sorts

background:

- Sorting involves arranging a collection of items in order,
- Sorting speeds up subsequent searching
- Arranges data in a human-useful way
- Provides intermediate step for advanced algorithms

stability

- let *x* = a[i], *y* = a[j], key(x) == key(y)
- "precedes" = occurs earlier in the array (smaller index)
- if *x* precedes *y* in a, then *x* precedes *y* in a'

adaptively

- behaviour/performance of algorithm affected by data values
- i.e. best/average/worst case performance differs

links:

- https://www.toptal.com/developers/sorting-algorithms
- https://visualgo.net/en/sorting

In analysing sorting algorithms:

- *N* = number of items = hi-lo+1
- *C* = number of comparisons between items
- *S* = number of times items are swapped

Aim to minimise *C* and *S*

Cases to consider for initial order of items:

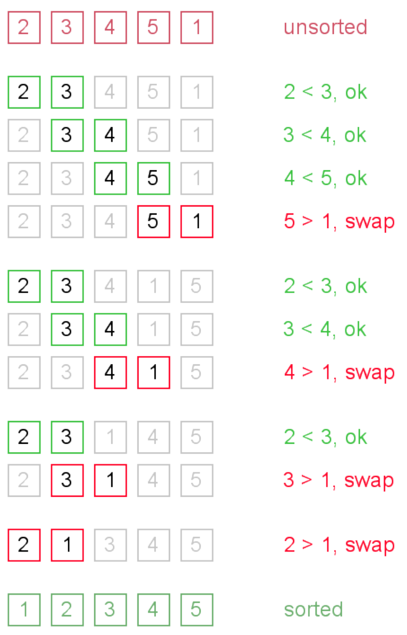
- random order: Items in a[lo..hi] have no ordering
- sorted order: a'[lo] ≤ a'[lo+1] ≤ ... ≤ a'[hi]
- revserse order: a'[lo] ≥ a'[lo+1] ≥ ... ≥ a'[hi]

```
Concrete framework:


typedef int Item;


#define key(A) (A)
#define less(A,B) (key(A) < key(B))
#define swap(A,B) {Item t; t = A; A = B; B =
t;}
#define swil(A,B) {if (less(A,B)) swap(A,B);}

// sorts a slice of an array of items
void sort(Item a[], int lo, int hi);

// checks sortedness (to validate functions
int isSorted(Item a[], int lo, int hi);
```

Oblivious bubble sort

Best case	Avg case	Worst case	stability
N sorted	N^2	N^2	N

unstable and unoptimised bubble sort, compares every adjacent values swop if out of order.



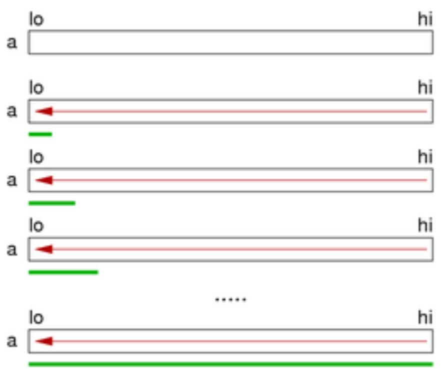
Bubble Sort With Early Exit

Best case	Avg case	Worst case	stability
N	N^2	N^2 reverse sorted input	Y

stable bubble sort that terminates when there have been no exchanges in one pass. Compares every adjacent values swop if out of order, however, if in a turn where no swops are made, then it will terminate after that turn.

