### By Adaptability

Few sorting algorithms complexity changes based on presortedness [quick sort]: presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

## 10.4 Other Classifications

Other way of classifying the sorting algorithms are:

- Internal Sort
- External Sort

### Internal Sort

Sort algorithms which use main memory exclusively during the sort are called *internal* sorting algorithms. This kind of algorithms assumes high-speed random access to all memory.

### External Sort

Sorting algorithms which uses external memory, such as tape or disk, during the sort comes under this category.

## 10.5 Bubble sort

Bubble sort is the simplest sorting algorithm. It works by iterating the input array from the first element to last, comparing each pair of elements and swapping them if needed. Bubble sort continues its iterations until no swaps are needed. The algorithm got its name from the way smaller elements "*bubble*" to the top of the list. Generally, insertion sort has better performance than bubble sort. Some researchers suggest that we should not teach bubble sort because of its simplicity and bad complexity.

The only significant advantage that bubble sort has over other implementations is that it can detect whether the input list is already sorted or not.

### Implementation

```
void BubbleSort(int A[], int n) {
	for (int pass = n - 1; pass >= 0; pass--)	{
		for (int i = 0; i < pass - 1 ; i++)	{
			if(A[i] > A[i+1]) {
				// swap elements
				int temp = x[i];
				A[i] = A[i+1];
				A[i+1] = temp;
			}
		}
	}
}
```

Algorithm takes O($n^2$) (even in best case). We can improve it by using one extra flag. When there are no more swaps, indicates the completion of sorting. If the list is already sorted, by using this flag we can skip the remaining passes.

```
void BubbleSortImproved(int A[], int n) {
	int pass, i, temp, swapped = 1;
	for (pass = n - 1; pass >= 0 && swapped; pass--) {
		swapped = 0;
```

```
            for (i = 0; i < pass - 1 ; i++) {
                    if(A[i] > A[i+1]) {
                            // swap elements
                            temp = A[i];
                            A[i] = A[i+1];
                            A[i+1] = temp;
                            swapped = 1;
                    }
            }
        }
}
```

This modified version improves the best case of bubble sort to $O(n)$.

## Performance

| Worst case complexity : $O(n^2)$ |
|---|
| Best case complexity (Improved version) : $O(n)$ |
| Average case complexity (Basic version) : $O(n^2)$ |
| Worst case space complexity : $O(1)$ auxiliary |

# 10.6 Selection Sort

Selection sort is an in-place sorting algorithm. Selection sort works well for small files. It is used for sorting the files with very large values and small keys. This is because of the fact that selection is made based on keys and swaps are made only when required.

### Advantages:

- Easy to implement
- In-place sort (requires no additional storage space)

### Disadvantages:

- Doesn't scale well: $O(n^2)$

## Algorithm

1. Find the minimum value in the list
2. Swap it with the value in the current position
3. Repeat this process for all the elements until the entire array is sorted

This algorithm is called *selection sort* since it repeatedly *selects* the smallest element.

## Implementation

```
void Selection(int A [], int n) {
        int i, j, min, temp;
        for (i = 0; i < n - 1; i++) {
                min = i;
                for (j = i+1; j < n; j++) {
                        if(A [j] < A [min])
                                min = j;
                }
                // swap elements
                temp = A[min];
                A[min] = A[i];
```

```
                A[i] = temp;
        }
}
```

## Performance

| Worst case complexity : $O(n^2)$ |
|---|
| Best case complexity : $O(n)$ |
| Average case complexity : $O(n^2)$ |
| Worst case space complexity: O(1) auxiliary |

## 10.7 Insertion sort

Insertion sort is a simple and efficient comparison sort. In this algorithm each iteration removes an element from the input data and inserts it into the correct position in the list being sorted. The choice of the element being removed from the input is random and this process is repeated until all input elements have been gone through.

