COMP90038 Algorithms and Complexity

Decrease-and-Conquer-by-a-Factor

Michael Kirley

Lecture 10

Semester 1, 2016

Decrease-and-Conquer Again

The last lecture looked at a problem solving approach that went like this: To solve a problem of size n, try to express the solution in terms of a solution to the same problem of size $n-1$.

A simple example was sorting: To sort an array of length n, just

1. sort the first $n-1$ items, then
2. locate the cell that should hold the last item, shift all elements to its right to the right, and place the last element.

This led to an $O(n^2)$ algorithm, insertion sort. We can implement this with recursion or iteration (we chose iteration).

Decrease-and-Conquer by a Factor

We now look at better utilization of the approach, often leading to methods with logarithmic time behaviour or better!

Decrease-by-a-constant-factor is exemplified by binary search.

Decrease-by-a-variable-factor is exemplified by interpolation search.

Let us look at these and other instances.

Binary Search

This is a well-known approach for searching for an element k in a sorted array.

Start by comparing against the array's middle element $A[m]$.
If $A[m] = k$ we are done.

If $A[m] > k$, search the sub-array up to $A[m-1]$ recursively.

If $A[m] < k$, search the sub-array from $A[m+1]$ recursively.

$$\underbrace{A[0] \ \cdots \ A[m-1]}_{\text{search here if } A[m] > k} \quad \overset{\overset{= k?}{\downarrow}}{A[m]} \quad \underbrace{A[m+1] \ \cdots \ A[n-1]}_{\text{search here if } A[m] < k}$$

Binary Search

Again, the formulation is naturally recursive, but here we use a non-recursive formulation:

function $\textsc{BinSearch}(A[0..n-1], k)$
    $lo \leftarrow 0$
    $hi \leftarrow n - 1$
    while $lo \leq hi$ **do**
        $m \leftarrow \lfloor (lo + hi)/2 \rfloor$
        if $A[m] = k$ **then**
            return m
        if $A[m] > k$ **then**
            $hi \leftarrow m - 1$
        else
            $lo \leftarrow m + 1$
    return -1

Complexity of Binary Search

The worst-case situation for binary search is when the sought-for k is not in the array. We have the following recursive equation for the cost, in terms of the size n of the array:

$$C(n) = \begin{cases} 1 & \text{if } n = 1 \\ C(\lfloor n/2 \rfloor) + 1 & \text{if } n > 1 \end{cases}$$

A closed form for this is

$$C(n) = \lfloor \log_2 n \rfloor + 1$$

In the worst case, searching for k in an array of size 1,000,000 requires 20 comparisons.

The average-case time complexity is also $\Theta(\log n)$.

Russian Peasant Multiplication

This way of doing multiplication utilises the fact that for even n,

$$n \cdot m = \frac{n}{2} \cdot 2m$$

whereas for odd n,

$$n \cdot m = \frac{n-1}{2} \cdot 2m + m$$

We can thus proceed by halving n, repeatedly, until $n = 1$.

n	m	
81	92	92
40	184	
20	368	
10	736	
5	1472	1472
2	2944	
1	5888	5888
		7452

Finding the Median

Given an array, an important problem is how to find the median, that is, an array value which is no larger than half the elements and no smaller than half.

More generally, we would like to solve the problem of finding the kth smallest element.

If the array is sorted, the solution is straight-forward, so one approach is to start by sorting (as we'll soon see, this can be done in time $O(n \log n)$).

However, sorting the array seems like overkill.

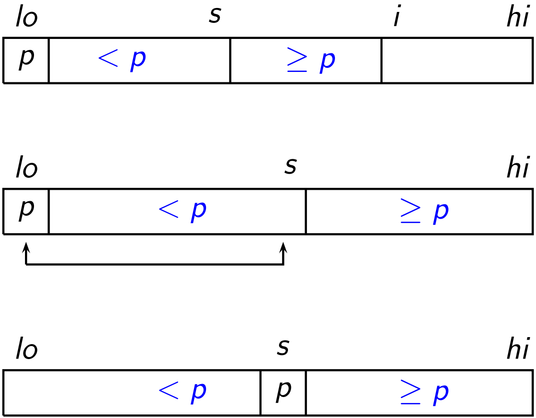
A Detour via Partitioning

Partitioning an array around some pivot element p means reorganizing the array so that all elements to the left of p are no greater than p, while those to the right are no smaller.

p		$\Rightarrow$	all $\leq p$	p	all $\geq p$

Lomuto Partitioning

function $\textsc{LomutoPartition}(A[lo..hi])$
    $p \leftarrow A[lo]$
    $s \leftarrow lo$
    for $i \leftarrow lo + 1$ **to** hi **do**
        if $A[i] < p$ **then**
            $s \leftarrow s + 1$
            $swap(A[s], A[i])$
    $swap(A[lo], A[s])$
    return s



