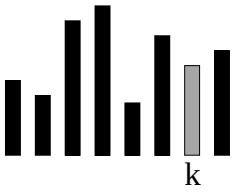
Quicksort

Quicksort is a *divide-and-conquer* method for sorting. It works as follows:

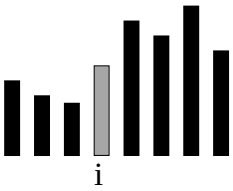
- **Selection**: A **pivot** element is selected.
- **Partition**: Place all of the smaller values to the left of the pivot, and all of the larger values to the right of the pivot. The pivot is now in the proper place.
- **Recur**: Recursively sort the values to the left and to the right of the pivot.

Quicksort: Illustration

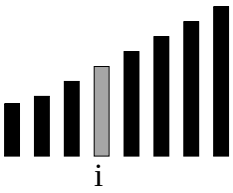
1. **Selection**: Pick a pivot $a[k]$.



2. **Partition**: Rearrange the elements so that:
   - $a[k]$ is in its final position $a[i]$
   - $j < i \Rightarrow a[j] \leq a[i]$
   - $j > i \Rightarrow a[j] \geq a[i]$

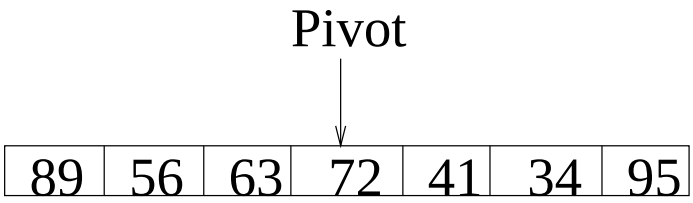


3. **Recur**: Recursively sort the subarrays $a[1, \ldots, i-1]$ and $a[i+1, \ldots, n]$.

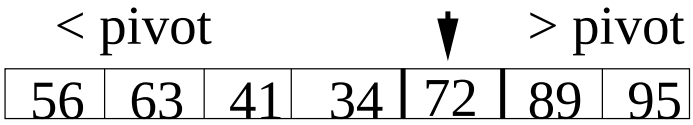


Quicksort Example

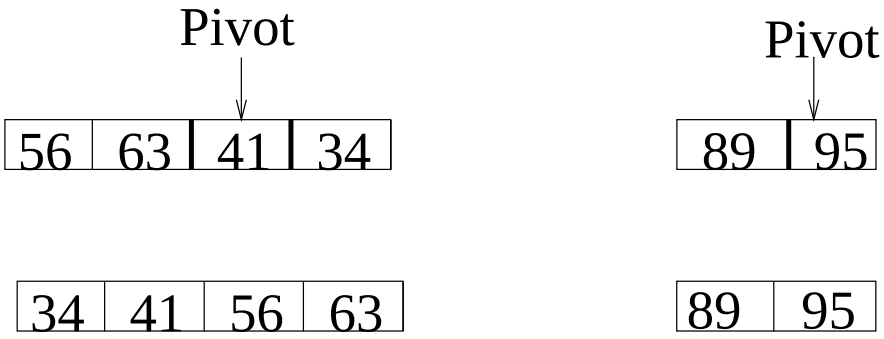
- Select the pivot.



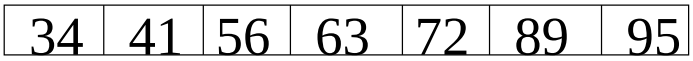
- Put the smalls on the left and bigs on the right.



- Continue recursively.



- Finish.



Quicksort: The Algorithm

```
void Quicksort(int A[],int l, int r) {
   if (r > l) {
     int p = Partition(A,l,r);
     Quicksort(A,l,p-1);
     Quicksort(A,p+1,r);
     }
}
```

- The function **Partition**
  - picks some k, $l \leq k \leq r$;
  - places $x = A[k]$ in its proper location, p;
  - assures that $A[j] \leq A[p]$ if $j < p$; and
  - assures that $A[j] \geq A[p]$ if $j > p$.
- This is not the only way to define quicksort.
- Some versions of **Quicksort** don't use a pivot element, but instead define left and right subarrays, promising that the values in the left array are less than those in the right array.