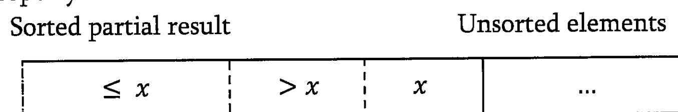
### Advantages

- Simple implementation
- Efficient for small data
- Adaptive: If the input list is presorted [may not be completely] then insertions sort takes $O(n + d)$, where $d$ is the number of inversions
- Practically more efficient than selection and bubble sorts even though all of them have $O(n^2)$ worst case complexity
- Stable: Maintains relative order of input data if the keys are same
- In-place: It requires only a constant amount O(1) of additional memory space
- Online: Insertion sort can sort the list as it receives it

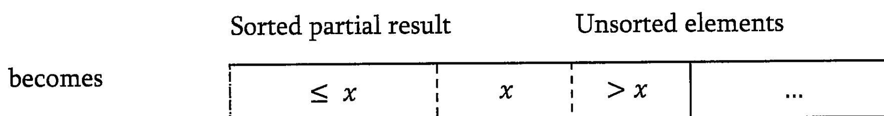
### Algorithm

Every repetition of insertion sort removes an element from the input data, inserts it into the correct position in the already-sorted list until no input elements remain. Sorting is typically done in-place. The resulting array after $k$ iterations has the property where the first $k + 1$ entries are sorted.

Sorted partial result Unsorted elements




Sorted partial result Unsorted elements




Each element greater than $x$ copied to the right as it is compared against $x$.

### Implementation

```
void InsertionSort(int a[], int n) {
        int i, j; int v;
        for (i = 2; i <= n - 1; i++) {
                v = A[i];
                j = i;
                while (A[j-1] > v && j >= 1) {
                        A[j] = A[j-1];
```

```
                j--;
            }
            A[j] = v;
        }
}
```

## Example

Given an array: 6 8 1 4 5 3 7 2 and the goal is to put them in ascending order.

6 8 1 4 5 3 7 2 (Consider index 0)
6 8 1 4 5 3 7 2 (Consider indices 0 - 1)
1 6 8 4 5 3 7 2 (Consider indices 0 - 2: insertion places 1 in front of 6 and 8)
1 4 6 8 5 3 7 2 (Process same as above is repeated until array is sorted)
1 4 5 6 8 3 7 2
1 3 4 5 6 7 8 2
1 2 3 4 5 6 7 8 (The array is sorted!)

## Analysis

### Worst case analysis

Worst case occurs when for every $i$ the inner loop has to move all elements $A[1], \ldots, A[i-1]$ (which happens when $A[i]$ = key is smaller than all of them), that takes $\Theta(i-1)$ time.

$$T(n) = \Theta(1) + \Theta(2) + \Theta(2) + \ldots\ldots + \Theta(n-1)$$
$$= \Theta(1 + 2 + 3 + \ldots\ldots + n - 1) = \Theta(\frac{n(n-1)}{2}) \approx \Theta(n^2)$$

### Average case analysis

For the average case, the inner loop will insert $A[i]$ in the middle of $A[1], \ldots, A[i-1]$. This takes $\Theta(i/2)$ time.

$$T(n) = \sum_{i=1}^{n} \Theta(i/2) \approx \Theta(n^2)$$

## Performance

| Worst case complexity : $O(n^2)$ |
|---|
| Best case complexity : $O(n^2)$ |
| Average case complexity : $O(n^2)$ |
| Worst case space complexity: $O(n^2)$ total, $O(1)$ auxiliary |

## Comparisons to Other Sorting Algorithms

Insertion sort is one of the elementary sorting algorithms with $O(n^2)$ worst-case time. Insertion sort is used when the data is nearly sorted (due to its adaptiveness) or when the input size is small (due to its low overhead). For these reasons and due to its stability, insertion sort is used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort.

**Note:**

- Bubble sort takes $\frac{n^2}{2}$ comparisons and $\frac{n^2}{2}$ swaps (inversions) in both average case and in worst case.
- Selection sort takes $\frac{n^2}{2}$ comparisons and $n$ swaps.
- Insertion sort takes $\frac{n^2}{4}$ comparisons and $\frac{n^2}{8}$ swaps in average case and in the worst case they are double.
- Insertion sort is almost linear for partially sorted input.
- Selection sort is best suits for elements with bigger values and small keys.

## 10.8 Shell sort

*Shell sort* (also called *diminishing increment sort*) is invented by *Donald Shell*. This sorting algorithm is a generalization of insertion sort. Insertion sort works efficiently on input that is already almost sorted.

