WS 2007/2008

Fundamental Algorithms

Dmytro Chibisov, Jens Ernst

Fakultät für Informatik
TU München

http://www14.in.tum.de/lehre/2007WS/fa-cse/

Fall Semester 2007

1. Insertion Sort

This sorting algorithm differs from SelectionSort only in the way the inductive step is carried out. In this case, the induction hypothesis is strengthened:

1. Insertion Sort

This sorting algorithm differs from SelectionSort only in the way the inductive step is carried out. In this case, the induction hypothesis is strengthened:

- We introduce the invariant that $A[]$ consists of a sorted (left-hand side) and an unsorted (right-hand side) region.
- Initially the sorted region is of size 1 (base case).
- In each iteration, the sorted region is enlarged by 1 element, as follows. The left-most element from the unsorted region is inserted into the sorted region at the appropriate position.
- This way, the sorted region remains sorted and grows by 1 element.

1. Insertion Sort

This sorting algorithm differs from SelectionSort only in the way the inductive step is carried out. In this case, the induction hypothesis is strengthened:

- We introduce the invariant that $A[]$ consists of a sorted (left-hand side) and an unsorted (right-hand side) region.
- Initially the sorted region is of size 1 (base case).
- In each iteration, the sorted region is enlarged by 1 element, as follows. The left-most element from the unsorted region is inserted into the sorted region at the appropriate position.
- This way, the sorted region remains sorted and grows by 1 element.

1. Insertion Sort

This sorting algorithm differs from SelectionSort only in the way the inductive step is carried out. In this case, the induction hypothesis is strengthened:

- We introduce the invariant that $A[]$ consists of a sorted (left-hand side) and an unsorted (right-hand side) region.
- Initially the sorted region is of size 1 (base case).
- In each iteration, the sorted region is enlarged by 1 element, as follows. The left-most element from the unsorted region is inserted into the sorted region at the appropriate position.
- This way, the sorted region remains sorted and grows by 1 element.

1. Insertion Sort

This sorting algorithm differs from SelectionSort only in the way the inductive step is carried out. In this case, the induction hypothesis is strengthened:

- We introduce the invariant that $A[]$ consists of a sorted (left-hand side) and an unsorted (right-hand side) region.
- Initially the sorted region is of size 1 (base case).
- In each iteration, the sorted region is enlarged by 1 element, as follows. The left-most element from the unsorted region is inserted into the sorted region at the appropriate position.
- This way, the sorted region remains sorted and grows by 1 element.

Example 1

$$
\begin{aligned}
A &= \boxed{5}\boxed{31264} \\
A &= \boxed{35}\boxed{1264} \\
A &= \boxed{135}\boxed{264} \\
A &= \boxed{1235}\boxed{64} \\
A &= \boxed{12356}\boxed{4} \\
A &= \boxed{123456}\boxed{}
\end{aligned}
$$

Algorithm (outline):

void sort(unsigned n){
  for $i := 2$ to n do
    // insert $A[i]$ into $A[1 \cdots i-1]$ at the right position
    j :=findpos($A[], i$)// ($\forall k < j : A[k] < A[i]$ and $A[j] \geq A[i]$)
    if ($j < i$) then
      shift $A[j \cdots i]$ cyclically to the right by 1 position
    fi
  od
}

Remark: The cyclic shift of $A[j \cdots i]$ places $A[i]$ at the appropriate position, while keeping $A[1 \cdots i]$ sorted.
The sorted region grows by 1 element.

Algorithm (outline):

```
void sort(unsigned n){
  for i := 2 to n do
    // insert A[i] into A[1···i−1] at the right position
    j :=findpos(A[], i)// (∀k < j : A[k] < A[i] and A[j] ≥ A[i])
    if (j < i) then
      shift A[j···i] cyclically to the right by 1 position
    fi
  od
}
```

Remark: The cyclic shift of $A[j \cdots i]$ places $A[i]$ at the appropriate position, while keeping $A[1 \cdots i]$ sorted.
The sorted region grows by 1 element.

Correctness of InsertionSort.

We introduce the *loop invariant* that $A[]$ consists of a sorted (left-hand side) and an unsorted (right-hand side) region. We must show three things about a loop invariant:

- **Initialisation** (the loop invariant is true before the first iteration of the loop): The loop starts at $i = 2$. Before $A[1 \cdots i-1]$ consists of the single element $A[1]$. Thus, the loop invariant trivially correct.
- **Maintenance** (if the invariant is true before an iteration, it remains true before the next iteration): all elements that a larger then $A[j]$ will be shifted to the right by position. $A[j]$ will be inserted at the empty and correct position. Thus, $A[1 \cdots j]$ is a sorted array.
- **Termination**: The for-loop terminates when i exceeds n $(i = n+1)$. Thus, at termination $A[1 \cdots (n+1)-1] = A[1 \cdots n]$ will be sorted and contain all original elements.
- Thus, the algorithm is correct !

