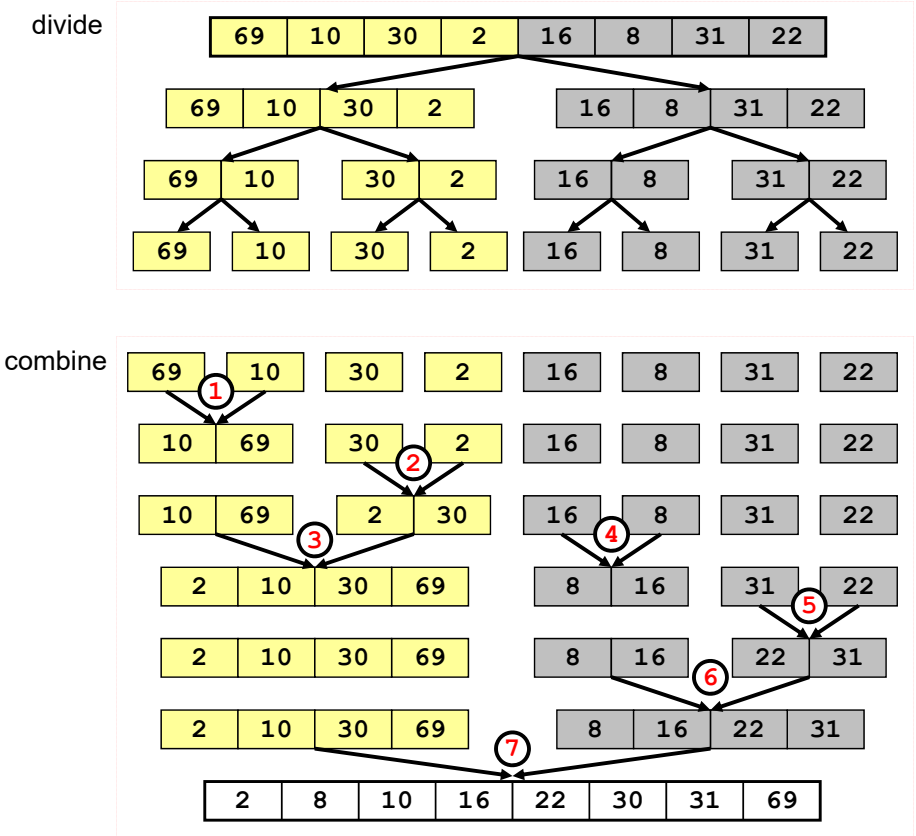
Sorting

Merge Sort

Idea: merge two sorted lists using temporary array.

- If the number of element is one, return;
- Otherwise, recursively merge sort the first half and the second half.
- The two sorted halves are merged together

Merge Sort


divide
69 10 30 2 16 8 31 22
69 10 30 2
16 8 31 22
69 10
30 2
16 8
31 22
69
10
30
2
16
8
31
22
combine
1
2
3
4
5
6
7
2 8 10 16 22 30 31 69

Merge Sort

```
void  MSort (ElementType A[], ElementType TmpArray[ ], int Left, int Right)
{
    int Center;
    if (Left < Right){
        Center = (Left + Right) / 2;
        MSort (A, TmpArray, Left, Center);
        MSort (A, TmpArray, Center+1, Right);
        Merge (A, TmpArray, Left, Center+1, Right);
    }
}
```

Merge Sort

```
void  Merge (ElementType A[], ElementType TmpArray[ ], int Lpos, int Rpos, int RightEnd)
{
    int i, LeftEnd, NumElements, TmpPos;
    LeftEnd = Rpos - 1;
    TmpPos = Lpos;
    NumElements = RightEnd - Lpos + 1;

    while (Lpos <= LeftEnd && Rpos <= RightEnd)
            if (A[Lpos] <= A[Rpos])
                TmpArray[TmpPos++] = A[Lpos++];
            else
                TmpArray[TmpPos++] = A[Rpos++];

    while (Lpos <= LeftEnd)
            TmpArray[TmpPos++] = A[Lpos++];
    while (Rpos <= RightEnd)
            TmpArray[TmpPos++] = A[Rpos++];

    for(i=0;   i<NumElements;   i++, RightEnd--)
            A[RightEnd] = TmpArray[RightEnd];
}
```

Merge Sort – Analysis

Divide and Conquer:

- Recursively Merge Sort the first half and the second half.
- Base case: when $n = 1$.
- Given two sorted halves, merge them.

