1 Different Types of Sorting Algorithms

This chapter discusses sorting algorithms that use other approaches than comparisons to determine the order of the elements.

The maximum efficiency of comparison sorts, i.e. sorting based on the comparisons of the elements, is also examined.

Finally, the chapter covers the factors for choosing an algorithm.
1.1 Other Sorting Algorithms

All sorting algorithms covered so far have been based on comparisons.

- They determine the correct order of elements based on comparing their values to each other.

It is possible to use other information besides comparisons to sort data.
Sorting by counting

Let’s assume that the value range of the keys is small, at most the same range than the amount of the elements.

• For simplicity we assume that the keys of the elements are from the set \{1, 2, \ldots, k\}, and \(k = O(n)\).

• The amount of elements with each given key is calculated.

• Based on the result the elements are placed directly into their correct positions.
COUNTING-SORT($A$, $B$, $k$)

1. for $i := 1$ to $k$ do  
   \hspace{1cm} (initialize a temp array $C$ with zero)
2. \hspace{1cm} $C[i] := 0$
3. for $j := 1$ to $A$.length do  
   \hspace{1cm} (calculate the amount of elements with key = $i$)
5. for $i := 2$ to $k$ do  
   \hspace{1cm} (calculate how many keys $\leq i$)
6. \hspace{2cm} $C[i] := C[i] + C[i - 1]$
7. for $j := A$.length downto 1 do  
   \hspace{1cm} (scan the array from end to beginning)
8. \hspace{2cm} $B[C[A[j].key]] := A[j]$  
   \hspace{1cm} (place the element into the output array)
   \hspace{1cm} (the next correct location is a step to the left)

The algorithm places the elements to their correct location in reverse order to guarantee stability.
Running-time:
- The first and the third for-loop take $\Theta(k)$ time.
- The second and the last for-loop take $\Theta(n)$ time.

$\Rightarrow$ The running time is $\Theta(n + k)$.
- If $k = O(n)$, the running-time is $\Theta(n)$.
- All basic operations are simple and there are only a few of them in each loop so the constant coefficient is also small.

**COUNTING-SORT** is not worth using if $k \gg n$.
- The memory consumption of the algorithm is $\Theta(k)$.
- Usually $k \gg n$.
  - for example: all possible social security numbers $\gg$ the social security numbers of TUT personnel
Sometimes there is a need to be able to sort based on several keys or a key with several parts.

- the list of exam results: sort first based on the department and then those into an alphabetical order
- dates: first based on the year, then the month, and the day
- a deck of cards: first based on the suit and then those according to the numbers

The different criteria are taken into account as follows

- The most significant criterion according to which the values of the elements differ determines the result of the comparison.
- If the elements are equal with each criteria they are considered equal.
The problem can be solved with a comparison sort (e.g. by using a suitable comparison operator in QUICKSORT or MERGESORT). Example: comparing dates:

```
DATE-COMpare(x, y)
1  if x.year < y.year then return "smaller"
2  if x.year > y.year then return "greater"
3  if x.month < y.month then return "smaller"
4  if x.month > y.month then return "greater"
5  if x.day < y.day then return "smaller"
6  if x.day > y.day then return "greater"
7  return "equal"
```

Sometimes it makes sense to handle the input one criterion at a time.

- For example it’s easiest to sort a deck of cards into four piles based on the suits and then each suit separately.
The range of values in the significant criteria is often small when compared to the amount of elements and thus **COUNTING-SORT** can be used.

There are two different algorithms available for sorting with multiple keys:

- **LSD-RADIX-SORT**
  - the array is sorted first according to the least significant digit, then the second least significant etc.
  - the sorting algorithm needs to be stable - otherwise the array would be sorted only according to the most significant criterion
  - **COUNTING-SORT** is a suitable algorithm
  - comparison algorithms are not worth using since they would sort the array with approximately the same amount of effort directly at one go

