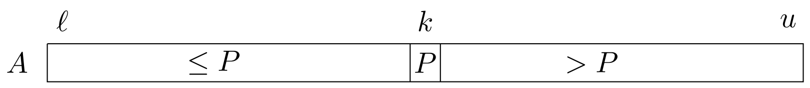
Lecture Notes, Fundamental Algorithms: The Quicksort Partition Procedure

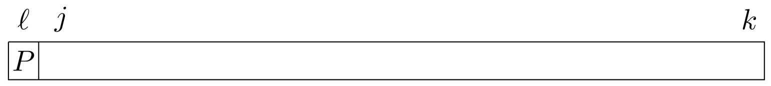
The Procedure Partition(A, ℓ, u, k) is given an array $A[\ell : u]$ of items, and partitions them about a pivot P, the item initially in $A[\ell]$, so that they are distributed as shown.

Figure 1:



Initially, we have:

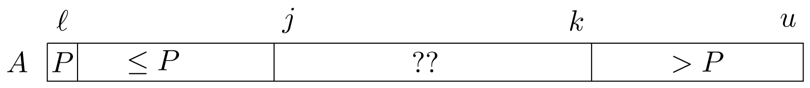
Figure 2:



with $j = \ell + 1$ and $k = u$.

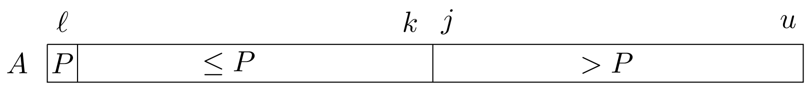
Partition will maintain the following ordering as it proceeds.

Figure 3:



The subarray $A[j : k]$ contains the items that have not yet been ordered relative to $A[\ell] = P$. So initially, $j = \ell + 1$ and $k = u$. Then so long as $A[j] \leq P$ we can keep incrementing j, and while $A[k] > P$ we can keep decrementing k. If both $A[j] > P$ and $A[k] \leq P$ it is safe to swap them, increment j and decrement k. We just iterate this process. Eventually, $A[j : k]$ will be empty, which will be indicated by having $j > k$, as shown below.

Figure 4:



We finish by swapping $A[\ell]$ and $A[k]$.

Pseudo-code is shown below.

```
Partition($A, \ell, u, k$)
    $P \leftarrow A[\ell]$; $j \leftarrow \ell + 1$; $k \leftarrow u$;
    **repeat**
        **while** ($j \leq k$ **and** $A[j] \leq P$) **do** $j \leftarrow j + 1$;
        **while** ($k \geq j$ **and** $A[k] > P$) **do** $k \leftarrow k - 1$;
        **if** $j < k$ **then** Swap($A[j], A[k]$); $j \leftarrow j + 1$; $k \leftarrow k - 1$
    **until** $j > k$;
    Swap($A[\ell], A[k]$)
```

The condition $k \geq j$ in the second while loop is not needed as $A[\ell] = P$, so k cannot be reduced below ℓ in value.

The ordering shown in Figure 3 is called an Invariant of the repeat loop. It is true at the start and end of each iteration of the loop, as is easily verified by looking at what each line of this code does. This implies that at the end of the repeat loop, when $j > k$, the configuration of the array is as shown in Figure 4. Therefore the final swap yields the desired final configuration as shown in Figure 1.

Invariants allow us to explain why loops do what we intend them to do. An invariant specifies the state of the variables at the point they hold, such as the beginning and end of each iteration. We can then deduce they also hold at the conclusion of the loop, along with the termination condition. Note that an invariant is a static property: it tells you what is true at a given point in the program. It does not tell you what happens during an iteration. But this separation of concerns actually makes correct loop design easier. The invariant tells you the property you want to maintain. It is then your task to figure out how to make progress while ensuring the invariant remains true.

Invariants are also useful for describing the input-output properties of procedures. They describe what the procedure does, not how it carries out its task.