In insertion sort, comparison happens between the adjacent elements. At most 1 inversion is eliminated for each comparison done with insertion sort. The variation used in shell sort is to avoid comparing adjacent elements until the last step of the algorithm. So, the last step of shell sort is effectively the insertion sort algorithm. It improves insertion sort by allowing the comparison and exchange of elements that are far away. This is the first algorithm which got less than quadratic complexity among comparison sort algorithms.

Shellsort uses a sequence $h1, h2, \ldots, ht$ called the *increment sequence*. Any increment sequence is fine as long as $h1 = 1$ and some choices are better than others. Shellsort makes multiple passes through input list and sorts a number of equally sized sets using the insertion sort. Shellsort improves on the efficiency of insertion sort by *quickly* shifting values to their destination.

### Implementation

```
void ShellSort(int A[], int array_size) {
	int i, j, h, v;
	for (h = 1; h = array_size/9; h = 3*h+1);
	for ( ; h > 0; h = h/3) {
		for (i = h+1; i = array_size; i += 1) {
			v = A[i];
			j = i;
			while (j > h && A[j-h] > v) {
				A[j] = A[j-h];
				j -= h;
			}
			A[j] = v;
		}
	}
}
```

Note that when $h == 1$, the algorithm makes a pass over the entire list, comparing adjacent elements, but doing very few element exchanges. For $h == 1$, shell sort works just like insertion sort, except the number of inversions that have to be eliminated is greatly reduced by the previous steps of the algorithm with $h > 1$.

### Analysis

Shell sort is efficient for medium size lists. For bigger lists, the algorithm is not the best choice. Fastest of all $O(n^2)$ sorting algorithms.

Disadvantage of Shell sort is that: it is a complex algorithm and not nearly as efficient as the merge, heap, and quick sorts. Shell sort is still significantly slower than the merge, heap, and quick sorts, but it is relatively simple algorithm which makes it a good choice for sorting lists of less than 5000 items unless speed important. It is also a good choice for repetitive sorting of smaller lists.

The best case in the Shell sort is when the array is already sorted in the right order. The number of comparisons is less. Running time of Shell sort depends on the choice of increment sequence.

### Performance

| Worst case complexity depends on gap sequence. Best known: $O(nlog^2n)$ |
|---|
| Best case complexity: $O(n)$ |

| Average case complexity depends on gap sequence |
| --- |
| Worst case space complexity: $O(n)$ |

# 10.9 Merge sort

Merge sort is an example of the divide and conquer.

## Important Notes

- *Merging* is the process of combining two sorted files to make one bigger sorted file.
- *Selection* is the process of dividing a file into two parts: $k$ smallest elements and $n - k$ largest elements.
- Selection and merging are opposite operations
    - selection splits a list into two lists
    - merging joins two files to make one file
- Merge sort is Quick sorts complement
- Merge sort accesses the data in a sequential manner
- This algorithm is used for sorting a linked list
- Merge sort is insensitive to the initial order of its input
- In Quick sort most of the work is done before the recursive calls. Quick sort starts with the largest subfile and finishes up with the small ones and as a result it needs stack and also this algorithm is not stable. Whereas Merge sort divides the list into two parts, then each part is conquered individually. Merge sort starts with the small subfiles and finishes up with the largest one and as a result it doesn't need stack and this algorithm is stable.

## Implementation

```
void Mergesort(int A[], int temp[], int left, int right) {
        int mid;
        if(right > left) {
                mid = (right + left) / 2;
                Mergesort(A, temp, left, mid);
                Mergesort(A, temp, mid+1, right);
                Merge(A, temp, left, mid+1, right);
        }
}
void Merge(int A[], int temp[], int left, int mid, int right) {
        int i, left_end, size, temp_pos;
        left_end = mid - 1;
        temp_pos = left;
        size = right - left + 1;
        while ((left <= left_end) && (mid <= right)) {
                if(A[left] <= A[mid]) {
                        temp[temp_pos] = A[left];
                        temp_pos = temp_pos + 1;
                        left = left +1;
                }
                else {  temp[temp_pos] = A[mid];
                        temp_pos = temp_pos + 1;
                        mid = mid + 1;
                }
        }
```

```
        while (left <= left_end) {
                temp[temp_pos] = A[left];
                left = left + 1;
                temp_pos = temp_pos + 1;
        }
        while (mid <= right) {
                temp[temp_pos] = A[mid];
                mid = mid + 1;
                temp_pos = temp_pos + 1;
        }
        for (i = 0; i <= size; i++) {
                A[right] = temp[right];
                right = right - 1;
        }
}
```

