How to prove that the pseudocode

```
1  L ← 0;
2  R ← n − 1;
3  found = FALSE;
4  while L ≤ R and found = FALSE do
5  |  mid ← (L + R)/2;
6  |  if A[mid] = x then
7  |  |  found = TRUE
8  |  else
9  |  |  if A[mid] < x then
10 |  |  |  L ← mid + 1
11 |  |  else
12 |  |  |  R ← mid − 1
13 |  |  end
14 |  end
15 end
16 return found;
```

Algorithm 1: BinarySearch($A[1..n], x$)

given in Algorithm 1 is indeed a correct implementation of binary search ? Note that establishing correctness of an algorithm is even more important than analyzing its time complexity. Is it not? During ESC101 or other occasions, one tries to establish correctness of an algorithm by testing the corresponding program on some test inputs. Note that even if your algorithm works correctly on a million input instances, there is no guarantee that it will work for all possible inputs. Indeed, majority of the algorithms have infinite number of input instances. So, realize that you do not

know any way of proving correctness of an algorithm. We shall, sometime later in this course, revisit this very important issue. Till then, keep pondering over it.

2 Finding 2-majority element in an array

Given an array A storing n elements, an element $x \in A$ is said to be 2-majority element of A if it occurs more than $n/2$ times. Observe that an array A can have at most one 2-majority element.

Problem 1. *Given an array A, design an efficient algorithm to compute the 2-majority element, if exists. The only operation one can do is to check if any two elements stored in A are identical or distinct.*

There is a trivial algorithm for the 2-majority element which runs in $O(n^2)$ times: *count the occurrence of each element of A.* So our objective is to design a subquadratic, or linear if possible, time algorithm for the 2-majority element.

Note: Some of you might feel tempted to use some efficient sorting algorithms for solving the given problem. Note that majority element, if any, in an array must be the median element. So we may sort the array and then verify if the median element is indeed a majority element. Unfortunately, this approach will NOT work for this problem because of the following reason. For sorting, it is assumed that there is a total order among the elements of the set to be sorted: for any two elements x, y, exactly one of the following three assertions must hold: $x = y$, $x < y$, $x > y$. It is this feature that allows you to compare two numbers during sorting. But, as mentioned in the problem, there need not be any such total order. A concrete example, for this problem is the following. There are n credits cards and we have a machine which can receive any two cards and determine if they are identical or not. The aim is to determine if there is any card which appears more than $n/2$ times in the collection.

An array can be seen as a multiset (a set where an element can occur multiple times). From now onwards, we shall use A interchangeably as multiset as well as array. We would like to exploit the idea underlying the following lemma whose proof is left as an exercise.

Lemma 1. *Let A be a multiset and u be its majority element. If we remove any two distinct element from A, u is the majority element of the remaining multiset (after removing the two distinct elements).*

Take a break from the reading for a few minutes and think of a way to exploit Lemma 1 to design an efficient algorithm for 2-majority element.

Based on Lemma 1, Algorithm 2 will scan array A from left to right and, whenever it finds two two distinct elements it cancels off. Interestingly, the algorithm uses only three additional variables and makes two passes over A to compute the majority element of A.

It follows easily that the running time of the algorithm is $O(n)$. How to establish the correctness of the algorithm ? Firstly observe that if A does not have any 2-majority element, the output of the algorithm FINDING_2-MAJORITY_ELEMENT is correct. So we need to prove the correctness of the algorithm when array A indeed has a majority element. Well, the algorithm described above is short and iterative like binary search. Can you think of ways of proving its correctness ?

```
count ← 0;
for i = 1 to n do
    if count=0 then
        x ← A[i];
        count ← 1;
    else
        if A[i] ≠ x then
            count ← count − 1;
        else
            count ← count + 1;
        end
    end
end
Count the occurrence of x in A and if it is more than n > n/2, return x else return NIL.
```

Algorithm 2: FINDING_2-MAJORITY_ELEMENT($A[1..n]$)

3 Local minima in a $n \times n$ grid

We are given a $n \times n$ grid storing distinct values in each cell. We can use a matrix M to store the grid. Consider any $0 \leq i, j \leq n-1$. The neighbors of the entry (i, j) is the set $\{(i-1, j), (i+1, j), (i, j-1), (i, j+1)\}$. Note that entries lying on the boundary of matrix may have only 2 or 3 neighbors. We say that there is a local minima at (i, j) if $M[i, j]$ is smaller than value stored at each of the neighboring entries of (i, j).

