Divide & Conquer: Arrays

Download the files `search.py` and `test.py`. The file `search.py` contains the skeleton code that you need to complete. The file `test.py` contains the code to test your solutions for the first part. You are encouraged to write similar test functions for the second part.

1 Searching through a sorted array

We consider the problem of searching a value in an array. An obvious solution with complexity $O(n)$, where n is the size of the array, consists in traversing the array (from left to right) until we find the input value or until we reach the end. This is usually called *linear search*. If the input array is assumed to be **sorted**, then it is possible to define more efficient algorithms using the divide & conquer pardigm. Therefore, we consider the following problem :

SEARCH-IN-SORTED-ARRAY
Input : A sorted array A and a value v
Output : An index i such that $A[i] = v$, or -1, if v does not occur in A

1.1 Binary Search

A first algorithm for solving SEARCH-IN-SORTED-ARRAY is *binary search* (seen in CSE 103) :

— compare v to the middle element of the array,

— if they are not equal, the half in which the target cannot lie is eliminated and the search recurses on the remaining half.

The function `binary_search_rec` in `search.py` contains a recursive implementation.

Question 1 (Binary search)**.** Implement `binary_search(A,v)` which solves SEARCH-IN-SORTED-ARRAY using *binary search* but in an iterative manner, without recursive calls.
Hint. Write a `while` loop that iteratively updates the left/right boundaries of the search space ; initially they are 0 and $n-1$ for an array of size n.

The function `compare_rec_while(size,nb_tests)` (in `test.py`) compares the performance of the recursive and iterative versions on arrays of length `size` (`nb_tests` is the number of tests to be run). For large enough values of `size` (≥ 10) you should observe that the iterative version runs faster.

Implement the function `cost_binary_search(n)` which computes the *worst case* cost [1] of a call in terms of comparisons (equality or less-than tests). This cost obeys the recurrence

$$C(1) = 1, \quad C(n) = C(\lceil n/2 \rceil) + 2.$$

1.2 Ternary Search

Ternary search is a variation of binary search where the input array is split in *three* parts, discarding two thirds at each iteration. The three parts are separated by

$$l + \lfloor (r-l)/3 \rfloor \text{ and } r - \lfloor (r-l)/3 \rfloor$$

assuming a sub-array from index l to index r.

1. The worst-case scenario is when the value v does not occur in the array.

Question 2 (Ternary search)**.** Implement `ternary_search(A,v)` which solves SEARCH-IN-SORTED-ARRAY using *ternary search* and without recursive calls.

Implement the function `cost_ternary_search(n)` which computes the *worst case* cost of a call in terms of comparisons (equality or less-than tests).

The function `compare_binary_ternary(n)` (in `test.py`) plots the costs of binary and ternary search, respectively, in function of the array size `n`. The blue line represents the cost of binary search while the red one the cost of ternary search. You should observe that the ternary search is not better (contrary to intuitions) ! ! ! Can you explain why ?

1.3 Exponential Search

Exponential search consists of two phases. First, it tries to find a range (l, r) in which the target value v would be present and then uses binary search inside this range to find the target's exact location (if it exists). It is named exponential search because it finds the range that could hold v by searching for the first exponent k for which the element at index 2^k is greater than the target. The index 2^k becomes the right boundary r of the search space and its "predecessor", the index 2^{k-1} becomes the left boundary l. This algorithm has been introduced for searching values in huge sized (infinite) arrays.

Question 3 (Exponential search)**.** Implement `exponential_search(A,v)` which solves SEARCH-IN-SORTED-ARRAY using *exponential search* and without recursive calls.

Implement the function `cost_exponential_search(v)` which computes the *worst case* cost of a call in terms of comparisons (equality or less-than tests). The complexity is now dominated by the target value v and not the array size. You should evaluate the number of comparisons needed to define the search space (l, r) and then, add the complexity of binary search for the *worst-case* size of that space.

1.4 Interpolation Search

Interpolation search is a variation of binary search where instead of comparing with the middle element, we compare with an element computed using linear interpolation. If the target value is closer to the first/last element, this search is likely to start the search towards the beginning/end of the array.

Interpolation search assumes that elements are linearly distributed, that is, they are of the form $A[i] = a \cdot i + b$, for some a and b. The validity of this assumption affects only performance and not correctness. Therefore, the first and last elements of a sub-array from index l to index r, and the target value v if present, would satisfy the following

$$\begin{aligned} A[l] &= a \cdot l + b \\ A[r] &= a \cdot r + b \\ v &= a \cdot k + b, \end{aligned}$$

for some index k. The equalities above imply that

$$k = l + (v - A[l]) \cdot \frac{r - l}{A[r] - A[l]}$$

We compare the target value v with $A[k]$ and recurse on $A[k+1 : r]$ or $A[l : k-1]$ as in binary search.

Question 4 (Interpolation search)**.** Implement `interpolation_search(A,v)` which solves SEARCH-IN-SORTED-ARRAY using *interpolation search* and without recursive calls.

1.5 Performance Comparison

In terms of asymptotic worst-case complexity, all the algorithms above fall under $O(log\ n)$ (for exponential search one can assume that $v = O(n)$). However, as suggested by the explanations above, an algorithm can perform better than another on some specific class of arrays : exponential search performs better than binary search on huge sized arrays, interpolation search performs better if the array elements are linearly distributed, etc.

Question 5. Run the function `compare_all(size,nb_tests)` (in `test.py`) to compare the performance of all the above search algorithms. Exponential or interpolation search start to be more efficient for larger values of `size`, e.g., exponential search for `size` ≥ 100000 and `nb_tests` ≥ 20 (results may vary from one machine to another).

2 Building on Binary Search

Question 6. Given a sorted array A, find the index of a given target's v **first** or **last** occurrence. Return -1 if the target is not present in the array. Note that binary search stops immediately as an occurrence is found. This occurrence may not be the first or the last.

Hint. Modify the binary search to continue searching even on finding the target (the direction is given by whether we are searching for the first or last occurrence).

Question 7. Given a sorted array A, a target value v, and an integer k, return the k closest elements to v (again, in an iterative manner). If v is smaller than the first array element return the first k elements and similarly, if v is bigger than the last array element, return the last k elements. The returned elements should be in the same order as in the input array. For instance,

Input : $A = [0, 2, 5, 7, 8, 11, 15], v = 6, k = 4$
Output : $[2, 5, 7, 8]$
Input : $A = [0, 2, 5, 7, 8, 11, 15], v = -1, k = 3$
Output : $[0, 2, 5]$
Input : $A = [0, 2, 5, 7, 8, 11, 15], v = 20, k = 2$
Output : $[11, 15]$

Hint. Binary search for value v and then, look around for the k closest values.

Question 8. Given a sorted array A containing duplicates, compute each element's frequency without traversing the whole array (again, in an iterative manner). Return a dictionary where keys are array elements and values their frequencies. For instance,

Input : $A = [2, 2, 2, 4, 4, 4, 5, 5, 6, 8, 8, 9]$
Output : $2 : 3, 4 : 3, 5 : 2, 6 : 1, 8 : 2, 9 : 1$

Hint. Use the divide & conquer schema in binary search and think about a "stopping" condition, that is, a class of sub-arrays for which computing frequencies is obvious.