Codility

Chapter 14

Binary search algorithm

The binary search is a simple and very useful algorithm whereby many linear algorithms can be optimized to run in logarithmic time.

14.1. Intuition

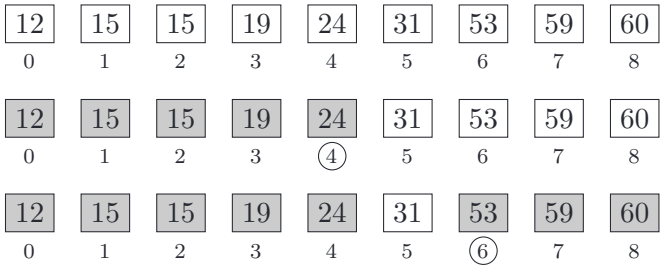
Imagine the following game. The computer selects an integer value between 1 and 16 and our goal is to guess this number with a minimum number of questions. For each guessed number the computer states whether the guessed number is equal to, bigger or smaller than the number to be guessed.

The iterative check of all the successive values $1, 2, \ldots, 16$ is linear, because with each question the set of the candidates is reduced by one.

The goal is to ask a question that reduces the set of candidates maximally. The best option is to choose the middle element, as doing so causes the set of candidates to be halved each time. With this approach, we ask the logarithmic number of questions at maximum.

14.2. Implementation

In a binary search we use the information that all the elements are sorted. Let's try to solve the task in which we ask for the position of a value x in a sorted array $a_0 \leqslant a_1 \leqslant \ldots \leqslant a_{n-1}$. Let's see how the number of candidates is reduced, for example for the value $x = 31$.



For every step of the algorithm we should remember the beginning and the end of the remaining slice of the array (respectively, variables *beg* and *end*). The middle element of the slice can easily be calculated as $mid = \lfloor \frac{beg+end}{2} \rfloor$.

14.1: Binary search in $O(\log n)$.

```
def binarySearch(A, x):
    n = len(A)
    beg = 0
    end = n - 1
    result = -1
    while (beg <= end):
        mid = (beg + end) / 2
        if (A[mid] <= x):
            beg = mid + 1
            result = mid
        else:
            end = mid - 1
    return result
```

The above algorithm will find the largest element which is less than or equal to x. In subsequent iterations the number of candidates is halved, so the time complexity is $O(\log n)$. It is noteworthy that the above implementation is universal; it is enough to modify only the condition inside the while loop.

14.3. Binary search on the result

In many tasks, we should return some integer that is both optimal and that meets certain conditions. We can often find this number using a binary search. We guess some value and then check whether the result should be smaller or bigger. At the start we have a certain range in which we can find the result. After each attempt the range is halved, so the number of questions can be estimated by $O(\log n)$.

Thus, the problem of finding the optimal value reduces to checking whether some value is valid and optimal. The latter problem is often much simpler, and the binary search adds only a $\log n$ factor to the overall time complexity.

14.4. Exercise

Problem: You are given n binary values $x_0, x_1, \ldots, x_{n-1}$, such that $x_i \in \{0, 1\}$. This array represents holes in a roof (1 is a hole). You are also given k boards of the same size. The goal is to choose the optimal (minimal) size of the boards that allows all the holes to be covered by boards.

Solution: The size of the boards can be found with a binary search. If size x is sufficient to cover all the holes, then we know that sizes $x+1, x+2, \ldots, n$ are also sufficient. On the other hand, if we know that x is not sufficient to cover all the holes, then sizes $x-1, x-2, \ldots, 1$ are also insufficient.

14.2: Binary search in $O(\log n)$.

```
def boards(A, k):
    n = len(A)
    beg = 1
    end = n
    result = -1
    while (beg <= end):
        mid = (beg + end) / 2
        if (check(A, mid) <= k):
            end = mid - 1
            result = mid
```

```
11          else:
12              beg = mid + 1
13      return result
```

There is the question of how to check whether size x is sufficient. We can go through all the indices from the left to the right and greedily count the boards. We add a new board only if there is a hole that is not covered by the last board.

14.3: Greedily check in $O(n)$.

```
1  def check(A, k):
2      n = len(A)
3      boards = 0
4      last = -1
5      for i in xrange(n):
6          if A[i] == 1 and last < i:
7              boards += 1
8              last = i + k - 1
9      return boards
```

The total time complexity of such a solution is $O(n \log n)$ due to the binary search time.

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.