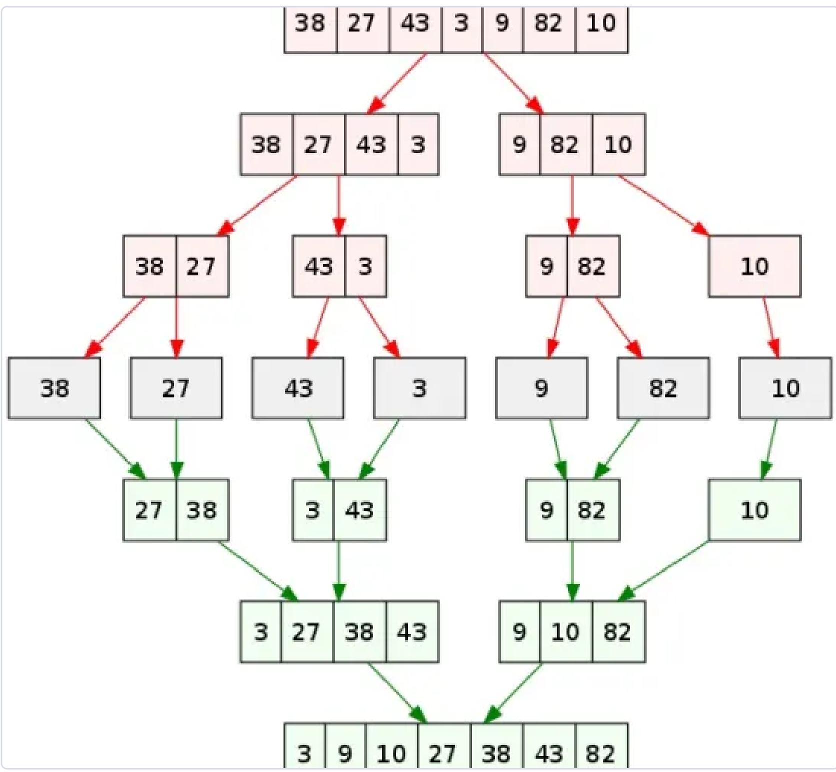
Sorting 3

Quick Sort

Given array S to be sorted
- If size of S < 1 then done;
- Pick any element v in S as the pivot
- Partition S-{v} (remaining elements in S) into two groups
- S1 = {all elements in S-{v} that are smaller than v}
- S2 = {all elements in S-{v} that are larger than v}
- Return {quicksort(S1) followed by v followed by quicksort(S2) }
- **Trick lies in handling the partitioning (step 3).**
- Picking a good pivot
- Efficiently partitioning in-place



Difference-Between-Quicksort-and-Merge-Sort_Figure-1.webp

Picking the Pivot

- Partition takes O(N) time.
- **Middle**
  - T(N) = 2 T(N/2) + N, T(N) = O(N log N) – same as the merge sort
- **Min/Max**
  - T(N) = T(N-1) + N, same as insertion sort, T(N) = O(??)
- **Strategy 1: Pick the first element in S**
  - works *only* if input is random
  - Quicksort is recursive, so sub-problems could be sorted
- **Strategy 2: Pick the pivot randomly**
  - Expensive operation
  - Usually works well

- works well with mostly sorted
- **Strategy 3: Median-of-three Partitioning**
  - Ideally, the pivot should be the *median* of input array S
  - divide the input into two almost equal partitions
  - Pivot = median of the left-most, right-most and center element
  - Solves the problem of sorted input

Example

• Example: Median-of-three Partitioning
– Let input S = {6, 1, 4, 9, 0, 3, 5, 2, 7, 8}
– left=0 and S[left] = 6
– right=9 and S[right] = 8
– center = (left+right)/2 = 4 and S[center] = 0
– Pivot
• = Median of S[left], S[right], and S[center]
• = median of 6, 8, and 0
• = S[left] = 6

Partitioning Algorithm

```C++
While (i < j)
        Move i to the right till we find a number greater than pivot
        Move j to the left till we find a number smaller than pivot
        If (i < j) swap(S[i], S[j])
//(The effect is to push larger elements to the right and smaller elements to the left)
Swap the pivot with S[i]
```

When Dealing With Small Arrays

- Insertion sort in the best under ten items
- Quick sort is recursive so it may take a long time sorting small arrays
- Hybrid (Switch between insertion and quick sort depending on array size)

Code Examples

```C++
template <typename Comparable>
void quicksort( vector<Comparable> & a, int left, int right )
{
        if( left + 10 <= right )
        {
                const Comparable & pivot = median3( a, left, right );
                // Begin partitioning
                int i = left, j = right - 1;
                for( ; ; ) {
                        while( a[ ++i ] < pivot ) { }
                        while( pivot < a[ --j ] ) { }
                        if( i < j )
                                std::swap( a[ i ], a[ j ] );
                        else
                                break;
        }
        std::swap( a[ i ], a[ right - 1 ] ); // Restore pivot
        quicksort( a, left, i - 1 ); // Sort small elements
        quicksort( a, i + 1, right ); // Sort large elements
        }
```

```
        else // Do an insertion sort on the subarray
                insertionSort( a, left, right )
}
```

Runtime

- Worst Case: $O(n2)$
- Best Case: $O(n * logn)$
- Average Case: $O(n * logn)$

Linear Time Sorts

- Comparison based sorting requires Ω(NlogN) time in the worst case
- Linear sorting applies to special cases

Bucket Sort

- Input: $A_1, A_2, , \dots A_N$ of positive integers smaller than M
- Output: Sorted list of integers
- Algorithm:
  - Keep an array with counts of each occurrence of the data
  - Set each count to 0
  - Need to know your data range
- Complexity? O(N+M)
- What if the data has other fields than the key?
  - Modify the count array to be an array of buckets
  - Is the sorting stable in this case?
- good if M is the same order as N
- limitation: needs an O(M) space so M cannot be very large

Radix Sort

- What if we want to use the idea of Bucket sort to sort based on social security number
- We can use a count array of size 1000 and perform bucket sort 3 times to sort based on the social security number
  - Use bucket sort to sort from the last sig bits to the most sig bits
  - Sort based on the last three digits in the first pass
  - sort based on the middle three digits in the second pass
  - sort based on the first three digits in the third pass
- Because bucket sort is stable
-