Bubble sort:

- make multiple passes from *N* to *i* (*i=0..N-1*)
- on each pass, swap any out-of-order adjacent pairs
- elements move until they meet a smaller element
- eventually smallest element moves to *i* th position
- repeat until all elements have moved to appropriate position
- stop if there are no swaps during one pass (already sorted)



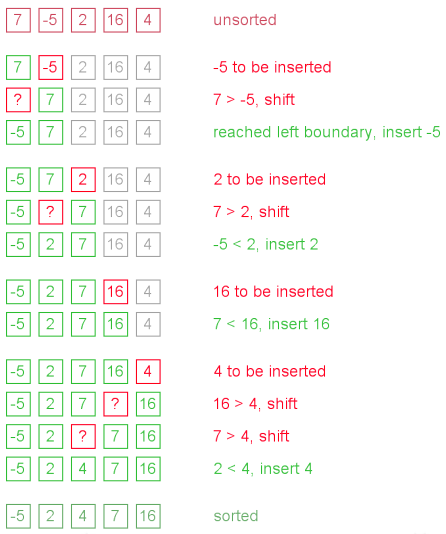
```
void bubbleSort(int a[], int lo, int hi)
{
   int i, j, nswaps;
   for (i = lo; i < hi; i++) {
      nswaps = 0;
      for (j = hi; j > i; j--) {
         if (less(a[j], a[j-1])) {
            swap(a[j], a[j-1]);
            nswaps++;
         }
      }
      if (nswaps == 0) break;
   }
}
```

Insertion sort

Best case	Avg case	Worst case	stability
N already sorted	N^2	N^2	Y

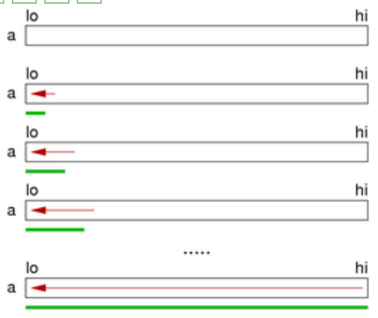
Take out one element, compare it with every subsequent element at the beginning (only). So the lower part of the sort is always sorted. Hence it inserts at the place where it needs.

- take first element and treat as sorted array (length 1)
- take next element and insert into sorted part of array so that order is preserved
- above increases length of sorted part by one
- repeat until whole array is sorted



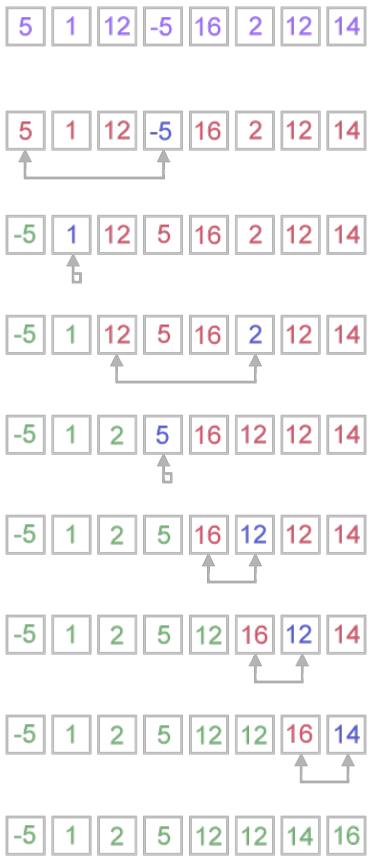
```
void insertionSort(int a[], int lo, int hi)
{
   int i, j, val;
   for (i = lo+1; i <= hi; i++) {
      val = a[i];
      for (j = i; j > lo; j--) {
         if (!less(val,a[j-1])) break;
         a[j] = a[j-1];
      }
      a[j] = val;
   }
}
```

```
void insertionSort(int a[], int lo, int hi)
{
   int i, j, val;
   for (i = hi; i > lo; i--)
      swil(a[i-1], a[i]);
   for (i = lo+2; i <= hi; i++) {
      val = a[i];
      for (j = i; less(val,a[j-1]); j--) {
         a[j] = a[j-1];
      }
      a[j] = val;
   }
}
```



Selection sort

Best case	*Avg case*	*Worst case*	*stability*
N^2	N^2	N^2	N



Checks out the “current” position data, and compare with every subsequent data (current to end) in the set, and if any data compared is smaller than current data, then it swops it. Hence, the power part of the sort is always sorted