LSD-RADIX-SORT\((A, d)\)
1. \(\text{for } i := 1 \text{ to } d \text{ do} \) (run through the criteria, least significant first)
2. \(\triangleright \text{ sort } A \) with a stable sort according to criterion \( i \)
• **MSD-RADIX-SORT**
  - the array is first sorted according to the most significant digit and then the subarrays with equal keys according to the next significant digit etc.
  - does not require the sorting algorithm to be stable
  - usable when sorting character strings of different lengths
  - checks only as many of the sorting criterions as is needed to determine the order
  - more complex to implement than LSD-RADIX-SORT

⇒ the algorithm is not given here
The efficiency of **Radix-Sort** when using **Counting-Sort**:  
- sorting according to one criterion: $\Theta(n + k)$  
- amount of different criteria is $d$  
  $\Rightarrow$ total efficiency $\Theta(dn + dk)$  
- $k$ is usually constant  
  $\Rightarrow$ total efficiency $\Theta(dn)$, or $\Theta(n)$, if $d$ is also constant  

**Radix-Sort** appears to be a $O(n)$ sorting algorithm with certain assumptions.  
Is it better than the comparison sorts in general?
When analyzing the efficiency of sorting algorithms it makes sense to assume that all (or most) of the elements have different values.

- For example INSERTION-SORT is $O(n)$, if all elements are equal.
- If the elements are all different and the size of value range of one criterion is constant $k$, $k^d \geq n \Rightarrow d \geq \log_k n = \Theta(\lg n)$
  $\Rightarrow$ RADIX-SORT is $\Theta(dn) = \Theta(n \lg n)$, if we assume that the element values are mostly different from each other.

RADIX-SORT is asymptotically as slow as other good sorting algorithms.

- By assuming a constant $d$, RADIX-SORT is $\Theta(n)$, but then with large values of $n$ most elements are equal to each other.
Advantages and disadvantages of \textsc{Radix-Sort}

Advantages:

\begin{itemize}
  \item \textsc{Radix-Sort} is able to compete in efficiency with \textsc{Quicksort} for example
    \begin{itemize}
      \item if the keys are 32-bit numbers and the array is sorted according to 8 bits at a time
        \begin{itemize}
          \item $k = 2^8$ and $d = 4$
          \item $\Rightarrow$ \textsc{Counting-Sort} is called four times
        \end{itemize}
    \end{itemize}
  \item \textsc{Radix-Sort} is well suited for sorting according to keys with multiple parts when the parts of the key have a small value range.
    \begin{itemize}
      \item e.g. sorting a text file according to the characters on the given columns (cmp. Unix or MS/DOS sort)
    \end{itemize}
\end{itemize}

Disadvantages:

\begin{itemize}
  \item \textsc{Counting-Sort} requires another array $B$ of $n$ elements where it builds the result and a temp array of $k$ elements.
    \begin{itemize}
      \item It requires $\Theta(n)$ extra memory which is significantly larger than for example with \textsc{Quicksort} and \textsc{Heapsort}.
    \end{itemize}
\end{itemize}
Bucket sort

Let’s assume that the keys are within a known range of values and the key values are evenly distributed.

• Each key is just as probable.
• For the sake of an example we’ll assume that the key values are between zero and one.
• Let’s use $n$ buckets $B[0]\ldots B[n-1]$.

\[
\text{BUCKET-SORT}(A)
\]

1. $n := A.length$
2. for $i := 1$ to $n$ do (go through the elements)
   3. \text{INSERT}(B[\lfloor n \cdot A[i] \rfloor], A[i]) (throw the element into the correct bucket)
4. $k := 1$ (start filling the array from index 1)
5. for $i := 0$ to $n - 1$ do (go through the buckets)
   6. while $B[i]$ not empty do (empty non-empty buckets...)
      7. $A[k] := \text{EXTRACT-MIN}(B[i])$ (... by moving the elements, smallest first...)
6. $k := k + 1$ (... into the correct location in the result array)
Implementation of the buckets:

- Operations \texttt{INSERT} and \texttt{EXTRACT-MIN} are needed.
  \(\Rightarrow\) The bucket is actually a priority queue.

- The size of the buckets varies a lot.
  