A Closer Look at Partition

- There are various methods that can be used to pick the pivot element, including:
  - Use leftmost element as the pivot.
  - Use the "median-of-three" rule to pick the pivot.
  - Use a random element as the pivot.
- After the pivot is picked, how do we put it in place and insure the partition properties?
  - There are several ways of accomplishing this.
  - We will look at one way to implement the "leftmost pivot" strategy.
  - The other methods can be implemented by slightly modifying this method.

The Standard Partition

- Here is one way to implement the leftmost pivot method.

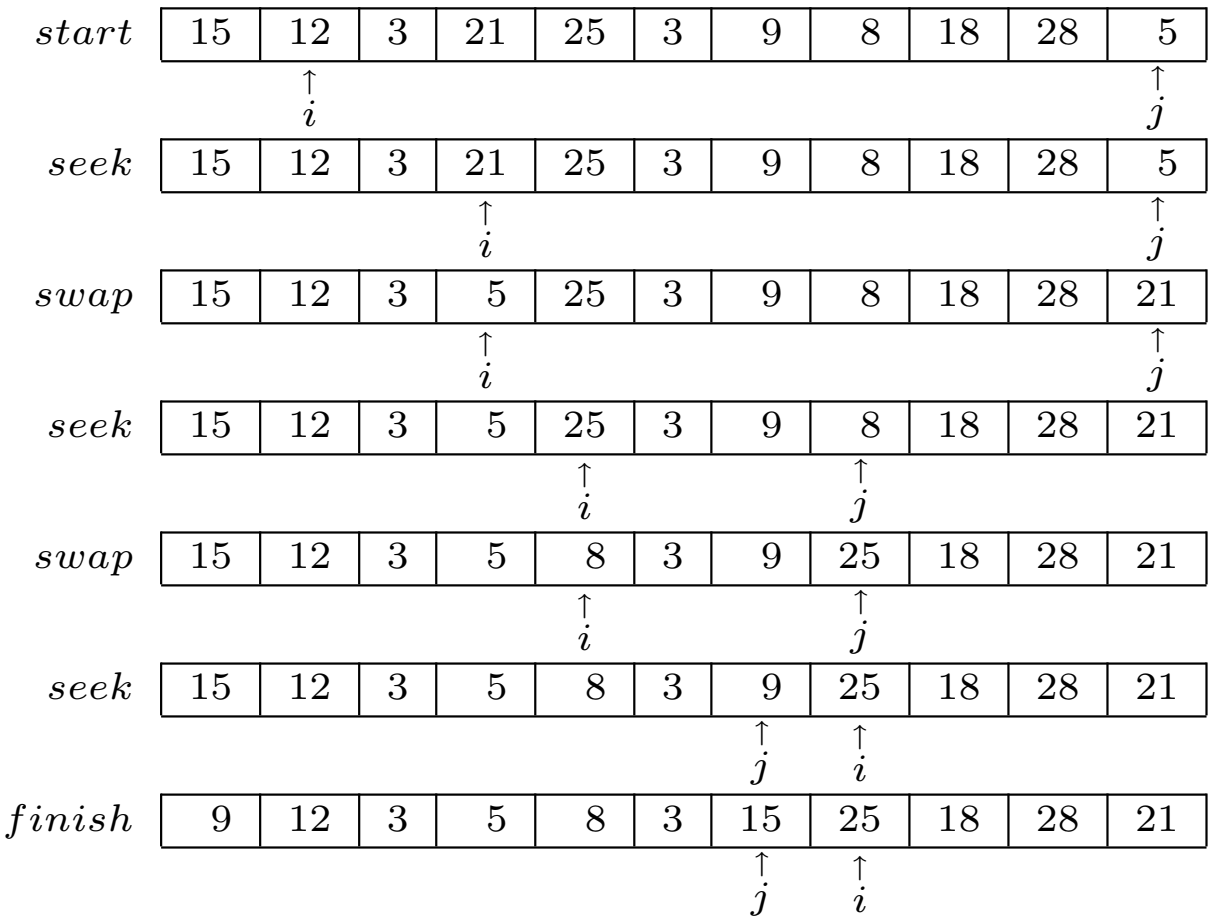
```
int Partition(int A[],int L,int R) {
    int i=L+1;
    int j=R;
    while(1) {
        while(i<=R && A[i]<A[L]) i++;
        while(A[j]>A[L])         j--;
        if(i<j) {
            Swap(A[i],A[j]);
            i++;
            j--;
        }
        else {
            Swap(A[L],A[j]);
            return j;
        }
     }
}
```