Correctness of InsertionSort.

We introduce the *loop invariant* that $A[]$ consists of a sorted (left-hand side) and an unsorted (right-hand side) region. We must show three things about a loop invariant:

- **Initialisation** (the loop invariant is true before the first iteration of the loop): The loop starts at $i = 2$. Before $A[1 \cdots i-1]$ consists of the single element $A[1]$. Thus, the loop invariant trivially correct.
- **Maintenance** (if the invariant is true before an iteration, it remains true before the next iteration): all elements that a larger then $A[j]$ will be shifted to the right by position. $A[j]$ will be inserted at the empty and correct position. Thus, $A[1 \cdots j]$ is a sorted array.
- **Termination**: The for-loop terminates when i exceeds n $(i = n+1)$. Thus, at termination $A[1 \cdots (n+1)-1] = A[1 \cdots n]$ will be sorted and contain all original elements.
- Thus, the algorithm is correct !

Correctness of InsertionSort.

We introduce the *loop invariant* that $A[]$ consists of a sorted (left-hand side) and an unsorted (right-hand side) region. We must show three things about a loop invariant:

- **Initialisation** (the loop invariant is true before the first iteration of the loop): The loop starts at $i = 2$. Before $A[1 \cdots i-1]$ consists of the single element $A[1]$. Thus, the loop invariant trivially correct.
- **Maintenance** (if the invariant is true before an iteration, it remains true before the next iteration): all elements that a larger then $A[j]$ will be shifted to the right by position. $A[j]$ will be inserted at the empty and correct position. Thus, $A[1 \cdots j]$ is a sorted array.
- **Termination**: The for-loop terminates when i exceeds n $(i = n+1)$. Thus, at termination $A[1 \cdots (n+1)-1] = A[1 \cdots n]$ will be sorted and contain all original elements.
- Thus, the algorithm is correct !

Correctness of InsertionSort.

We introduce the *loop invariant* that $A[]$ consists of a sorted (left-hand side) and an unsorted (right-hand side) region. We must show three things about a loop invariant:

- **Initialisation** (the loop invariant is true before the first iteration of the loop): The loop starts at $i = 2$. Before $A[1 \cdots i-1]$ consists of the single element $A[1]$. Thus, the loop invariant trivially correct.
- **Maintenance** (if the invariant is true before an iteration, it remains true before the next iteration): all elements that a larger then $A[j]$ will be shifted to the right by position. $A[j]$ will be inserted at the empty and correct position. Thus, $A[1 \cdots j]$ is a sorted array.
- **Termination**: The for-loop terminates when i exceeds n $(i = n + 1)$. Thus, at termination $A[1 \cdots (n+1) - 1] = A[1 \cdots n]$ will be sorted and contain all original elements.
- Thus, the algorithm is correct !

Correctness of InsertionSort.

We introduce the *loop invariant* that $A[]$ consists of a sorted (left-hand side) and an unsorted (right-hand side) region. We must show three things about a loop invariant:

- **Initialisation** (the loop invariant is true before the first iteration of the loop): The loop starts at $i = 2$. Before $A[1 \cdots i-1]$ consists of the single element $A[1]$. Thus, the loop invariant trivially correct.
- **Maintenance** (if the invariant is true before an iteration, it remains true before the next iteration): all elements that a larger then $A[j]$ will be shifted to the right by position. $A[j]$ will be inserted at the empty and correct position. Thus, $A[1 \cdots j]$ is a sorted array.
- **Termination**: The for-loop terminates when i exceeds n $(i = n+1)$. Thus, at termination $A[1 \cdots (n+1)-1] = A[1 \cdots n]$ will be sorted and contain all original elements.
- Thus, the algorithm is correct !

Correctness of InsertionSort.

We introduce the *loop invariant* that $A[]$ consists of a sorted (left-hand side) and an unsorted (right-hand side) region. We must show three things about a loop invariant:

- **Initialisation** (the loop invariant is true before the first iteration of the loop): The loop starts at $i = 2$. Before $A[1 \cdots i-1]$ consists of the single element $A[1]$. Thus, the loop invariant trivially correct.
- **Maintenance** (if the invariant is true before an iteration, it remains true before the next iteration): all elements that a larger then $A[j]$ will be shifted to the right by position. $A[j]$ will be inserted at the empty and correct position. Thus, $A[1 \cdots j]$ is a sorted array.
- **Termination**: The for-loop terminates when i exceeds n $(i = n+1)$. Thus, at termination $A[1 \cdots (n+1)-1] = A[1 \cdots n]$ will be sorted and contain all original elements.
- Thus, the algorithm is correct !