  - usually the amount of elements in the bucket is \(\approx 1\)
  
  - however it is possible that every element end up in the same bucket

  \(\Rightarrow\) an implementation that uses a heap would require \(\Theta(n)\) for each bucket, \(\Theta(n^2)\) in total

- On the other hand, the implementation does not need to be very efficient for large buckets since they are rare. \(\Rightarrow\) In practise the buckets should be implemented as lists.

  - \texttt{INSERT} links the incoming element to its correct location in the list, \(\Theta(\text{list length})\) time is used

  - \texttt{EXTRACT-MIN} removes and returns the first element in the list, \(\Theta(1)\) time is used
the average efficiency of \texttt{BUCKET-SORT}:

- We assumed the keys are evenly distributed.
  \[\Rightarrow\] On average one element falls into each bucket and very rarely a significantly larger amount of elements fall into the same bucket.

- The first \texttt{for}-loop runs through all of the elements, \(\Theta(n)\).

- The second \texttt{for}-loop runs through the buckets, \(\Theta(n)\).

- The \texttt{while}-loop runs through all of the elements in all of its iterations in total once, \(\Theta(n)\).

- \texttt{INSERT} is on average constant time, since there is on average one element in the bucket.

- \texttt{EXTRACT-MIN} is constant time.

\[\Rightarrow\] The total running-time is \(\Theta(n)\) on average.

In the slowest case all elements fall into the same bucket in an ascending order.

- \texttt{INSERT} takes a linear amount of time

\[\Rightarrow\] The total running-time is \(\Theta(n^2)\) in the worst-case.
1.2 How fast can we sort?

Sorting an array actually creates the permutation of its elements where the original array is completely sorted.

- If the elements are all different, the permutation is unique. ⇒ Sorting searches for that permutation from the set of all possible permutations.

For example the functionality of **INSERTION-SORT**, **MERGE-SORT**, **HEAPSORT** and **QUICKSORT** is based on comparisons between the elements.

- Information about the correct permutation is collected only by comparing the elements together.

What would be the smallest amount of comparisons that is enough to find the correct permutation for sure?
• An array of $n$ elements of different values has $1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ i.e. $n!$ permutations.

• The amount of comparisons needed must find the only correct alternative.

• Each comparison $A[i] \leq A[j]$ (or $A[i] < A[j]$) divides the permutations into two groups: those where the order of $A[i]$ and $A[j]$ must be switched and those where the order is correct so...

  – one comparison in enough to pick the right alternative from atmost two

  – two comparisons in enough to pick the right one from atmost four

  – …

  – $k$ comparisons in enough to pick the right alternative from atmost $2^k$

  $\Rightarrow$ choosing the right one from $x$ alternatives requires at least $\lceil \lg x \rceil$ comparisons
• If the size of the array is $n$, there are $n!$ permutations
  ⇒ At least $\lceil \lg n! \rceil$ comparisons is required
  ⇒ a comparison sort algorithm needs to use $\Omega(\lceil \lg n! \rceil)$ time.

How large is $\lceil \lg n! \rceil$ ?

• $\lceil \lg n! \rceil \geq \lg n! = \sum_{k=1}^{n} \lg k \geq \sum_{k=\lceil n/2 \rceil}^{n} \lg \frac{n}{2} \geq \frac{n}{2} \cdot \lg \frac{n}{2} = \frac{1}{2} n \lg n - \frac{1}{2} n = \Omega(n \lg n) - \Omega(n) = \Omega(n \lg n)$

• on the other hand $\lceil \lg n! \rceil < n \lg n + 1 = O(n \lg n)$
  ⇒ $\lceil \lg n! \rceil = \Theta(n \lg n)$
Every comparison sort algorithm needs to use $\Omega(n \lg n)$ time in the slowest case.

- On the other hand HEAPSORT and MERGE-SORT are $O(n \lg n)$ in the slowest case.
  
  $\Rightarrow$ In the slowest case sorting based on comparisons between elements is possible in $\Theta(n \lg n)$ time, but no faster.

- HEAPSORT and MERGE-SORT have an optimal asymptotic running-time in the slowest case.

