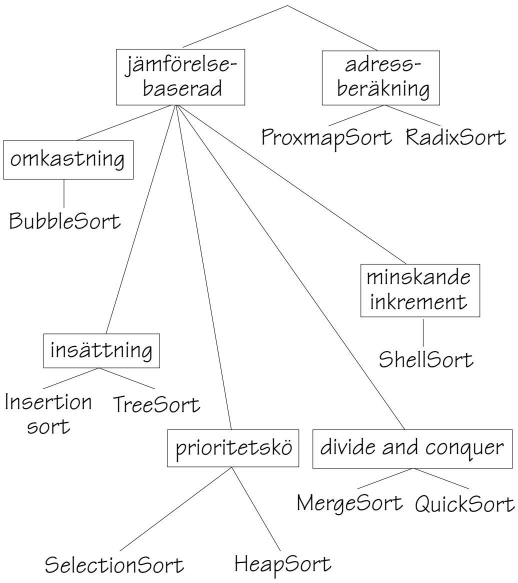
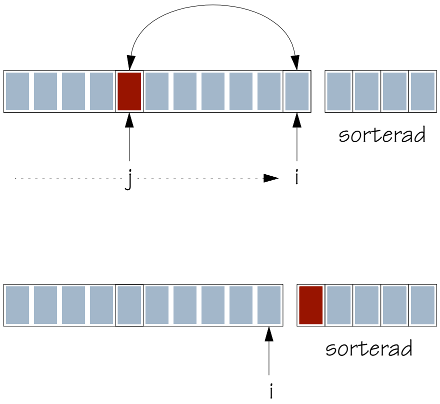
Sortering



Sortering med prioritetskö

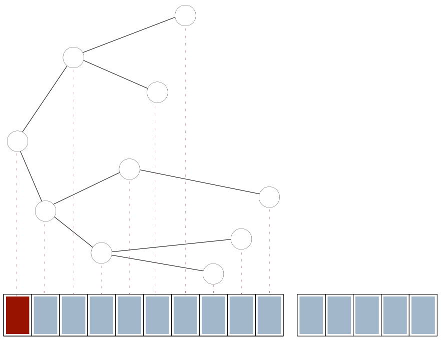


SelectionSort

```
void SelectionSort(KeyType[] A) {

  int      i, j, k;
  KeyType  temp;

  i = A.length - 1;
  while ( i > 0) {
    j = i;
    for (k = 0; k < i; k++) {
      if ( A[k].compareTo(A[j])  >  0 )  j = k ;
    }
    // Byt plats på A[i] och A[j]
    temp = A[i]; A[i] = A[j]; A[j] = temp;

    i--;
  }
}
```

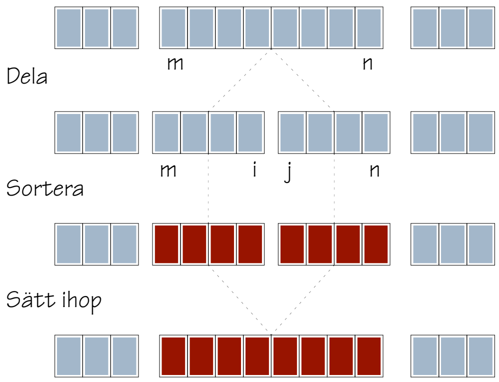
Tidskomplexitet: $O(n^2)$

HeapSort



Tidskomplexitet:
- Skapa heap från osorterad array: O(n)
- Delete: O(log n)
- HeapSort: O(n log n)

Divide-and-Conquer



MergeSort

```
void mergeSort(int[] A) {
  int[] B = new int[A.length];
  mergeSort(A, B, 0, A.length - 1);
}

void mergeSort(int[] A, int[] B, int m, int n) {
  if( m < n ) {
    int mid = (m + n)/2;
    mergeSort(A, B, m, mid);
    mergeSort(A, B, mid + 1, n);
    merge(A, B, m, mid + 1, n);
  }
}

void merge(int[] A, int[] B, int i, int j, int n) {
  // Delarray 1 = A[i:j-1]
  // Delarray 2 = A[j:n]
  int m = j - 1;
  int k = i;
  int l = i;

  while( i <= m ) {  // Fläta samman
    while( j <= n && A[j] < A[i]) {
      B[l] = A[j];
      l++;
      j++;
    }
    B[l] = A[i];
    l++;
    i++;
  }
  while( k < l ) {    // Kopiera tillbaka B[k:l]
    A[k] = B[k];
    k++;
  }
}
```

