Ex. Binary Search

Let \( A \) be an array of integers (say) with \( n = \text{length}[A] \). We will adopt the convention that array indices begin at 1. Thus

\[
A[1 \ldots n] = (A[1], \ldots, A[n])
\]

Let \( A[p \ldots r] \) denote the subarray

\[
A[p \ldots r] = (A[p], \ldots, A[r])
\]

If \( p > r \), we understand this to be an empty array.

Assume \( A[1 \ldots n] \) is sorted in increasing order (with possible repeated elements.)

Binary search is a D&C algorithm which locates a given target \( t \) in the subarray \( A[p \ldots r] \). If an index \( i \) is found such that \( A[i] = t \), then \( i \) is returned, otherwise \( 0 \) is returned.
BinSearch \((A, p, r, t)\) (Ass: \(A[p..r]\) sorted.)

1.) if \(p > r\)
2.) return \(0\)
3.) \(q \leftarrow \left\lfloor \frac{p + r}{2} \right\rfloor\)
4.) if \(A[q] = t\)
5.) return \(q\)
6.) if \(A[q] < t\)
7.) return BinSearch \((A, q+1, r, t)\)
8.) return BinSearch \((A, p, q-1, t)\)

Ex. \(A = (1, 2, 3, 4, 5, 6, 7, 8, 9)\)

1 2 3 4 5 6 7 8 9

This binary search tree represents the order in which the target \(t\) is compared to elements of \(A\).

The call to BinSearch on the full array \(A[1..n]\) is just

\(\text{BinSearch} (A, 1, n, t)\)
Theorem (Correctness of BinSearch)

BinSearch returns either an index \( i \) such that \( A[i] = t \), or returns \( 0 \) if no such \( i \) exists.

Proof

Use induction on \( m = r - p + 1 = \text{length}[A[p \ldots r]] \).

I. Base

If \( m = 0 \) the subarray does not contain target \( t \). Also \( m = 0 \) implies \( r = p - 1 < p \), so that \( 0 \) is returned on line 2.

II. (Strong) Induction

Let \( m > 0 \) and assume that BinSearch returns the correct index on any subarray of length less than \( m \).

Now \( m > 0 \) implies \( r > p - 1 \) whence \( r \geq p \). Thus \( \mathfrak{d} = \lceil \frac{p + n}{2} \rceil \) (line 3) implies \( p \leq \mathfrak{d} \leq r \).

If \( A[\mathfrak{d}] = t \) then obviously BinSearch returns correct index on line 5.

If \( A[\mathfrak{d}] < t \) then \( A[\{\mathfrak{d}+1 \ldots r\}] \) is a subarray of length

\[ r - (\mathfrak{d} + 1) - 1 = r - \mathfrak{d} - 1 \leq r - p < r - p + 1 = m. \]
The induction hypothesis guarantees that a correct value is returned on line 7.

If on the other hand, $A[q] = t$ then $A[p\ldots(q-1)]$ is of length

$$(q-1)-p+1 = q-p < r-p+1 = n,$$

so the induction hypothesis insures that a correct value is returned on line 8.

In all cases a correct value is returned, and the proof is complete.

The (worst case) analysis of binary search is very simple.

$$T(n) = \begin{cases} \Theta(1) & n = 0 \\ 1 + T(\frac{n}{2}) + \Theta(1) & n \geq 1 \end{cases}$$

$$n \log_2 n = n^0 = 1 = \Theta(1),$$

so case 2 of the master theorem says

$$T(n) = \Theta(\log n).$$
EX. MERGE SORT

At T=0, A is an array of numbers with n = length(A).

```
MergeSort recursively sorts A subarray A[p...r] of A[1...n] where 1 ≤ p ≤ r ≤ n.
```

```
MergeSort(A, p, r)
```

1. if r ≤ p
2. q ≤ \lfloor \frac{p + r}{2} \rfloor
3. MergeSort(A, p, q)
4. MergeSort(A, q + 1, r)
5. Merge(A, p, q, r)

```
```

Merge(A, p, q, r) is a sub-algorithm which combines the sorted sub-arrays A[p...q] and A[q+1...r] into one sorted subarray A[p...r].

```
```

\[
\begin{array}{c}
\text{Sorted} \\
A[p...q] \\
\end{array}
\quad \quad
\begin{array}{c}
\text{Sorted} \\
A[(q+1)...r] \\
\end{array}
\quad \quad
\downarrow \\
\text{Merge} \\
\end{array}
\]

\[
A[p...r] \\
\text{Sorted}
\]
To sort the entire array $A[1...n]$ we call MergeSort($A, 1, n$).