Finding the kth Smallest Element

Here is how we can use partitioning to find the kth smallest element.

```
function QUICKSELECT(A[lo..hi], k)
    s ← LOMUTOPARTITION(A[lo..hi])
    if s = lo + k − 1 then
        return A[s]
    else
        if s > lo + k − 1 then
            QUICKSELECT(A[lo..s − 1], k)
        else
            QUICKSELECT(A[s + 1..hi], k − (s + 1))
```

Example: Finding the Fifth Smallest Element

s	i								
72	29	64	86	33	89	38	32	94	42

Example: Finding the Fifth Smallest Element

s	i								
72	29	64	86	33	89	38	32	94	42
	s	i							
72	29	64	86	33	89	38	32	94	42

Example: Finding the Fifth Smallest Element

s	i								
72	29	64	86	33	89	38	32	94	42
	s	i							
72	29	64	86	33	89	38	32	94	42
		s		i					
72	29	64	86	33	89	38	32	94	42

Example: Finding the Fifth Smallest Element

s	i								
72	29	64	86	33	89	38	32	94	42
	s	i							
72	29	64	86	33	89	38	32	94	42
		s		i					
72	29	64	86	33	89	38	32	94	42
			s			i			
72	29	64	33	86	89	38	32	94	42

Example: Finding the Fifth Smallest Element

s	*i*								
72	29	64	86	33	89	38	32	94	42
	s	*i*							
72	29	64	86	33	89	38	32	94	42
		s		*i*					
72	29	64	86	33	89	38	32	94	42
			s			*i*			
72	29	64	33	86	89	38	32	94	42
				s			*i*		
72	29	64	33	38	89	86	32	94	42

Example: Finding the Fifth Smallest Element

s	*i*								
72	29	64	86	33	89	38	32	94	42
	s	*i*							
72	29	64	86	33	89	38	32	94	42
		s		*i*					
72	29	64	86	33	89	38	32	94	42
			s			*i*			
72	29	64	33	86	89	38	32	94	42
				s			*i*		
72	29	64	33	38	89	86	32	94	42
					s				*i*
72	29	64	33	38	32	86	89	94	42

Example: Finding the Fifth Smallest Element

s	i								
72	29	64	86	33	89	38	32	94	42
	s	i							
72	29	64	86	33	89	38	32	94	42
		s		i					
72	29	64	86	33	89	38	32	94	42
			s			i			
72	29	64	33	86	89	38	32	94	42
				s			i		
72	29	64	33	38	89	86	32	94	42
					s				i
72	29	64	33	38	32	86	89	94	42
						s			i
72	29	64	33	38	32	42	89	94	86

Example: Finding the Fifth Smallest Element

Now swap 72 and 42, to bring 72 into its correct position in the array (index 6, as it was the seventh smallest element):

42 29 64 33 38 32 72 89 94 86

We wanted the fifth smallest element, so repeat the process with

42 29 64 33 38 32

42 being the new pivot.

Example: Finding the Fifth Smallest Element

s	*i*				
42	29	64	33	38	32
	s		*i*		
42	29	64	33	38	32
		s		*i*	
42	29	33	64	38	32
			s		*i*
42	29	33	38	64	32
				s	*i*
42	29	33	38	32	64

Finally the pivot is swapped into its correct position, which happens to be the fifth (index 4):

				s	*i*
32	29	33	38	42	64

Hence QUICKSELECT returns 42.

Although the worst case complexity for QUICKSELECT is quadratic, its average-case complexity is linear.

Interpolation Search

If the elements of a sorted array are distributed reasonably evenly, we can do better than binary search!

Think about how you search for an entry in the telephone directory: If you look for 'Zobel', you make a rough estimate of where to do the first probe—very close to the end of the directory.

This is the idea in interpolation search.

When searching for k in the array segment $A[lo]$ to $A[hi]$, take into account where k is, relative to $A[lo]$ and $A[hi]$.

Interpolation Search

Instead of computing, as in binary search,

$$m \leftarrow \lfloor (lo + hi)/2 \rfloor$$

we perform a linear interpolation between the points $(lo, A[lo])$ and $(hi, A[hi])$. That is, we use

$$m \leftarrow lo + \left\lfloor \frac{k - A[lo]}{A[hi] - A[lo]}(hi - lo) \right\rfloor$$

Interpolation search has average-case complexity $O(\log \log n)$.

It is the right choice for search in large arrays when elements are uniformly distributed.

Next Week

Learn to divide and conquer!

Read Levitin Chapter 5, but skip 5.4.