# Sorting
# — T.07 —

**Summary:**

- Coding conventions
- Bubble sort and shaker sort
- Insertion sort and Shell sort
- Quick sort
- Merge sort
- Heap sort
- Tree sort
- Other sorting routines (rank sort, selection sort)
- Computational complexity summary

**Highly recommended bibliography for this lecture:**

- **Algorithms**, Robert Sedgewick and Kevin Wayne, fourth edition, Addison Wesley, 2011

**Recommended bibliography for this lecture:**

- **Analysis of Algorithms**, Jeffrey J. McConnell, second edition, Jones and Bertlett Publishers, 2008.
- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **Estruturas de Dados e Algoritmos em C**, António Adrego da Rocha, terceira edição, FCA.

**Other useful stuff (on the web):**

- Visualization (animation) of how some algorithms work (see for example the animations in http://www.sorting-algorithms.com/.
- Another algorithm animation resource.
- Description and useful information about many sorting algorithms.

# Coding conventions

The sorting routines that will be presented in the next slides are all called with the same arguments. They all have three arguments (in this order):

`T *data` : a pointer to the **beginning** of the array where the data to be sorted is located.

Each element of the array is of type `T`. Because the `C` programming language does not have templates, in practice `T` will have to be replaced by the actual data type. (The data type has to allow comparisons.)

`int first` : the index of the **first** element of the array that will be sorted.

`int one_after_last` : **one plus** the index of the **last** element of the array that will be sorted.

So, **only** the elements `data[first]`, `data[first + 1]`, ..., `data[one_after_last - 1]` will actually be sorted by the function. All other elements will be left untouched. In other words, the index sorting interval will be $[\mathbf{first}, \mathbf{one\_after\_last}[$, which is closed on the left and open on the right. To sort the entire array set `first` to zero and `one_after_last` to the number of elements of the array.

This way of doing things, which is similar to how ranges work in the `Python` programming language, is quite convenient when splitting the array into smaller parts (say, to sort each one of them separately).

# Bubble sort and shaker sort (part 1a, bubble sort description)

The bubble sort algorithm, although very simple, is almost always worse that other sorting algorithms (such as the insertion sort algorithm to be explained later). It has, however, pedagogical interest. It compares adjacent array entries and exchanges them if they are nor ordered. It can be described as follows:

To sort `a[0]`,...,`a[n-1]` in increasing order do:

1. [Do pass.] For $i$ equal to $0, 1, \ldots, n-2$ do step 2. When that is finished, go to step 3.
2. [Compare and exchange.] If `a[i]` is larger than `a[i+1]` exchange them.
3. [Terminate or do it again.] Terminate the algorithm if no exchange was done in the last pass (steps 1 and 2). Otherwise, go to step 1 and do it all again.

If the array is already sorted, only one pass is necessary (to confirm that it is already sorted). If the array is sorted in decreasing order, $n$ passes are necessary (each pass moves the largest array element not yet in its place to its proper place). In general, the computational complexity of the best, average, and worst cases of the bubble sort algorithm are $O(n)$, $O(n^2)$, and $O(n^2)$, respectively.

# Bubble sort and shaker sort (part 1b, bubble sort code)

The following C code is one possible implementation of the bubble sort algorithm. (In the slides of this lecture, all sorting algorithms will have the same interface.)

```
void bubble_sort(T *data,int first,int one_after_last)
{ // sort data[first],...,data[one_after_last-1] in increasing order
  int i,i_low,i_high,i_last;

  i_low = first;
  i_high = one_after_last - 1;
  while(i_low < i_high)
  {
    for(i = i_last = i_low;i < i_high;i++)
      if(data[i] > data[i + 1])
      {
        T tmp = data[i];
        data[i] = data[i + 1];
        data[i + 1] = tmp;
        i_last = i;
      }
    i_high = i_last;
  }
}
```

To sort an array named `abc` with $10$ elements, just do `bubble_sort(abc,0,10)`.