QuickSort

Problem: Sortera A[m:n]

Princip:

Om A[m:n] innehåller 0 eller 1 element så är vi klara

Annars

- Plocka bort ett element N ur A[m:n]
- Partitionera A[m:n] i två arrayer; A[m:i-1] med alla element mindre än N och A[i+1:n] med alla element större än eller lika med N;
- Sortera de två arrayerna
- A[i] = N

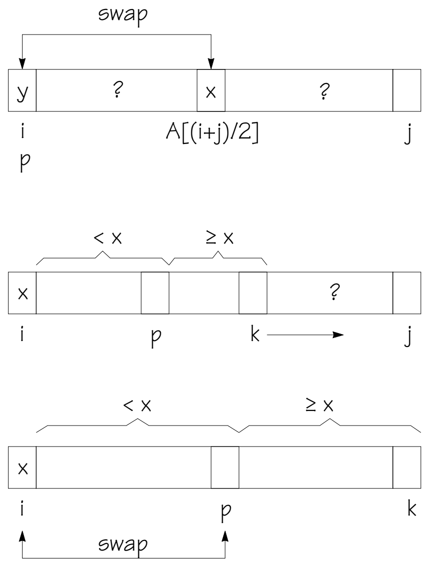
QuickSort

```
void QuickSort(KeyType[] A, int m, int n)  {
  if( m < n ) {
    int p = partition(A, m, n);
    QuickSort(A, m, p - 1);
    QuickSort(A, p + 1, n);
  }
}

int partition(KeyType[] A, int i, int j)  {

  KeyType  pivot, temp;
  int  k, middle, p;

  middle = ( i + j ) / 2 ;
  pivot = A[middle];
  A[middle] = A[i];
  A[i] = pivot;
  p = i;

  for( k = i+1; k <= j; k++ ) {
    if( A[k].compareTo(pivot) < 0 ) {
      p++;
      temp = A[p];
      A[p] = A[k];
      A[k] = temp;
    }
  }
  temp = A[i]; A[i] = A[p]; A[p] = temp;
  return p;
}
```

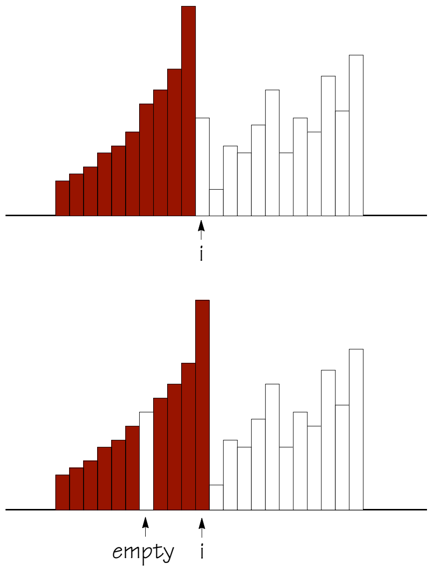
Värsta fallet: $O(n^2)$

Normalfallet: $O(n \log n)$

Partition



InsertionSort



InsertionSort

```
void InsertionSort(KeyType[] A) {

  int       i,empty;
  KeyType   K;
  boolean   notSorted;
  int       n = A.length;

  for (i = 1;  i < n;  i++)  {
    K = A[i];
    empty = i;
    notSorted = (A[empty-1].compareTo(K) > 0);
    while( notSorted ) {
      A[empty] = A[empty - 1];
      empty--;
      if( empty > 0 ) {
        notSorted = (A[empty-1].compareTo(K) > 0);
      } else {
        notSorted = false;
      }
    } // end while

    A[empty] = K;
  } // end for

}
```

Värsta fall: $O(n^2)$

Bästa fall: $O(n)$ (om redan sorterad)