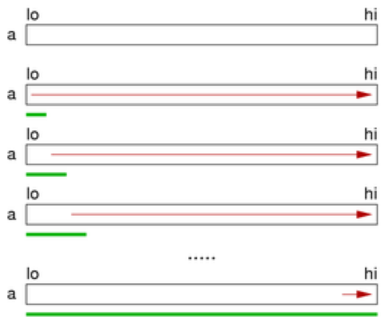
Simple, non-adaptive method:

- find the smallest element, put it into first array slot
- find second smallest element, put it into second array slot
- repeat until all elements are in correct position

"Put in x^th^ array slot" is accomplished by:

- swapping value in x^th^ position with x^th^ smallest value
-

Each iteration improves "sortedness" by one element



```
void selectionSort(int a[], int lo, int hi)
{
   int i, j, min;
   for (i = lo; i < hi-1; i++) {
      min = i;
      for (j = i+1; j <= hi; j++) {
         if (less(a[j],a[min])) min = j;
      }
      swap(a[i], a[min]);
   }
}
```

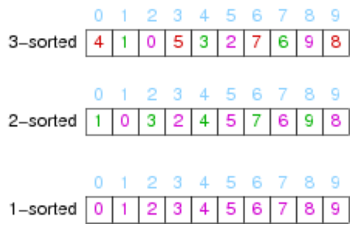
Shell sort: improving on insertion

Insertion sort:	Shellsort: basic idea
• based on exchanges that only involve adjacent items • already improved above by using moves rather than swaps • "long distance" moves may be more efficient	• array is *h*-sorted if taking every *h*'th element yields a sorted array • an *h*-sorted array is made up of *n/h* interleaved sorted arrays • Shellsort: *h*-sort array for progressively smaller *h*, ending with 1-sorted

Best case	*Avg case*	*Worst case*	*stability*
N	N^2	N^2	N

Shell sort Times 4

shell sort with intervals ..., 4096, 1024, 256, 64, 16, 4, 1. Shell sort times 4 uses a gap sequence of 1, 4, 16, 64, ... and has a quadratic worst case and linear best case.



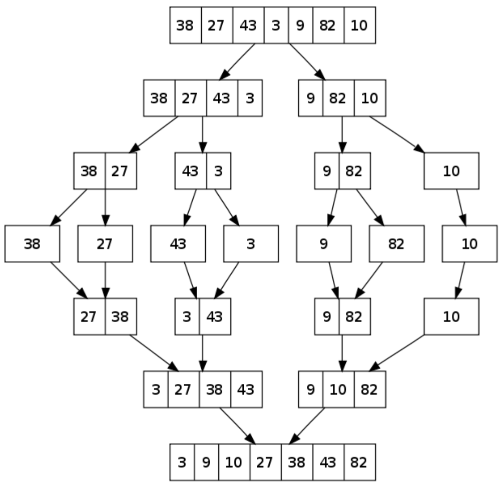
Shell sort Segwick-Like

shell sort with intervals ..., 4193, 1073, 281, 23, 8, 1

```
void shellSort(int a[], int lo, int hi)
{
   int hvals[8] = {701, 301, 132, 57, 23, 10, 4, 1};
   int g, h, start, i, j, val;

   for (g = 0; g < 8; g++) {
      h = hvals[g];
      start = lo + h;
      for (i = start; i < hi; i++) {
         val = a[i];
         for (j = i; j >= start && less(val,a[j-h]); j -= h)
            move(a, j, j-h);
         a[j] = val;
      }
   }
}
```

Merge sort

Best case	***Avg case***	***Worst case***	***stability***
nlogn	nlogn	nlogn	Y



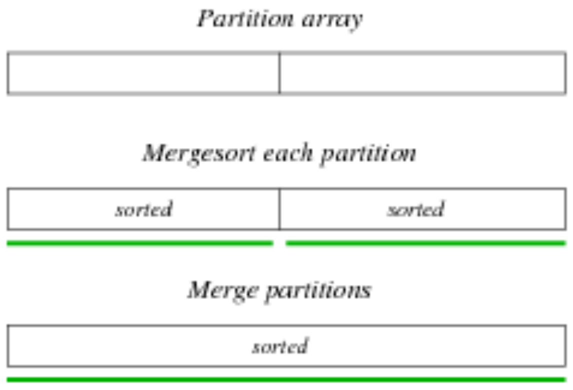
Merge sort breaks an array into two sub-arrays, recursively sorts these arrays and merges them back into a single sorted array. It has time complexity of n log n regardless of the input, and is stable.