Implementation: Each recursive call makes use of a temporary array.

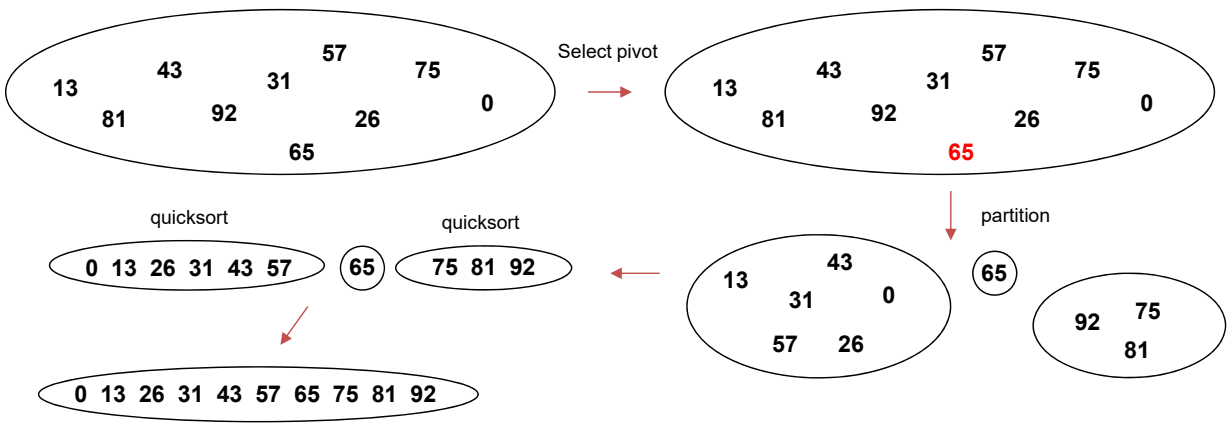
- Temporary array declared locally for each recursive call of Merge
- Dynamically allocate and free the minimum amount of temporary array.

Analysis: $O(n \log n)$ in the worst-case

- $T(1) = 1$
- $T(n) = 2T\left(\frac{n}{2}\right) + n$
- Solve the recurrence by 'telescoping' or 'substitution'.