### Analysis

In Merge sort the input list is divided into two parts and solve them recursively. After solving the sub problems merge them by scanning the resultant sub problems. Let us assume $T(n)$ is the complexity of Merge sort with $n$ elements. The recurrence for the Merge Sort can be defined as:

Recurrence for Mergesort is $T(n) = 2T(\frac{n}{2}) + \Theta(n)$.

Using Master theorem, we get, $T(n) = \Theta(nlogn)$.

**Note**: For more details, refer *Divide and Conquer* chapter.

### Performance

| Worst case complexity : $\Theta(nlogn)$ |
|---|
| Best case complexity : $\Theta(nlogn)$ |
| Average case complexity : $\Theta(nlogn)$ |
| Worst case space complexity: $\Theta(n)$ auxiliary |

## 10.10 Heapsort

Heapsort is a comparison-based sorting algorithm and is part of the selection sort family. Although somewhat slower in practice on most machines than a good implementation of Quick sort, it has the advantage of a more favorable worst-case $\Theta(nlogn)$ runtime. Heapsort is an in-place algorithm but is not a stable sort.

### Performance

| Worst case performance: $\Theta(nlogn)$ |
|---|
| Best case performance: $\Theta(nlogn)$ |
| Average case performance: $\Theta(nlogn)$ |
| Worst case space complexity: $\Theta(n)$ total, $\Theta(1)$ auxiliary |

For other details on Heapsort refer *Priority Queues* chapter.

## 10.11 Quicksort

The quick sort is an example for divide-and-conquer algorithmic technique. It is also called *partition exchange sort*. It uses recursive calls for sorting the elements. It is one of famous algorithm among comparison based sorting algorithms.

Divide: The array $A[low \ldots high]$ is partitioned into two non-empty sub arrays $A[low \ldots q]$ and $A[q+1 \ldots high]$, such that each element of $A[low \ldots high]$ is less than or equal to each element of $A[q+1 \ldots high]$. The index $q$ is computed as part of this partitioning procedure.

Conquer: The two sub arrays $A[low \ldots q]$ and $A[q+1 \ldots high]$ are sorted by recursive calls to Quick sort.

## Algorithm

The recursive algorithm consists of four steps:

1) If there is one or no elements in the array to be sorted, return.
2) Pick an element in the array to serve as "*pivot*" point. (Usually the left-most element in the array is used.)
3) Split the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot.
4) Recursively repeat the algorithm for both halves of the original array.

## Implementation

```
Quicksort( int A[], int low, int high ) {
        int pivot;
        /* Termination condition! */
        if( high > low ) {
                pivot = Partition( A, low, high );
                Quicksort( A, low, pivot-1 );
                Quicksort( A, pivot+1, high );
        }
}
int Partition( int A, int low, int high ) {
        int left, right, pivot_item = A[low];
        left = low;
        right = high;
        while ( left < right ) {
                /* Move left while item < pivot */
                while( A[left] <= pivot_item )
                        left++;
                /* Move right while item > pivot */
                while( A[right] > pivot_item )
                        right--;
                if( left < right )
                        swap(A,left,right);
        }
        /* right is final position for the pivot */
        A[low] = A[right];
        A[right] = pivot_item;
        return right;
}
```

## Analysis

Let us assume that T(n) be the complexity of Quick sort and also assume that all elements are distinct. Recurrence for $T(n)$ depends on two subproblem sizes which depend on partition element. If pivot is $i^{th}$ smallest element then exactly $(i - 1)$ items will be in left part and $(n - i)$ in right part. Let us call it as $i$ –split. Since each element has equal probability of selecting it as pivot the probability of selecting $i^{th}$ element is $\frac{1}{n}$.

