Algorithm — II

Divide-and-Conquer

```
Algorithm DAndC (P) { // P is the problem
    if Small(P) then return S(P);
        // Small(P) is a boolean fn that determines whether the input
        // is small enough that the answer can be computed
        // without splitting.
        // If it is so, then S(P) is invoked
    else {
        divide P into smaller instances P1, P2, ..., Pk, k >= 1;
        apply DAndC to each of these smaller instances;
        return Combine(DAndC(P1), DAndC(P2), ..., DAndC(Pk));
    }
}
```

Binary Search:

Let a_i, $1 \leq i \leq n$ be a list of elements in non-decreasing order.
The problem is to determine whether a given element x is present in the list and if it is present, determine j such that $a_j = x$.
If x is not present then set $j = 0$.
Let the problem be $P = (n, a_i, \ldots, a_l, x)$ at an arbitrary instance.
Let Small(P) be true if $n = 1$
In that case, S(P) will take the value i if $x = a_i$ else $S(P) = 0$.
Also, the time taken for this is $\Theta(1)$.
If P has more than one element, it can be divided into a new subproblem as follows:

Pick an index $q \in \{i, .., l\}$ and compare x with a_q.

(i) $x = a_q \Rightarrow$ problem is solved

(ii) $x < a_q \Rightarrow$ x to be searched in $a_i, a_{i+1}, \ldots, a_{q-1}$ and P becomes $(q-i, a_i, \ldots, a_{q-1}, x)$

(iii) $x > a_q \Rightarrow$ x to be searched in $a_{q+1}, \ldots, a_l$ and P becomes $(l-q, a_{q+1}, \ldots, a_l, x)$.

Reduction into new subproblem takes $\Theta(1)$ time.
If q is always chosen such that a_q is the middle element, i.e, $q = \lfloor (n+1)/2 \rfloor$ then the algorithm is known as binary search.
See that the answer to the new subproblem is answer to the original problem P; thus, there is no need to "combining"!

```
Algorithm BinarySearch(a, i, l, x) {
    if (l = i) then {
        if (x = a[i]) then return i;
        else return 0;
    }
    else {
        mid = ⌊(i+l)/2⌋;
        if (x = a[mid]) then return mid;
        else if (x < a[mid]) then return BinarySearch(a, i, mid-1, x);
            else return BinarySearch(a, mid+1, l, x);
    }
}
```

Correctness of Binary Search:

Assume that all statements work as expected and that comparisons such as $x < a[mid]$ are appropriately carried out.

Initially low $= 1$, high $= n$, $n \geq 0$ and $a[1] \leq a[2] \leq \cdots \leq a[n]$.

If $n = 0$, the while loop is not entered and 0 is returned.

Otherwise we observe that each time through the loop the possible elements to be checked for equality with x are $a[low]$, $a[low+1]$, $\ldots$, $a[mid]$, $\ldots$, $a[high]$. If $x = a[mid]$ then the algorithm terminates successfully. Otherwise the range is narrowed by either increasing low to mid+1 or decreasing high to mid−1.

