Finding Second Largest Element in an Array

The Problem

Input: An array $A[1..N]$ of integers and N, the number of elements in the array.

Output: The second largest number in the array.

Problem Analysis

This is an instance of the Selection problem: $V(N, k)$: given an array of numbers of size N find kth largest (or smallest) element.

Our goal here is to design an efficient algorithms for just the $V(N, 2)$ instance of this this problem. We design this algorithm in two steps.

Step 1. In a $V(N, 1)$ algorithm (i.e., selection of the largest element), the largest element **must be compared to the second-largest**.

Proof. To find the largest element, we must keep comparing elements of the array to each other, until **all but one element** lose a comparison. The second largest element **must** lose a comparison. *But it can only lose to the largest element.*

Conclusion 1. We can find the second largest element as follows:

1. Find the largest element.
2. Collect all elements of the array that were directly compared to the largest element.
3. Find the largest element among them.

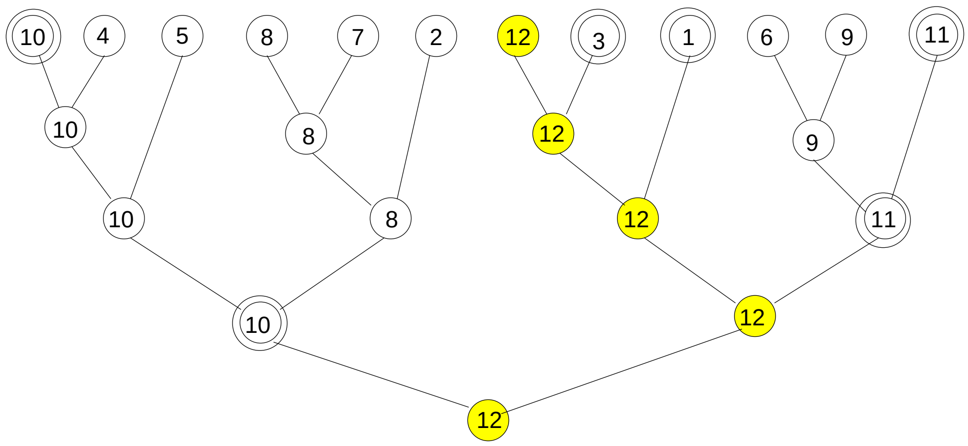

Figure 1: Example of work of FindMaxRecursive() algorithm on an array of 12 elements. Four elements (1, 3, 11, 10: double-circled above) are compared to the largest element (12, highlighted).

Step 2. To design an efficient algorithm for finding the second largest element using the observation above, we need an algorithm for finding the largest element, where the largest element is compared to *relatively few* other elements.

The simple FindMax() algorithm that uses linear scan is not good because in the worst case (A[1] is the largest element), the largest element participates in all $N-1$ comparisons.

However, the FindMaxRecursive(), that uses the tournament approach to finding the largest number will work. The work of this algorithm on an array of 12 numbers[1] is shown on Figure 1.

The algorithm itself is repeated below.

ALGORITHM FindMaxRecursive(I,J, A[I..J])
begin
  if I = J **then return** $A[I]$; `//base case`
  max1← FindMaxRecursive(I, I+(J-I)/2, A);
  max2← FindMaxRecursive(1+I+(J-I)/2,J, A);
  if max1>max2 **then return** max1
  else return max2;
end

Proposition. In ALGORITHM FindMaxRecursive() the largest element participates in at most $\lceil \log_2(N) \rceil$ comparisons.

Proof. We show that **any** element of the array A participates in no more than $\lceil \log_2(N) \rceil$.

On each recursive step of the algorithm, the algorithm splits the range $[I..J]$ (i.e, $J-I+1$ element) of elements in the array A into two parts. We observe, that the size of the first part (used in the first call) is $\lceil \frac{(J-I+1)}{2} \rceil$, while the size of the second part (used in the second call) is $\lfloor \frac{J-I+1}{2} \rfloor$. Starting with the $[1..N]$ range of N numbers, the first call (FindMaxRecursive(I,