# Bubble sort and shaker sort (part 2a, shaker sort description)

In the bubble sort algorithm a small entry near the end of the array may require many passes until it migrates to its final position. The so-called cocktail-shaker sort algorithm (it also only has pedagogical interest) attempts to ameliorate this problem by doing "up" and "down" passes. It can be described as follows:

To sort `a[0]`,...,`a[n-1]` in increasing order do:

1. [Do up pass.] For $i$ equal to $0, 1, \ldots, n-2$ do step 2. When that is finished, go to step 3.
2. [Compare and exchange.] If `a[i]` is larger than `a[i+1]` exchange them.
3. [Terminate or do down pass.] Terminate the algorithm if no exchange was done in the last up pass (steps 1 and 2). Otherwise, go to step 4.
4. [Do down pass.] For $i$ equal to $n-1, n-2, \ldots, 1$ do step 5. When that is finished, go to step 6.
5. [Compare and exchange.] If `a[i]` is smaller than `a[i-1]` exchange them.
6. [Terminate or do it again.] Terminate the algorithm if no exchange was done in the last down pass (steps 4 and 5). Otherwise, go to step 1 and do it all again.

Like bubble sort, the computational complexity of the best, average, and worst cases of the shaker sort algorithm are $O(n)$, $O(n^2)$, and $O(n^2)$, respectively.

# Bubble sort and shaker sort (part 2b, shaker sort code)

The following C code is one possible implementation of the shaker sort algorithm.

```
void shaker_sort(T *data,int first,int one_after_last)
{
  int i,i_low,i_high,i_last;

  i_low = first;
  i_high = one_after_last - 1;
  while(i_low < i_high)
  {
    // up pass
    for(i = i_last = i_low;i < i_high;i++)
      if(data[i] > data[i + 1])
      {
        T tmp = data[i];
        data[i] = data[i + 1];
        data[i + 1] = tmp;
        i_last = i;
      }
    i_high = i_last;
    // down pass
    for(i = i_last = i_high;i > i_low;i--)
      if(data[i] < data[i - 1])
      {
        T tmp = data[i];
        data[i] = data[i - 1];
        data[i - 1] = tmp;
        i_last = i;
      }
    i_low = i_last;
  }
}
```

# Insertion sort and Shell sort (part 1, insertion sort description and code)

Insertion sort, and its very simple improvement Shell sort, are very simple sorting algorithms that can be used in practice to sort small arrays (for larger arrays other sorting algorithms, with better asymptotic computational complexity, should be used). Insertion sort can be described as follows:

To sort `a[0]`,...,`a[n-1]` in increasing order do:

1. [Do pass.] For $i$ equal to $1, 2, \ldots, n-1$ do step 2. When that is finished, terminate the algorithm.
2. [Insert.] For `j` equal to $i, i-1, \ldots, 1$ and while `a[j]` is smaller than `a[j-1]` exchange `a[j]` with `a[j-1]`.

Like the bubble and shaker sort algorithms, the computational complexity of the best, average, and worst cases of the insertion sort algorithm are $O(n)$, $O(n^2)$, and $O(n^2)$, respectively. **However**, the multiplicative constants hidden behind the asymptotic notation are better (smaller) for the insertion sort algorithm.

The following C code is one possible implementation of the insertion sort algorithm.

```
void insertion_sort(T *data,int first,int one_after_last)
{
  int i,j;

  for(i = first + 1;i < one_after_last;i++)
  {
    T tmp = data[i];
    for(j = i;j > first && tmp < data[j - 1];j--)
      data[j] = data[j - 1];
    data[j] = tmp;
  }
}
```

# Insertion sort and Shell sort (part 2, Shell sort description and code)

The Shell sort algorithm is nothing more that successive applications of the insertion sort algorithm to sub-arrays of the entire array. In a pass with stride h the entire array a[0],a[1],...,a[n-1] is subdivided into the sub-arrays

a[0],a[h+0],a[2h+0],... a[1],a[h+1],a[2h+1],... ··· a[h-1],a[2h-1],a[3h-1],...

Any sequence of strides is possible, as long as the last one is a stride of 1 (i.e., the last pass is just a single insertion sort). The computational complexity of this algorithm depends on the sequence of strides that is used (the best sequences of strides is not known). There are sequences of strides that make the algorithm $o(n^2)$. For example, when $h(s) = 9 \cdot 2^s - 9 \cdot 2^{s/2} + 1$ when $s$ is even and $h(s) = 8 \cdot 2^s - 6 \cdot 2^{(s+1)/2} + 1$ when it is odd, where $s$ is the number of the pass (counting starts at the last one) gives rise to a $O(n^{4/3})$ algorithm.