- We pick the leftmost element as the pivot.
- We travel from the endpoints to the center, swapping elements that are in the wrong half.

Partition Example

15	12	3	21	25	3	9	8	18	28	5

- Here, $p = 15$.



- The pivot has index 6, so *Partition* returns the value 6.

Other Choices for Pivot

- **Random:**
  - We pick a random element of the sequence as the pivot.
  - The average complexity does not depend on the distribution of input sequences.
- **Median-of-Three:**
  - The partitioning element is chosen to be:

$$median(a[l], a[(l+r)/2], a[r])$$

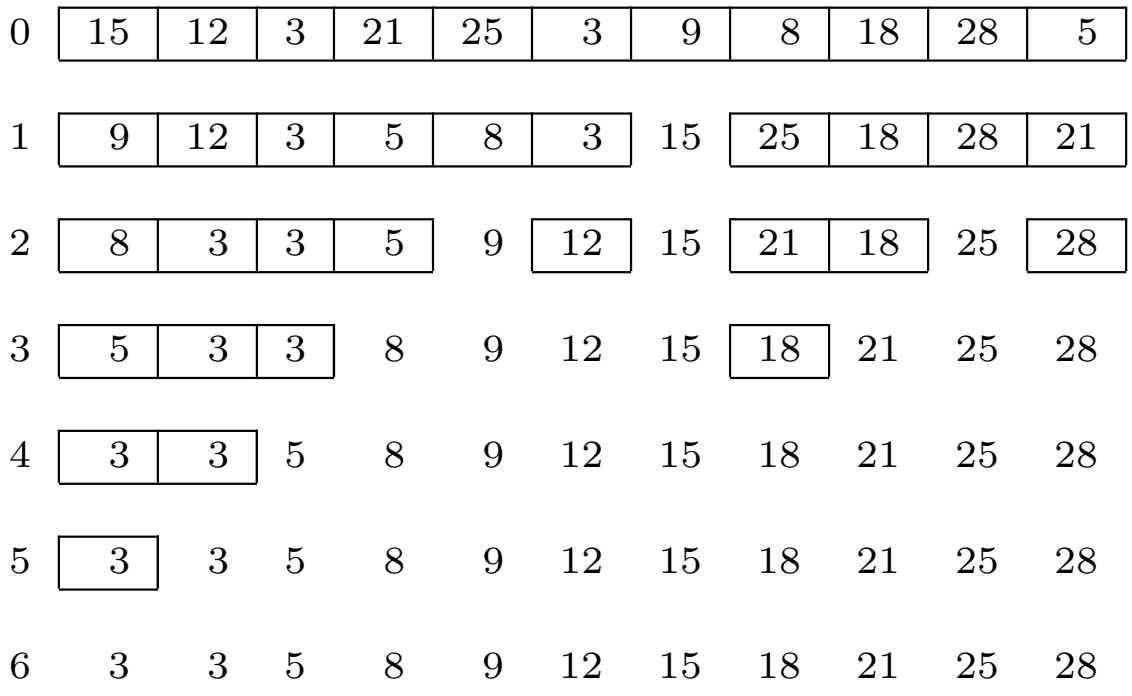
  - This method is unlikely to generate a "degenerate" partition.
- How can we modify the previous Partition to implement these?
- Is there any better way to implement these?
- Is there any better way to choose pivot?