[1] We purposefully chose the number that is not a power of 2.

```
ALGORITHM FindSecondMax(N, A[1..N]) returns
begin
  Compared←FindMaxTournament(1,N,A[1..N]);
  Compared2←FindMaxTournament(2,Compared[0],Compared[2..Compared[0]]);
  return Compared2[1]
end
```

```
FUNCTION FindMaxTournament(I,J, A[I..J],N) returns Compared[0..K]
begin
  if I = J then     //base case
    Compared[0..N];
    Compared[0]← 1;
    Compared[1]← A[I];
    return Compared;
  endif

  Compared1← FindMaxTournament(I, I+(J-I)/2, A,N);
  Compared2← FindMaxTournament(1+I+(J-I)/2,J, A,N);
  if Compared1[1]>Compared2[1] then
    K←Compared1[0]+1;
    Compared1[0]←K;
    Compared1[K]←Compared2[1];
    return Compared1;
  else
    K←Compared2[0]+1;
    Compared2[0]←K;
    Compared2[K]←Compared1[1];
    return Compared2;
  endif
end
```

Figure 2: Algorithm FindSecondMax() for finding the second largest element in an array, and function FindMaxTournament() used in it.

.

I+(J-I)/2, A)) can be recursively repeated $\lceil \log_2(N) \rceil$ times. Each recursive call yields a single comparison. Therefore each element of the array can be compared to no more than $\lceil \log_2(N) \rceil$ other numbers.

Efficient Algorithm for Finding the second largest element

Using the two observations from above, an efficient algorithm for finding the second largest number will work as follows:

1. Find the largest element of the array using the *tournament method*.
2. Collect all elements of the array that were compared to the largest element.
3. Find the largest element among the elements collected in step 2 (can use any method here).

Step 2 Issues. How can we *efficiently* collect all elements of the input array that were compared to the largest element of the array?

Essentially, we would like to associate with the largest element of A an array $Compared[]$ of elements A was compared to. This, however, needs to be done carefully. We do not know upfront which array element is going to be the largest, so we will carry information about all comparisons an array element won *until this element loses a comparison.*

From the technical side, we will change the FindMaxRecursive() algorithm to return an array of integers $Compared[0..K]$. We will establish the following convention:

- $Compared[0] = K$ (i.e., the 0th element of the array holds the length of the array);
- $Compared[1] = \mathsf{max}$ (i.e., the first element of the array is the number that FindMaxRecursive() would return;
- $Compared[2], \ldots, Compared[K]$ are all numbers with which max has been compared thus far.

Using these conventions, the ALGORITHM FindSecondMax() can be written as shown in Figure 2.

Algorithm Analysis: Running time and number of comparisons. From our study of FindMax() and FindMaxRecursive() algorithms, we know that both of them have running time $O(N)$ and use $N-1$ comparisons for an array of size N.

Using the **Proposition** from above, we analyze the running time and the number of comparisons for FindSecondMax() as follows:

- The first call to FindMaxTournament() uses $N-1$ comparisons and has running time $O(N)$.
- The second call to FindMaxTournament() passes an array of size at most $\lceil \log_2(N) \rceil$ and therefore, it uses $\lceil \log_2(N) \rceil - 1$ comparisons and runs in $O(\log_2(N))$.
- Therefore, FindSecondMax():
  - Uses $N - 1 + \lceil \log_2(N) \rceil - 1 = N + \lceil \log_2(N) \rceil - 2$ comparisons;
  - Has running time $O(N) + O(\log_2(N)) = O(N)$.

Lower Bounds

Note: In general, proving lower bounds on running time of algorithms is very hard, and requires long proofs.

Trivial lower bounds. Time to read input. If size of input is N, **and** the problem requires all input to be read, than $\Omega(N)$ is a *trivial* lower bound on the running time for any algorithm solving the problem.

Lower Bound for Finding Largest Number

Theorem. Finding the largest number in an array of N numbers **cannot be done** in less than $N-1$ comparisons.

Proof. Each comparison eliminates one candidate for the largest number. In an array of N numbers $N-1$ numbers **must be eliminated**. Therefore, *at least* $N-1$ comparisons must take place.

Note. Compining with the known upper bound on the number of comparisons, we obtain:

> Algorithms FindMax() and FindMaxRecursive() are optimal in terms of number of comparisons.

Lower Bound for Finding Second Largest Number

We state the following theorem without proof.

Theorem. Finding the largest number in an array of N numbers **requires at least** $N + \lceil \log_2(N) \rceil - 2$ comparisons.

Note. Combined with our upper bound we obtain the following:

> Algorithm FindSecondMax() is optimal in the number of comparisons.