The following C code is one possible implementation of the Shell sort algorithm.

```
void shell_sort(T *data,int first,int one_after_last)
{
  int i,j,h;

  for(h = 1;h < (one_after_last - first) / 3;h = 3 * h + 1)
    ; // when h is choosen in this way the number of passes is O(log n)
  while(h >= 1)
  { // for each stride h, use insertion sort
    for(i = first + h;i < one_after_last;i++)
    {
      T tmp = data[i];
      for(j = i;j - h >= first && tmp < data[j - h];j -= h)
        data[j] = data[j - h];
      data[j] = tmp;
    }
    h /= 3;
  }
}
```

# Quick sort (part 1, description)

In practice, a well implemented quick sort algorithm appears to be the fastest sorting algorithm. It is a recursive algorithm that can be described as follows:

To sort `a[lo]`,...,`a[hi-1]` in increasing order do:

1. [Terminal case.] If `hi` − `lo` is smaller than a predetermined value (say, $\mathbf{20}$), use insertion sort to sort the array and terminate the algorithm.
2. [Select pivot.] Select one of the array elements to be the "pivot"
3. [Partition array.] Subdivide the array into three parts: the array elements that are smaller than the pivot (these are placed at the beginning), the pivot, or array elements equal to the pivot in certain implementations (these are placed at the middle), and the array elements that are larger than the pivot (these are placed at the end).
4. [Recurse.] Apply the same algorithm to the smaller than the pivot and to the larger than the pivot parts of the array. After doing this terminate the algorithm.

The partition step requires $\boldsymbol{O(n)}$ work, where $\boldsymbol{n}$ = `hi` − `lo` (see C code of the next slides). The computational complexity of the best, average, and worst cases of the quick sort algorithm are $\boldsymbol{O(n \log n)}$, $\boldsymbol{O(n \log n)}$, and $\boldsymbol{O(n^2)}$, respectively. Without care, the worst case occurs when the array is already sorted!

# Quick sort (part 2, code)

The following C code is one possible implementation of the quick sort algorithm.

```
void quick_sort(T *data,int first,int one_after_last)
{
  int i,j,one_after_small,first_equal,n_smaller,n_larger,n_equal;
  T pivot,tmp;

  if(one_after_last - first < 20)
    insertion_sort(data,first,one_after_last);
  else
  {
    //
    // select pivot (median of three, the pivot's position will be one_after_last-1)
    //
#   define POS1  (first)
#   define POS2  (one_after_last - 1)
#   define POS3  ((first + one_after_last) / 2)
#   define TEST(pos1,pos2)  do if(data[pos1] > data[pos2])                                    \
                              { tmp = data[pos1]; data[pos1] = data[pos2]; data[pos2] = tmp; } \
                              while(0)
    TEST(POS1,POS2);  // bitonic
    TEST(POS1,POS3);  // sort of
    TEST(POS2,POS3);  // 3 items
#   undef POS1
#   undef POS2
#   undef POS3
#   undef TEST
```

— the code continues on the next slide —

— continuation of the code of the previous slide —

```
//
// 3-way partition. At the end of the while loop the items will be partitioned as follows:
// |first  "smaller part"|one_after_small  "larger part"|first_equal  "equal part"|one_after_last
//
one_after_small = first;
first_equal = one_after_last - 1;
pivot = data[first_equal];
i = first;
while(i < first_equal)
  if(data[i] < pivot)
  { // place data[i] in the "smaller than the pivot" part of the array
    tmp = data[i];
    data[i] = data[one_after_small]; // tricky! this does the right thing when
    data[one_after_small] = tmp;     //   i == one_after_small and when i > one_after_small
    i++;
    one_after_small++;
  }
  else if(data[i] == pivot)
  { // place data[i] in the "equal to the pivot" part of the array
    first_equal--;
    tmp = data[i];               // this is known to be the pivot, but we do it in this way
    data[i] = data[first_equal]; //   to make life easier to those that need to adapt this
    data[first_equal] = tmp;     //   code so that it deals with more complex data items
  }
  else
  { // data[i] becomes automatically part of the "larger than the pivot" part of the array
    i++;
  }
n_smaller = one_after_small - first;
n_larger = first_equal - one_after_small;
n_equal = one_after_last - first_equal;
```

