Algorithm Analysis

Algorithm Analysis

An algorithm is a sequence of operations that accomplishes a task, or solves a problem. We demand that an algorithm be

- correct – if the algorithm's input satisfies the algorithm's assumptions, the algorithm always produces correct output –

and want an algorithm to be

- efficient – the algorithm uses the least amount of resources necessary to accomplish its task.

Today we'll learn how to analyze the efficiency of an algorithm in the context of two fundamental problems in computation: searching and sorting.

The Search Problem

The search problem is defined formally by the following input-output specifications:

- Input: A sequence of n elements A $=<$ a1, a2, ..., an $>$ and a value v
- Output: An index i such that v $=$ A[i], or a special value such as -1 or nil if v does not appear in A.

We'll see that the assumptions we can make about the input affects the efficiency of the algorithms we can use to search it.

Linear Search

If we can make no assumptions about the order of the array, our only option is linear search:

```
# A is an array, and v is the value 'were searching for
LINEAR-SEARCH(A, v):
  for i = 1 to A.length
      if A[i] = v then
          return i
return -1
```

Exercise: implement this algorithm in Python

Algorithmic Efficiency

We can characterize algorithmic efficiency in terms of space complexity (how much storage an algorithm requires) or time complexity (how "fast" an algorithm runs). Almost always primarily concerned with time complexity.

- Note that we want to eliminate platform-specific factors, like speed of the particular computer an algorithm runs on. So we characterize algorithm performance in terms of
- input size, n, and
- order of growth as a function of input size.

An efficient algorithm beats an inefficient one even if the inefficient algortithm is run on a far superior computer.

Efficiency of Linear Search

Assuming each operation has a fixed cost, we can count the operations performed for a worst-case input as follows:

Step	Cost	Times
`for i= 1to A.length`	c_1	n
`if A[i] = v then`	c_2	n
`return i`	c_3	0
`return -1`	c_4	1

Adding up the number of times each statement is executed we get:
$T(n) = c_1 n + c_2 n + c_4 = (c_1 + c_2)n + c_4$

We discard constant terms, constant factors, and lower-order terms to say that the worst case running time is

O(n)

And pronounce this “order n” or “Big-O of n.”

Binary Search

If the array is sorted you can use a binary search:

```
BINARY-SEARCH(A, v):
    p := 1, r := A.length
    while p  r
        q :=  (p+r) /2
        if A[q] = v then
            return q
        if A[q] > v then
            r := q - 1
        else
            p=q+1
    return -1
```

Intuitively: We check the midpoint of the array (q). - If the array is empty ($p > r$), the query value was not found. - If the midpoint holds the value, return the midpoint. - If the midpoint holds a value greater than our search value, repeat the process with the lower half of the array. - If the midpoint holds a value less than our search value, repeat the process with the upper half of the array.

Efficiency of Binary Search

The key to analyzing the efficiency of BINARY-SEARCH is realizing that the array is halved in each iteration of the while loop. - In the worst case BINARY-SEARCH runs until the size of the array (r − p) goes from n to 1 by successive halving. - This is equivalent to going from 1 to n by successive doubling. Counting the number of times x we need to double to get from 1 to n is

$$2x = n$$

so

$$x = lgn$$

and the worst-case running time of BINARY-SEARCH is $O(\lg n)$