BubbleSort

```
void BubbleSort(KeyType[] A)  {

  int  i;
  KeyType temp;
  boolean notSorted;
  int n = A.length;

  do {
    notSorted = false;
    for (i = 0;  i < n - 1;  i++) {
      if (A[i].compareTo(A[i+1]) >0) {
        temp = A[i];
        A[i] = A[i + 1];
        A[i + 1] =temp;
        notSorted = true;
      }
    }
  } while( notSorted );

}
```

Värsta fallet: $O(n^2)$

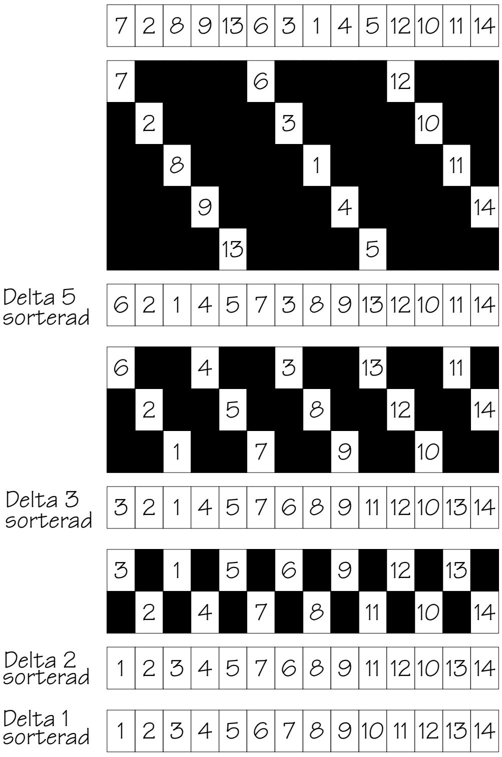
Normalfallet: $O(n^2)$

Bästa fallet: $O(n)$ (om redan sorterad)

ShellSort

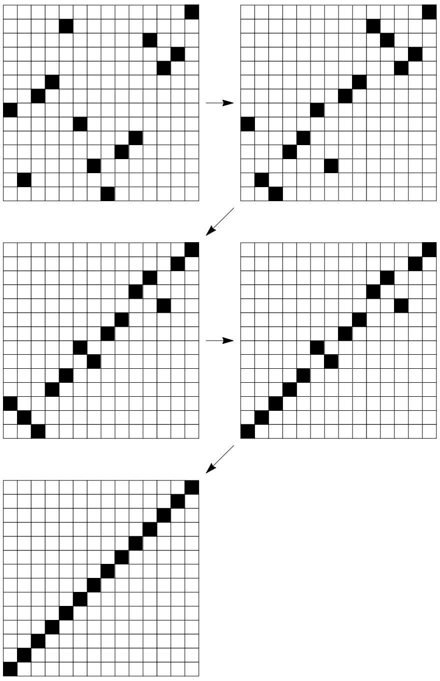
Princip:
- Grovsortera
- Sortera i mindre regioner
- osv till hela arrayen är sorterad

Exempel



	7	2	8	9	13	6	3	1	4	5	12	10	11	14
Delta 5 sorterad	6	2	1	4	5	7	3	8	9	13	12	10	11	14
Delta 3 sorterad	3	2	1	4	5	7	6	8	9	11	12	10	13	14
Delta 2 sorterad	1	2	3	4	5	7	6	8	9	11	12	10	13	14
Delta 1 sorterad	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Exempel (forts.)



ShellSort

```
void ShellSort(KeyType[] A) {

  int  i, di;

  di = A.length;

  do {
    di = 1 + di / 3;
    for( i = 0; i < di; i++ ) {
      deltaIsort(A, i, di);
    }
  } while( di > 1 );
}
```

ShellSort (forts.)

```
void deltaIsort(KeyType[] A, int i, int di){

  int       j, k;
  KeyType   keyToInsert;
  boolean   notSorted;
  int       n = A.length;

  j = i + di;

  while( j < n ) {
    keyToInsert = A[j];
    k = j;
    notSorted = true;
    do {
      if(A[k-di].compareTo(keyToInsert) <= 0) {
        notSorted = false;
      } else {
        A[k] = A[k - di];
        k = k - di;
        if( k == i ) notSorted = false;
      }
    } while( notSorted );

    A[k] = keyToInsert;
    j = j + di;
  }
}
```

Normalfallet: $O(n^{1,2})$