**Best Case:** Each partition splits array in halves and gives

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(nlogn),$$ [using *Divide and Conquer* master theorem]

**Worst Case:** Each partition gives unbalanced splits and we get

$$T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$$ [*using Subtraction and Conquer master theorem*]

The worst-case occurs when the list is already sorted and last element chosen as pivot.

**Average Case:** In the average case of Quick sort, we do not know where the split happens. For this reason, we take all possible values of split locations, add all of their complexities and divide with $n$ to get the average case complexity.

$$T(n) = \sum_{i=1}^{n} \frac{1}{n}(runtime\ with\ i - split) + n + 1$$

$$= \frac{1}{n}\sum_{i=1}^{N}\big(T(i-1) + T(n-i)\big) + n + 1$$

//since we are dealing with best case we can assume $T(n-i)$ and $T(i-1)$ are equal

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

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

Multiply both sides by $n$.

$$n\mathrm{T}(n) = 2\sum_{i=0}^{n-1} T(i) + n^2 + n$$

Same formula for $n - 1$.

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

Subtract the $n-1$ formula from $n$.

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

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

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

Divide with $n(n+1)$.

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

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

.

.

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

$$= O(1) + O(2logn)$$

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

$$T(n) = O\big((n+1)\ logn\big) = O(nlogn)$$

Time Complexity, $T(n) = O(nlogn)$.

### Performance

| Worst case Complexity: $O(n^2)$ |
|---|
| Best case Complexity: $O(nlogn)$ |
| Average case Complexity: $O(nlogn)$ |
| Worst case space Complexity: $O(1)$ |

### Randomized Quick sort

In average-case behavior of Quicksort, we assumed that all permutation of the input numbers are equally likely. However, we cannot always expect it to hold. We can add randomization to an algorithm in order to reduce the probability of getting worst case in Quick sort.

There are two ways of adding randomization in Quick sort: either by randomly placing the input data in the array or by randomly choosing an element in the input data for pivot. The second choice is easier to analyze and implement. The change will only be done at the Partition algorithm.

In normal Quicksort, *pivot* element was always the leftmost element in the list to be sorted. Instead of always using $A[low]$ as the *pivot* we will use a randomly chosen element from the subarray $A[low..high]$ in the randomized version of Quicksort. It is done by exchanging element $A[low]$ with an element chosen at random from $A[low..high]$. This ensures that the *pivot* element is equally likely to be any of the $high - low + 1$ elements in the subarray. Since the pivot element is randomly chosen, we can expect the split of the input array to be reasonably well balanced on average. This can help in preventing the worst-case behavior of quick sort which in unbalanced partitioning occurs.

Even though, the randomized version improves the worst case complexity, its worst case complexity is still $O(n^2)$. One way to improve the $Randomized - QuickSort$ is to choose the pivot for partitioning more carefully than by picking a random element from the array. One common approach is to choose the pivot as the median of a set of 3 elements randomly selected from the array.

## 10.12 Tree Sort

Tree sort uses a binary search tree. It involves scanning each element of the input and placing it into its proper position in a binary search tree. This has two phases:

- First phase is creating a binary search tree using the given array elements.
- Second phase is traverse the given binary search tree in inorder, thus resulting in a sorted array.

### Performance

The average number of comparisons for this method is $O(nlogn)$. But in worst case, number of comparisons is reduced by $O(n^2)$, a case which arises when the sort tree is skew tree.

## 10.13 Comparison of Sorting Algorithms

| Name | Average Case | Worst Case | Auxiliary Memory | Is Stable? | Other Notes |
|---|---|---|---|---|---|
| Bubble | $O(n^2)$ | $O(n^2)$ | 1 | yes | Small code |
| Selection | $O(n^2)$ | $O(n^2)$ | 1 | no | Stability depends on the implementation. |
| Insertion | $O(n^2)$ | $O(n^2)$ | 1 | yes | Average case is also $O(n + d)$, where $d$ is the number of inversions |
| Shell | - | $O(nlog^2n)$ | 1 | no | |
| Merge | $O(nlogn)$ | $O(nlogn)$ | depends | yes | |
| Heap | $O(nlogn)$ | $O(nlogn)$ | 1 | no | |
| Quick Sort | $O(nlogn)$ | $O(n^2)$ | $O(logn)$ | depends | Can be implemented as a stable sort depending on how the pivot is handled. |