— the code continues on the next slide —

— continuation of the code of the previous slide —

```
    j = (n_equal < n_larger) ? n_equal : n_larger;
    for(i = 0;i < j;i++)
    { // move the "equal to the pivot" part of the array to the middle
      tmp = data[one_after_small + i];
      data[one_after_small + i] = data[one_after_last - 1 - i];
      data[one_after_last - 1 - i] = tmp;
    }
    //
    // recurse
    //
    quick_sort(data,first,first + n_smaller);
    quick_sort(data,first + n_smaller + n_equal,one_after_last);
  }
}
```

# Merge sort (part 1, description)

Merge sort is a recursive algorithm that can be described as follows:

To sort `a[lo]`,...,`a[hi-1]` in increasing order do:

1. [Terminal case.] If `hi` − `lo` is smaller than a predetermined value (say, $\mathbf{20}$), use insertion sort to sort the array and terminate.
2. [Subdivide.] Subdivide the array into two nearly equal parts (the "first half" and the "second half".
3. [Recurse.] Apply the same algorithm the two two halves
3. [Merge.] merge the two already sorted halves. After doing this terminate the algorithm.

The merge step requires $\boldsymbol{O(n)}$ work, where $\boldsymbol{n}$ = `hi` − `lo` (see C code in the next slide). The computational complexity of the best, average, and worst cases of the merge sort algorithm are all $\boldsymbol{O(n \log n)}$.

# Merge sort (part 2, code)

The following C code is one possible implementation of the merge sort algorithm.

```
void merge_sort(T *data,int first,int one_after_last)
{
  int i,j,k,middle;
  T *buffer;

  if(one_after_last - first < 40) // do not allocate less than 40 bytes
    insertion_sort(data,first,one_after_last);
  else
  {
    middle = (first + one_after_last) / 2;
    merge_sort(data,first,middle);
    merge_sort(data,middle,one_after_last);
    buffer = (T *)malloc((size_t)(one_after_last - first) * sizeof(T)) - first; // no error check!
    i = first;  // first input (first half)
    j = middle; // second input (second half)
    k = first;  // merged output
    while(k < one_after_last)
      if(j == one_after_last || (i < middle && data[i] <= data[j]))
        buffer[k++] = data[i++];
      else
        buffer[k++] = data[j++];
    for(i = first;i < one_after_last;i++)
      data[i] = buffer[i];
    free(buffer + first);
  }
}
```

# Heap sort (part 1, description)

The heap sort algorithm uses a max-heap (see the slides of the T.05 lecture). It can be described as follows:

To sort `a[0]`,...,`a[n-1]` in increasing order do:

1. [Construct heap.] For $i = 0, 1, \ldots, n-1$ put `a[i]` in the max-heap.
2. [Sort.] For $i = n-1, n-2, \ldots, 0$ remove the largest element of the max-heap and store it in `a[i]`. After doing this terminate the algorithm.

Given the way this algorithm is structured the array and the heap can share the same memory area. The computational complexity of the best, average, and worst cases of the heap sort algorithm are, like for the merge sort algorithm, all $O(n \log n)$. Heap sort, however, does not require extra space.

# Heap sort (part 2, code)

The following C code is one possible implementation of the heap sort algorithm.

```
void heap_sort(T *data,int first,int one_after_last)
{
  int i,j,k,n;
  T tmp;

  data += first - 1;          // adjust pointer (data[first] becomes data[1])
  n = one_after_last - first; // number of items to sort
  //
  // phase 1. heap construction
  //
  for(i = n / 2;i >= 1;i--)
    for(j = i;2 * j <= n;j = k)
    {
      k = (2 * j + 1 <= n && data[2 * j + 1] > data[2 * j]) ? 2 * j + 1 : 2 * j;
      if(data[j] >= data[k])
        break;
      tmp = data[j];
      data[j] = data[k];
      data[k] = tmp;
    }
```