Mergesort: basic idea

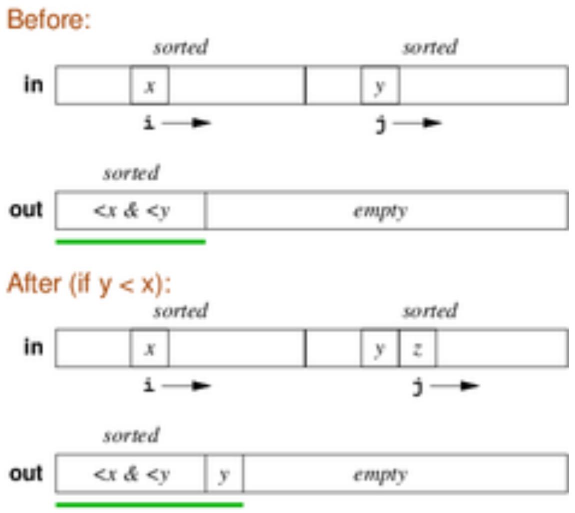
- split the array into two equal-sized paritions
- (recursively) sort each of the partitions
- merge the two partitions into a new sorted array
- copy back to original array

Merging: basic idea

- copy elements from the inputs one at a time
- give preference to the smaller of the two
- when one exhausted, copy the rest of the other



```
void mergesort(Item a[], int lo, int hi)
{
   int mid = (lo+hi)/2;
   if (hi <= lo) return;
   mergesort(a, lo, mid);
   mergesort(a, mid+1, hi);
   merge(a, lo, mid, hi);
}
```



Example of use (`typedef int Item`):

```
int nums[10] = {32,45,17,22,94,78,64,25,55,42};
mergesort(nums, 0, 9);
```

```
void merge(Item a[], int lo, int mid, int hi)
{
   int  i, j, k, nitems = hi-lo+1;
   Item *tmp = malloc(nitems*sizeof(Item));

   i = lo; j = mid+1; k = 0;

   while (i <= mid && j <= hi) {
     if (less(a[i],a[j]))
        tmp[k++] = a[i++];
     else
        tmp[k++] = a[j++];
   }

   while (i <= mid) tmp[k++] = a[i++];
   while (j <= hi) tmp[k++] = a[j++];


   for (i = lo, k = 0; i <= hi; i++, k++)
      a[i] = tmp[k];
   free(tmp);
}
```

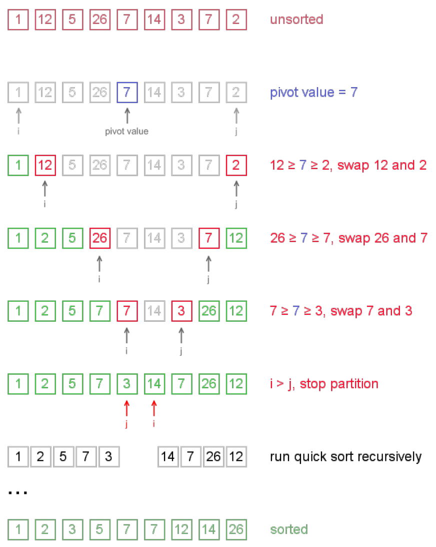
Non-recursive Mergesort

```
#define min(A,B)  ((A < B) ? A : B)

void mergesort(Item a[], int lo, int hi)
{
   int i, m;
   int end;
   for (m = 1; m <= lo-hi; m = 2*m)  {
      for (i = lo; i <= hi-m; i += 2*m) {
         end = min(i+2*m-1, hi);
         merge(a, i, i+m-1, end);
      }
   }
}
```