**Theorem.**

After MergeSort($A, p, r$) is called, the sub-array $A[p...r]$ is sorted.

**Proof.**

We use induction on $m = r - p + 1 = \text{length}[A[p...r]]$.

I. If $m = 1$ then $r = p$ and $A[p...r]$ is already sorted. Indeed, the algorithm does nothing in this case.

II. Let $m > 1$ and assume that any call to MergeSort on a subarray of length less than $m$ results in that subarray being sorted.

Now $m > 1$ implies $r > p$, so line 2 is executed setting $q = \lfloor \frac{p + r}{2} \rfloor$. This implies $p < q < r$:

$$2p < p + r \Rightarrow p < \frac{p + r}{2} \Rightarrow p \leq \lfloor \frac{p + r}{2} \rfloor \Rightarrow p < q < r \Rightarrow p + r < 2r \Rightarrow \frac{p + r}{2} < r \Rightarrow \lfloor \frac{p + r}{2} \rfloor < r \Rightarrow p < q < r$$
Thus

\[ \text{length } [A[p..q]] = q - p + 1 < r - p + 1 = m \]

and

\[ \text{length } [A[(q+1)..<r]] = r - (q+1) + 1 = r - q \leq r - p < r - p + 1 = m \]

The induction hypothesis therefore insures that \( A[p..q] \) and \( A[(q+1)..<r] \) are sorted after lines 3 and 4 are executed. Hence \( A[p..<r] \) is sorted after the execution of line 5.

This completes the proof.

Analysis (Worst Case)

\[
T(n) = \begin{cases} 
\Theta(1) \\
T(\lceil n/2 \rceil) + T(\lceil m/2 \rceil) + \Theta(n) 
\end{cases}
\]

This can be simplified to \( T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n) \). And the Master Theorem (Case 2) gives

\[ T(n) = \Theta(n \log n) \]
Ex. Quicksort. (7.1-7.4)

Again, A[p ... r] is sorted recursively.

Quicksort(A, p, r)
1.) if p < r
2.) q = Partition(A, p, r)
3.) Quicksort(A, p, q-1)
4.) Quicksort(A, q+1, r)

Partition(A, p, r)
1.) i < p-1
2.) for i = p to (r-1)
3.) if A[i] <= A[r]
4.) i <= i+1
5.) A[i] <= A[i]
7.) return (i+1)

The subroutine Partition(A, p, r) rearranges A[p ... r] into two (possibly empty) subarrays A[p .. (q-1)] and A[(q+1) ... r] such that:

\[ A[p .. (q-1)] \leq A[q] \leq A[(q+1) .. r] \]

\[ \text{not sorted} \quad \uparrow \quad \text{pivot} \quad \text{not sorted} \]
Ex. Partition

A_1 \rightarrow

8 6 1 3 7 2 5

A_1 \leftarrow \text{Pivot}

8 6 1 3 7 2 5

8 6 1 3 7 2 5

1 6 8 3 7 2 5

1 6 8 3 7 2 5

1 3 8 6 7 2 5

1 3 8 6 7 2 5

1 3 2 6 7 8 5

1 3 2 6 7 8 5

1 3 2 6 7 8 5

1 3 2 6 7 8 5

\underline{1 3 2 4}

\begin{array}{c}
A[p..(4-1)]
\end{array}

\rightarrow

\begin{array}{c}
A[(4+1)..r]
\end{array}

\text{Pivot}
**Exercise**

- Verify these claims on examples.
- Prove them.

It follows that when `partition` returns:

\[ A[p..(q-1)] \leq A[q] \leq A[(q+1)..n] \]

The run time of `partition` (in terms of worst, and average cases) is \( \Theta(m) \) where \( m = q - p + 1 \), since loop 2-s steps through the entire subarray \( A[p..(q-1)] \), then the pivot is set in place.
Exercise:
Prove the correctness of Quicksort by induction on the length of the sub-
array \( A[p...r] \) : \( n = r-p+1 \).

The run time of Quicksort depends heavily on the value of \( n \) and
returns by partition. If the subarrays \( A[p...(r-1)] \) and \( A[r+1...r+n] \)
are not balanced (i.e., of roughly equal size), then performance is
insulted. In this case, one recursive call to Quicksort is on a subarray which is
inefficiently long.

The worst case occurs when the
array is already sorted, then partition returns

\[
A[p...(r-1)] \leq A[r] \leq \ldots \text{empty...}
\]

\[
\text{sorted} \quad 4
\]

Let \( T(n) \) denote the worst case run

time of Quicksort (i.e., with \( A[1...n] \)
already sorted.)