— the code continues on the next slide —

— continuation of the code of the previous slide —

```
  //
  // phase 2. sort (by successively removing the largest element)
  //
  while(n > 1)
  {
    tmp = data[1]; // largest
    data[1] = data[n];
    data[n--] = tmp;
    for(j = 1;2 * j <= n;j = k)
    {
      k = (2 * j + 1 <= n && data[2 * j + 1] > data[2 * j]) ? 2 * j + 1 : 2 * j;
      if(data[j] >= data[k])
        break;
      tmp = data[j];
      data[j] = data[k];
      data[k] = tmp;
    }
  }
}
```

# Tree sort (part 1, description)

The tree sort algorithm uses an ordered binary tree (see the slides of the T.05 lecture). It can be described as follows:

To sort `a[0]`,...,`a[n-1]` in increasing order do:

1. [Construct tree.] For $i = 0, 1, \ldots, n-1$ put `a[i]` in the ordered binary tree.
2. [Sort.] Traverse the tree using an in-order depth-first algorithm (see the `traverse_in_order_recursive` function in the T.05 lecture) and place the node contents back in `a[i]`. After doing this terminate the algorithm.

The computational complexity of the best and average cases of the tree sort algorithm are, like for the quick sort, merge sort, and heap sort algorithms, $O(n \log n)$. The worst case of tree sort is, however, $O(n^2)$, just like quick sort. This can happen when the when the input data is already in sorted order (either increasing of decreasing), because it those cases the tree degenerates into a linked list. Repeated values also pose problems.

Using a balanced binary tree reduces the computational complexity of the worst case to $O(n \log n)$.

# Other sorting routines (part 1: rank sort)

The rank sort algorithm sorts by first counting the number of elements of the array that are smaller than each given element. This is called ranking, and gives the final sorted position of each element. Two extra arrays are needed: one to store the rank and another to keep a copy of the initial array.

The following C code is one possible implementation of the "rank" sort algorithm.

```
void rank_sort(T *data,int first,int one_after_last)
{
  int i,j,*rank;
  T *buffer;

  rank = (int *)malloc((size_t)(one_after_last - first) * sizeof(int)) - first; // no error check!
  for(i = first;i < one_after_last;i++)
    rank[i] = first;
  for(i = first + 1;i < one_after_last;i++)
    for(j = first;j < i;j++)
      rank[(data[i] < data[j]) ? j : i]++;
  buffer = (T *)malloc((size_t)(one_after_last - first) * sizeof(T)) - first; // no error check!
  for(i = first;i < one_after_last;i++)
    buffer[i] = data[i];
  for(i = first;i < one_after_last;i++)
    data[rank[i]] = buffer[i];
  free(buffer + first);
  free(rank + first);
}
```

This algorithm has a fixed computational complexity of $\Theta(n^2)$.

# Other sorting routines (part 2: selection sort)

The selection sort algorithm sorts an array of $n$ elements by making $n-1$ passes over the array. Each pass finds the largest element of the still unsorted portion of the array and at the end of the pass swaps it with last active element (the one at the end of the unsorted portion of the array).

The following C code is one possible implementation of the selection sort algorithm.

```
void selection_sort(T *data,int first,int one_after_last)
{
  int i,j,k;

  for(i = one_after_last - 1;i > first;i--)
  {
    for(j = first,k = 1;k <= i;k++)
      if(data[k] > data[j])
        j = k;
    if(j < i)
    {
      T tmp = data[i];
      data[i] = data[j];
      data[j] = tmp;
    }
  }
}
```

This algorithm also has a fixed computational complexity of $\Theta(n^2)$.

# Computational complexity summary

The following table presents the computational complexity (of the number of comparisons and data movements) of the best, average, and worst cases of the sorting algorithms described in this lecture ($n$ is the array size).

| algorithm | best | average | worst | comments |
|---|---|---|---|---|
| bubble sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ | |
| shaker sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ | |
| insertion sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ | |
| Shell sort | ? | ? | ? | |
| quick sort | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ | |
| merge sort | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | requires extra space |
| heap sort | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | |
| tree sort | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ (*) | |
| rank sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | requires extra space |
| selection sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | |

(*) The worst case is reduced to $(n \log n)$ if a balanced ordered binary tree is used.

# Ineffective sorts (spoiler)


StackSort connects to StackOverflow, searches for 'sort a list', and downloads and runs code snippets until the list is sorted.