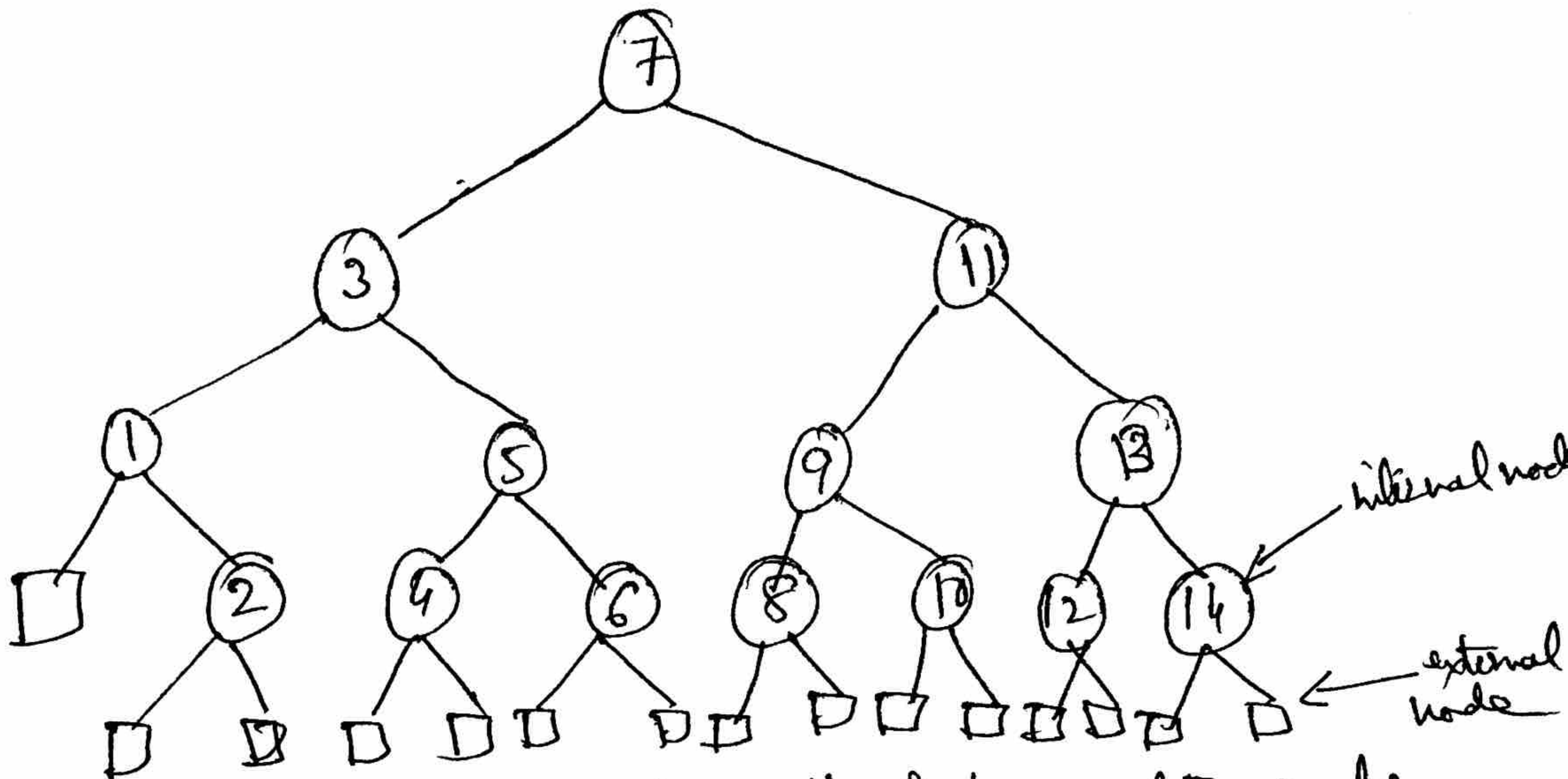
Clearly the narrowing of the range does not affect the outcome of the search. If low becomes greater than high, then x is not present and hence the loop is ~~exited~~ exited.

Bounds of Binary Search:

Let $n \in [2^{k-1}, 2^k)$.

Consider a binary decision tree with nodes for mid.

For example, for $n = 14$, we have the following:



If x is present, then the algorithm will end at one of the circular nodes that lists the index into the array where x is found.
If x is not present, the algorithm will terminate at one of the square nodes.
All successful searches end at a circular node whereas all unsuccessful searches end at a square node.
If $2^{k-1} \le n < 2^k$, then all circular nodes are at levels $1, 2, \ldots, k$ whereas all square nodes are at levels k and $k+1$ (with root at level 1).
The number of element comparisons needed to terminate at a circular node on level i is i whereas the number of element comparisons needed to terminate at a square node at level i is only $i-1$.
So, the time for a successful search is $O(\log n)$ and the time for an unsuccessful search is $\Theta(\log n)$.

Max-Min:

Let a_i, $1 \le i \le n$ be a list of elements.
The problem is to find the maximum and the minimum items.
Let $P = (n, a[i], \ldots, a[j])$ denote an arbitrary instance of the problem.
Let Small(P) be true when $n \le 2$. In this case, the maximum and minimum are a[i] if $n = 1$. If $n = 2$, the problem can be solved by making one comparison.
If there are more than two elements, P has to be divided into smaller instances.
For example, we might divide P into the two instances,

$P_1 = (\lfloor n/2 \rfloor, a[1], \cdots a[\lfloor n/2 \rfloor])$ and
$P_2 = (n - \lfloor n/2 \rfloor, a[\lfloor n/2 \rfloor + 1], \ldots, a[n])$.

After dividing P into two smaller subproblems, we can solve them by recursively invoking the same divide-and-conquer algorithm.
How can we combine the solutions for P_1 and P_2 to obtain a solution for P?
If MAX(P) and MIN(P) are the maximum and minimum of the elements in P, then MAX(P) is ~~larger~~ the larger of MAX(P_1) and MAX(P_2). Also, MIN(P) is the smaller of MIN(P_1) and MIN(P_2).

```
Algorithm MaxMin(i, j, max, min) {
        if (i = j) then max = min = a[i];
        else if (i = j-1) then {
            if (a[i] < a[j]) then {
                max = a[j]; min = a[i];
            }
            else {
                max = a[i]; min = a[j];
            }
        }
        else {
            mid = ⌊(i+j)/2⌋;
            MaxMin (i, mid, max, min);
            MaxMin (mid+1, j, max1, min1);
            if (max < max1) then max = max1;
            if (min > min1) then min = min1;
        }
}
```

II