Problem 2. *Design an efficient algorithm to report any local minima in M.*

Let us first develop some familiarity with the problem. Along the similar lines of local minima, we can define global minima as the entry of M storing the smallest value. It is easy to design an $O(n^2)$ time algorithm to compute a global minima. Since global minima is also a local minima, we have an $O(n^2)$ time trivial algorithm for finding a local minima. So our aim is to design a better algorithm for local minima (Note that the aim is to just output a local minima and not to output all local minima; otherwise we could not hope to break $O(n^2)$ bound).

Our solution will be based on two simple principles which we are quite familiar with in our life as well.

1. We should respect every new idea that solves a problem even partially or only in special cases.
2. Principle of simplification: This principle says that if a problem is difficult to solve, try to solve its simpler version and then extend it suitably.

Thinking over the problem for a few minutes, one would come up with Procedure EXPLORE(M) for finding a local minima. It is not immediate to see that the above procedure is indeed an algorithm for local minima. So let us ponder over it for a few moments. Verifying whether an entry is a local minima takes $O(1)$ time. Firstly observe that if the procedure terminates, it indeed reports a local minima. So the only issue left to establish that EXPLORE is an algorithm is the following: Does it terminates in finite time (some function of n). Note that the procedure EXPLORE always moves to

```
1 Let c be any entry in M;
2 while c is not a local minima do
3 |  c ← the neighbor of c storing value smaller than the value stored at c.
4 end
5 return c;
```

Procedure EXPLORE(M)

an entry storing value smaller than the current entry. So it will not revisit any entry twice. Since there are n^2 number of entries, it will indeed terminate in $O(n^2)$ steps. Hence the procedure we mentioned above is indeed an algorithm for local minima. However, this algorithm does not beat the $O(n^2)$ bound (Give reasons). Though the above algorithm does not give us a better worst case bound, it certainly adds to our insight into the problem. So applying the first principle (Respecting every new idea ...), we should not throw it into trash. It will prove to be helpful later.

Spending a few more minutes over the problem makes one feel that it is quite nontrivial and difficult to design an algorithm which takes less than $O(n^2)$ time. So let us apply the principle of simplification. In this case it would mean: working on a 1-dimension grid instead of a 2-dimensional grid. In other words, we need to find a local minima in an array A of size n.

3.1 An efficient algorithm for finding local minima in an array

Let A be an array storing n distinct elements and we need to find a local minima in it. We may first verify if any of the extreme ends of A are local minima. If so, we are done. Otherwise, one might like to explore the middle entry. Suppose it is also not a local minima. What should we do now ? The key idea is to visualize the array A as a sequence of vertical bars each of height equal to the value stored at the corresponding entry in A. See Figure 1.

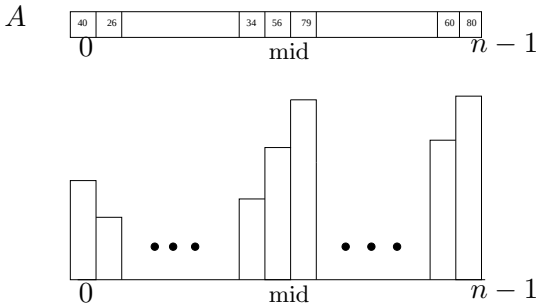

Figure 1: Visualizing the entries of array as vertical bars of appropriate heights

This visualization shows that there must be a local minima in the left half of the array. The proof is inspired from the simple algorithm EXPLORE(): If we execute algorithm EXPLORE starting from the left neighbor of $A[mid]$, the algorithm will never enter the right half of the array. This suggests an algorithm for local minima which is similar in flavor as the binary search algorithm!

Homework 1. *Design an $O(\log n)$ time algorithm to compute a local minima in an array A.*

Try to extend the above algorithm to solve our main problem: local minima in an $n \times n$ grid.