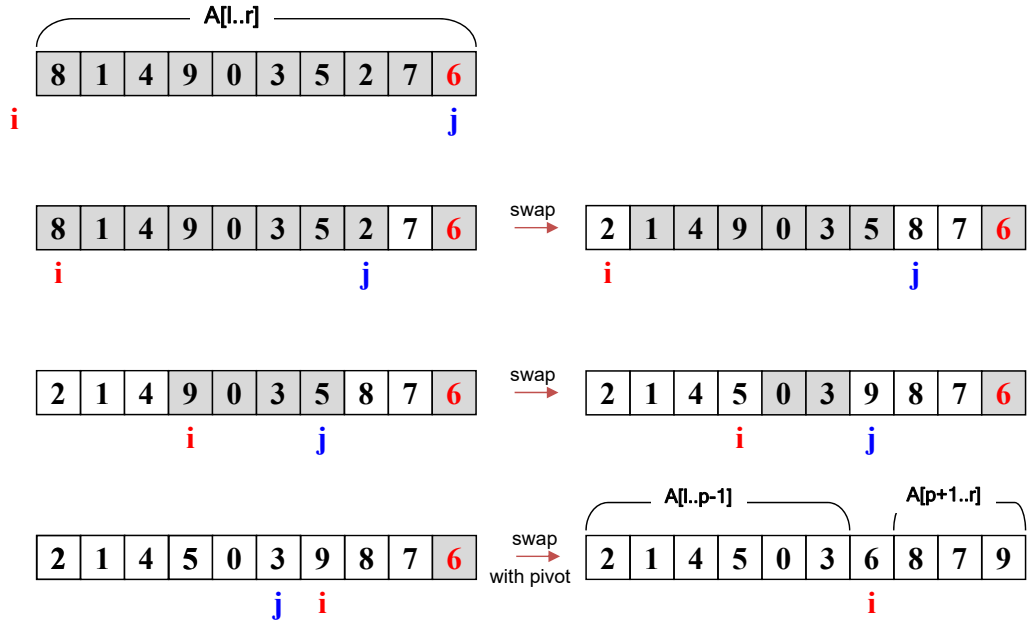
Quick Sort

- Divide-and-Conquer for sorting subarray A[l..r]
  - **Divide**: partition the array A[l..r] into two subarrays A[l..p-1] and A[p+1..r] such that all elements in A[l..p-1] are less than or equal to a pivot element A[p], which is, in turn, less than or equal to A[p+1..r].
  - **Conquer**: Sort the two subarrays A[l..p-1] and A[p+1..r] by recursive calls to quicksort.
  - **Combine**: Since the subarrays are sorted in place, no work is needed.



Shaded region: not yet partitioned, Unshaded region: partitioned

pivot = 6



Quick Sort

```
Quicksort(A, l, r)
{
  if (l >= r) return;
  p = Partition(A, l, r);
  Quicksort(A, l, p-1);
  Quicksort(A, p+1, r);
}

Partition(A, l, r)
{
  pivot = select_pivot(A, l, r);
  i = l - 1,  j = r;
  for( ; ; ) {
    while( A[--j] > pivot );
    while( A[++i] < pivot );
    if ( i < j ) swap(&A[i], &A[j]);
    else {
      swap(&A[i], &A[r]);
      return i;
    }
  }
}
```

Quick Sort – picking the pivot

- Use the first element → worst if the input is presorted or in reverse order
- Choose the pivot randomly → safe, but does not reduce the average running time while random number generation is costly.
- Median-of-Three → randomness does not help here, either. Choose the median of the leftmost, rightmost, and center elements;
  - What happens with the presorted input?
  - Experiment shows that the running time is reduced by about 5 percent.

Quick Sort – picking the pivot

$T(n) = T(i) + T(n - i - 1) + n \qquad (T(0) = T(1) = 0)$

Performance depends on the selection of pivot

worst-case partitioning: divide $n - 1$ and pivot

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

best-case partitioning: divide $n/2$ and $n/2$ elements

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2T(n/4) + 2n \\
&= \ldots \\
&= O(n \log n)
\end{aligned}
$$

average-case partitioning:

- Assume that the size of a partition is equally likely (that is, probability is $1/n$)
- The average value of $T(i)$ or $T(n-i-1)$ is $\frac{1}{n}\sum_{j=0}^{n-1} T(j)$
- $T(n) = \frac{2}{n}[\sum_{j=0}^{n-1} T(j)] + n$
- We already know $T(n) = O(n \log n)$ from the average case analysis of unbalanced binary search tree.

This average performance requires good selection of pivot!

- Median-of-Three Partitioning: take median of the left, right and center elements in A[l..r].

$$T(n) = \frac{2}{n}[\sum_{j=0}^{n-1} T(j)\ ] + n$$

$$nT(n) = 2[\sum_{j=0}^{n-1} T(j)\ ] + n^2$$

$$(n-1)T(n-1) = 2[\sum_{j=0}^{n-2} T(j)\ ] + (n-1)^2$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - 1$$

$$nT(n) = (n+1)T(n-1) + 2n$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1}$$

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2\sum_{i=3}^{n+1} \frac{1}{i}$$

$$\frac{T(n)}{n+1} = O(\log n),\ \ T(n) = O(n \log n)$$

Comparison of Different Sorting Algorithms

(unit: seconds)

n	Insertion Sort $O(n^2)$	Shellsort $O(n^{7/6})$(?)	Heapsort $O(n\log n)$	Quicksort $O(n\log n)$	Quicksort (opt.) $O(n\log n)$
10	0.00044	0.00041	0.00057	0.00052	0.00046
100	0.00675	0.00171	0.00420	0.00284	0.00244
1000	0.59564	0.02927	0.05565	0.03153	0.02587
10000	58.864	0.42998	0.71650	0.36765	0.31532
100000	NA	5.7298	8.8591	4.2298	3.5882
1000000	NA	71.164	104.68	47.065	41.282

* QuickSort (optimized): Cutoff used for small size inputs, median-of-three partitioning

http://www.sorting-algorithms.com/