Sorting

- Efficient **searching** (find) often depends on information being **sorted**
- **Sorting** and **searching** are major areas in CS and make use of a number of ADTs (sequences & trees)
- A sorted sequence is the most common example
  - dictionary
  - telephone directory
  - library catalogues
- Trees may also be sorted
  - BST / B-Tree

Sorting Algorithms

- Bubble Sort
- Insertion Sort
- Selection Sort
- Shellsort
- Merge Sort (>=2 streams)
- Quicksort
- Sort: S x R → S
  - R is often > or <

- Collections may be
  - unsorted and sorted by a given sort function
  - sorted from creation e.g. by modifying the add function
- Sorting may also be
  - internal (in memory)
  - external (disk / files)

Bubble Sort - O(n²)

swap(x, y) { t=x; x=y; y=t;}

elements “bubble” to the “top” of a “vertical array”

```
for ( i in 1 up to n-1)
    for ( j in n down to i+1)
        if A[j] < A[j-1]
            swap(A [j], A[j-1] )
```

i=1	i=2	i=3	i=4
j=6..2	j=6..3	j=6..4	j=6..5
123456	123456	123456	123456
pekavs	apeksv	aepksv	aekpsv
pekasv	apeksv	aepksv	aekpsv
pekasv	apeksv	aepksv	
peaksv	apeksv		
paeksv			
i=5 j=6	**sorted array**		
aekpsv	aekpsv		

Insertion Sort - O(n^2)

swap(x, y) { t=x; x=y; y=t;}
on the i^{th} pass, A[i] is inserted into the right place
A[0]=* where * < any value

```
for i = 2 up to n {
    j=i; while A[j] < A[j-1] {
        swap(A[j], A[j-1]);
        j=j-1;
        }
    }
```

Init 0123456 *pekavs	i=2 j=2.. 0123456 *pekavs *epkavs	i=3 j=3.. 0123456 *epkavs *ekpavs	i=4 j=4.. 0123456 *ekpavs *ekapvs *eakpvs *aekpvs
i=5 j=5.. j=6.. 0123456 *aekpvs	i=6 0123456 *aekpvs	 sorted array 0123456 *aekpsv__*aekpsv	

Selection Sort - O(n^2)

swap(x, y) { t=x; x=y; y=t;}

in ith pass, select lowest value and

swap with A[i]

for i = 1 up to n-1{

select min(A[i]..A[n]) → A[k];

swap(A[i], A[k]);

}

Init 123456 pekavs	i=1 k=4 123456 pekavs aekpvs	i=2 k=2 123456 aekpvs aekpvs	i=3 k=3 123456 aekpvs aekpvs
i=4 i=5 k=4 k=6 123456 aekpvs aekpvs	 sorted array 123456 aekpvs aekpsv	 123456 aekpsv	

Shellsort - $O(n^2)$

```
swap(x, y) { t=x; x=y; y=t;}
inc = size(A) div 2;
while inc > 0 {
 for i = inc + 1 up to n {
  j = i - inc;
   while j > 0 if A[j] > A[j+inc]
   {
     swap(A[j], A[j+inc];
     j=j-inc;
     } else j = 0;
  }
 inc = inc div 2;
 }
```

Init inc=3 pekavs	i=4 j=1,-2 pekavs aekpvs	i=5 j=2,0 aekpvs	i=6 j=3,0 aekpvs
inc=1	i=2 j=1,0 aekpvs	i=3 j=2,0 aekpvs	i=4 j=3,0 aekpvs
i=5 j=4 ,0 aekpvs	i=6 j=5,4,0 aekpvs	inc=0 aekpsv	sorted array aekpsv

Merge Sort - O(n log$_n$)

For a list L length n = 2^k

```
List: msort(List)
{
 if n=1 return L
 else {
     divide L into L1, L2
     length n/2;
     return
      merge (msort(L1),
                 msort(L2));
     }
  }
```

Init pekavsbz

peka vsbz
pe ka vs bz
p e k a v s b z

n=1 => merge phases start

p e k a v s b z
ep ak sv bz
aekp bsvz
abekpsvz

sorted array !

Quicksort - $O(n \log_n) / O(n^2)$

Quicksort has 4 steps

1) if size(S) = 0 or 1 return(S)
2) choose a pivot v in S **(with care)**
3) **partition** S to L, R such that

    all x in L <= v (the pivot value)

    all x in R > v (the pivot value)

4) **merge** (QS(L), v, QS(R))

Init   pekavs   size=6 pivot = p

L = eka   R = vs
size=3 pivot=e   size=2 pivot=v

L=a	R=k	L=s	R=¤
size=1	1	1	0

merge((a),e,(k))   merge((s),v,(¤))

merge((aek), p, (sv))
=> aekpsv