- Sorting is truly asymptotically more time consuming than finding the median value, which can be done in the slowest possible case in $O(n)$. 
1.3 Choosing an algorithm

The key factor in choosing an algorithm is usually its efficiency in that situation. However, there are other factors:

- implementation is easy
  - is there a suitable algorithm already available?
  - is the advantage of improved efficiency worth the effort of implementing a more complex algorithm?
  - simpler code may not contain errors as easily
  - a simpler solution is easier to maintain

- precision of the results
  - with real numbers round-off errors can be a significant problem

- variation in the running-time
  - e.g. in signal processing the running-time must not vary at all
The programming environment also sets its limitations:

- many languages require that a maximum size is defined for arrays
  ⇒ algorithms using arrays get a compile time, artificial upper limit
  - with list structures and dynamic arrays the algorithm works as long as there is memory available in the computer

- the memory can suddenly run out with lists, but not with arrays of a fixed size
  ⇒ list structures are not always suitable for embedded systems

- in some computers the space reserved for recursion is much smaller than the space for the rest of the data
  ⇒ if a lot of memory is needed, a non-recursive algorithm (or implementation) must be chosen
If the efficiency is the primary factor in the selection, at least the following should be taken into account:

- Is the size of the input data so large that the asymptotic efficiency gives the right idea about the overall efficiency?
- Can the worst-case be slow if the efficiency is good in the average case?
- Is memory use a factor?
- Is there a certain regularity in the input that can be advantageous?
  - with one execution?
  - with several executions?
- Is there a certain regularity in the input that often is the worst-case for some algorithm?
Example: contacts

• operations
  – finding a phonenumber based on the name
  – adding a new contact into the phonebook
  – removing a contact and all the related information

• assumptions
  – additions and removals are needed rarely
  – phonenumber queries are done often
  – additions and removals are done in groups
1st attempt: unsorted array

<table>
<thead>
<tr>
<th>Virtanen</th>
<th>Järvinen</th>
<th>Lahtinen</th>
</tr>
</thead>
<tbody>
<tr>
<td>123 555</td>
<td>123 111</td>
<td>123 444</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>n</td>
</tr>
</tbody>
</table>

- Adding a new name to the end: $O(1)$.
- Searching by scanning the elements from the beginning (or end): $O(n)$.
- Removing by moving the last element to the place of the removed element: $O(1) + \text{search costs} = O(n)$.

$\Rightarrow$ The solution is not suitable since the operations that are needed often are slow.
2nd attempt: sorted array, 1st version

• Adding the new names directly to their correct location in the alphabetical order. The rest of the elements are moved one step forwards: $O(n)$.

• Removing by moving the elements one step backwards: $O(n)$.

• Searching with BIN-SEARCH: $\Theta(\lg n)$.

⇒ The search is efficient but the removal and addition are still slow.

• The solution seems better than the first attempt if our original assumption is correct and searches are done more frequently than additions and removals.
3rd attempt: an almost sorted array

- Keep most of the array sorted and a small unsorted segment at the end of the array (size $O(1)$).
- Additions are done to the segment at the end: $O(1)$.
- Search is done first with binary search in the sorted part and then if needed by scanning the unsorted part at the end: $O(\lg n) + O(l)$.
- Removal is done by leaving the name in and changing the number to 0: $O(1) + \text{search costs} = O(\lg n) + O(l)$.
- When the unsorted end segment has become too large, sort the entire array: $\Theta(n \lg n)$.
- A mediocre solution, but
  - $\Theta(l)$ can be large
  - sorting every now and then costs time
4th attempt: a sorted array, 2nd. version

- Let’s use the information that additions and removal are done in groups to our advantage.
- Sort the groups of additions and removals.
- Merge the array and the group of additions (like with \texttt{MERGE}) by removing the elements in the group of removals simultaneously.
- Now the search is still logarithmic.
- Addition and removal uses $O(l \lg l) + O(p \lg p) + O(n)$, when $l$ is the amount of additions and $p$ is the amount of removals
- Pretty good!

The problem could also be solved with dynamic sets covered later on the course and naturally, with programming languages’ library implementations such as the C++ containers.