Quick sort

Best case	*Avg case*	*Worst case*	*stability*
nlogn	nlongn	N^2 sorted or reverse sorted	N

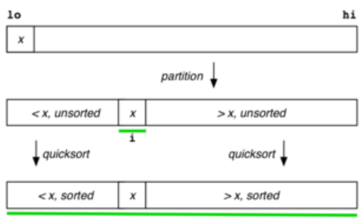
normal quick sort (pivot is the last element). Vanilla quick sort chooses the first element of an array as the pivot, and scans through the rest of the array, putting elements less than the pivot to the left and those greater than the pivot to the right. It then recursively sorts the two sub-arrays formed.



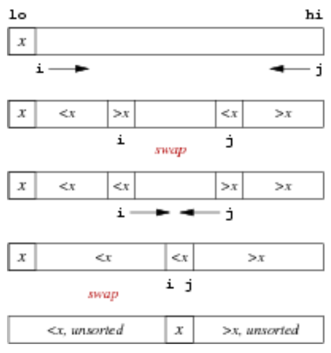
Quicksort: basic idea

- choose an item to be a "pivot"
- re-arrange (partition) the array so that
    - all elements to left of pivot are smaller than pivot
    - all elements to right of pivot are greater than pivot
- (recursively) sort each of the partitions

```
void quicksort(Item a[], int lo, int hi)
{
   int i;
   if (hi <= lo) return;
   i = partition(a, lo, hi);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```



```
int partition(Item a[], int lo, int hi)
{
   Item v = a[lo];
   int  i = lo+1, j = hi;
   for (;;) {
      while (less(a[i],v) && i < j) i++;
      while (less(v,a[j]) && j > i) j--;
      if (i == j) break;
      swap(a,i,j);
   }
   j = less(a[i],v) ? i : i-1;
   swap(a,lo,j);
   return j;
}
```



Quick Sort Median of Three

Best case	*Avg case*	*Worst case*	*stability*
nlogn	nlogn	N^2	N

pivot is the median of first last and middle elements
To improve upon quick sort, we choose a pivot that is the median of the first, middle and last items. This makes the quadratic worst case very unlikely.

```
void medianOfThree(Item a[], int lo, int hi)
{
   int mid = (lo+hi)/2;
   if (less(a[mid],a[lo])) swap(a, lo, mid);
   if (less(a[hi],a[mid])) swap(a, mid, hi);
   if (less(a[mid],a[lo])) swap(a, lo, mid);


   swap(a, mid, lo+1); ~~swap(a, lo, mid);~~
}
```

```
void quicksort(Item a[], int lo, int hi)
{
   int i;
   if (hi-lo < Threshhold) { ... return; }
   medianOfThree(a, lo, hi);
   i = partition(a, lo+1, hi-1);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

Randomised Quick Sort

Best case	***Avg case***	***Worst case***	***stability***
nlogn	nlogn	N^2	N

list is shuffled then vanilla quick-sorted
To further improve upon quick sort, we choose a random pivot. This makes the quadratic worst case almost impossible.

Non-recursive (stack) :

```
void quicksortStack (Item a[], int lo, int hi)
{
   int i;  Stack s = newStack();
   StackPush(s,hi); StackPush(s,lo);
   while (!StackEmpty(s)) {
      lo = StackPop(s);
      hi = StackPop(s);
      if (hi > lo) {
         i = partition (a,lo,hi);
         StackPush(s,hi); StackPush(s,i+1);
         StackPush(s,i-1); StackPush(s,lo);
      }
   }
}
```

Bogo sort

Bogo sort choose two random array elements, if out-of-order then swap, repeat	N	N.n!	Infinite	N	Bogo sort tests if the array is sorted, then exits if it is or else permutes it randomly and repeats the process. If input is perfectly sorted, it will execute in linear time, or else the expected time complexity is n.n!, but could be infinite in the worst case. It is obviously unstable.