Other Considerations

- **Correctness:** We need to consider two factors
  - Since the algorithm is recursive, does it terminate?
  - When it does terminate, does it return a sorted list?
- **Optimization:**
  - What happens if we sort small subsequences with another algorithms?
  - What if we ignore small subsequences? That is, we use **Quicksort** on subarrays larger than some threshold. Can we then finish sorting with another algorithm for increased efficiency? (Hint: The array is now mostly sorted.)

Analyzing Quicksort

- Partition can be done in time $O(n)$, where n is the size of the array. Can you prove this?
- The cost of all calls to partition at any depth in the recursion has time complexity $O(n)$, where n is the size of the original array. Can you prove this?
- Thus, the complexity of **Quicksort** is the complexity of partition times the depth of the furthest recursive call.
- **Example:**



Quicksort Time Complexity

- The operation which is performed most often is the *compare*. Can you prove this?
- Let $T(n)$ be the number of *compares* required by **Quicksort**.
- If the pivot ends up at position i, then we have

$$T(n) = T(n-i) + T(i-1) + n.$$

  Unfortunately, i may be different for each subarray, and at each level of recursion.
- We have seen already that $T(n) = O(n \times k)$, where k is the depth of the deepest recursive call.
- What is k in general?
- We will look at the best case, worst case, and average case complexities.

Best and Worst Case Complexity

- **Best-case:**
  - The best case is clearly if the pivot always ends up at the center of the subarray.
  - Intuitively, this would lead to recursive depth of at most $\log n$.
  - We can actually prove this. We have

    $$T(n) \leq 2T(n/2) + n.$$

    We have seen previously that

    $$T(n) = \Theta(n \log n).$$

- **Worst-Case**
  - in the worst-case the pivot is always the first or last element of the subarray.
  - In this case, we have

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= n + (n-1) + \ldots + 1 \\ &= n(n+1)/2 = O(n^2) \end{aligned}$$

Average Case Complexity

- We assume that the pivot has the same probability $(1/n)$ to go into each of the n possible positions. This gives

$$\begin{aligned} T_a(n) &= n + \frac{1}{n}\sum_{i=1}^{n}(T_a(i-1) + T_a(n-i)) \\ &= n + \frac{1}{n}\sum_{i=1}^{n} T_a(i-1) + \frac{1}{n}\sum_{i=1}^{n} T_a(n-i) \\ &= n + \frac{2}{n}\sum_{i=1}^{n} T_a(i-1) \end{aligned}$$

- The last step comes from the fact that the two sums are the same, but in reverse order.
- We have seen that in the best case, $T(n) = O(n \log n)$ and in the worst case, $T(n) = O(n^2)$.
- We guess that $T_a(n) \leq an \log n + b$, for two positive constants a and b.
- We will prove this by induction.

Proof that $T(n) \leq an \log n + b$

- We can pick a and b so that the condition holds for $T(1)$.
- Assume it holds for all $k < n$. Then

$$\begin{aligned} T_a(n) &= n + \frac{2}{n} \sum_{k=1}^{n} T_a(k-1) \\ &= n + \frac{2}{n} \sum_{k=1}^{n-1} T_a(k) \\ &\leq n + \frac{2}{n} \sum_{k=1}^{n-1} (ak \log k + b) \\ &= n + \frac{2b}{n}(n-1) + \frac{2a}{n} \sum_{k=1}^{n-1} k \log k \end{aligned}$$

- It can be shown that

$$\sum_{k=1}^{n-1} k \log k \leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2.$$

- Substituting, we get

$$\begin{aligned}
T_a(n) &= n + \frac{2b}{n}(n-1) + \frac{2a}{n} \sum_{k=1}^{n-1} k \log k \\
&\leq n + \frac{2b}{n}(n-1) + \frac{2a}{n} \left(\frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) \\
&= n + 2b - \frac{2b}{n} + an \log n - \frac{a}{4} n \\
&= an \log n + b + (n + b - \frac{2b}{n} - \frac{a}{4} n) \\
&\leq an \log n + b + (n + b - \frac{a}{4} n) \\
&\leq an \log n + b.
\end{aligned}$$

- The last step can be obtained by choosing a large enough.
- Thus, $T(n) = O(n \log n)$.