When trying to minimize the number of comparisons, we need to optimize function findpos(). Several options exist for finding the position at which $A[i]$ has to be inserted.

1.1 Linear Search

Walking from left to right through the relevant part of $A[]$ to find the appropriate insert position j costs $j \leq i$ comparisions. Hence, the number of key comparisons is $c_{fp}(i) = O(i)$ for the i-th iteration.

Even the expected number of comparisons is $\Theta(i)$, assuming that all input permutations are equally likely.
The total complexity of InsertionSort then turns out to be

$$\sum_{i=2}^{n} c_{fp}(i) = O(n^2).$$

This is no improvement over SelectionSort.

When trying to minimize the number of comparisons, we need to optimize function findpos(). Several options exist for finding the position at which $A[i]$ has to be inserted.

1.1 Linear Search

Walking from left to right through the relevant part of $A[]$ to find the appropriate insert position j costs $j \leq i$ comparisions. Hence, the number of key comparisons is $c_{fp}(i) = O(i)$ for the i-th iteration.
Even the expected number of comparisons is $\Theta(i)$, assuming that all input permutations are equally likely.
The total complexity of InsertionSort then turns out to be

$$\sum_{i=2}^{n} c_{fp}(i) = O(n^2).$$

This is no improvement over SelectionSort.

When trying to minimize the number of comparisons, we need to optimize function findpos(). Several options exist for finding the position at which $A[i]$ has to be inserted.

1.1 Linear Search

Walking from left to right through the relevant part of $A[]$ to find the appropriate insert position j costs $j \leq i$ comparisions. Hence, the number of key comparisons is $c_{fp}(i) = O(i)$ for the i-th iteration.

Even the expected number of comparisons is $\Theta(i)$, assuming that all input permutations are equally likely.

The total complexity of InsertionSort then turns out to be

$$\sum_{i=2}^{n} c_{fp}(i) = O(n^2).$$

This is no improvement over SelectionSort.

1.2 Binary Search

Remember this for the rest of your life: Searching in a sorted array never requires a linear scan. Instead, we can apply a technique known as Binary Search. (Remember, for now no two keys have the same value). Idea:

1.2 Binary Search

Remember this for the rest of your life: Searching in a sorted array never requires a linear scan. Instead, we can apply a technique known as Binary Search. (Remember, for now no two keys have the same value). Idea:

- Compare the element to be searched to the element at the middle position of the array (or one of the two middle positions).
- If it matches, the occurrence has been found.
- Otherwise, an occurrence can only exist either to the right or to the left of this position, not both. The comparison shows on which side we should continue the search. The other side is excluded once and for all.
- Now the search region is cut in half. Continue recursively on the appropriate side until a match is found of the search region becomes empty.

1.2 Binary Search

Remember this for the rest of your life: Searching in a sorted array never requires a linear scan. Instead, we can apply a technique known as Binary Search. (Remember, for now no two keys have the same value). Idea:

- Compare the element to be searched to the element at the middle position of the array (or one of the two middle positions).
- If it matches, the occurrence has been found.
- Otherwise, an occurrence can only exist either to the right or to the left of this position, not both. The comparison shows on which side we should continue the search. The other side is excluded once and for all.
- Now the search region is cut in half. Continue recursively on the appropriate side until a match is found of the search region becomes empty.

1.2 Binary Search

Remember this for the rest of your life: Searching in a sorted array never requires a linear scan. Instead, we can apply a technique known as Binary Search. (Remember, for now no two keys have the same value). Idea:

- Compare the element to be searched to the element at the middle position of the array (or one of the two middle positions).
- If it matches, the occurrence has been found.
- Otherwise, an occurrence can only exist either to the right or to the left of this position, not both. The comparison shows on which side we should continue the search. The other side is excluded once and for all.
- Now the search region is cut in half. Continue recursively on the appropriate side until a match is found of the search region becomes empty.

1.2 Binary Search

Remember this for the rest of your life: Searching in a sorted array never requires a linear scan. Instead, we can apply a technique known as Binary Search. (Remember, for now no two keys have the same value). Idea:

- Compare the element to be searched to the element at the middle position of the array (or one of the two middle positions).
- If it matches, the occurrence has been found.
- Otherwise, an occurrence can only exist either to the right or to the left of this position, not both. The comparison shows on which side we should continue the search. The other side is excluded once and for all.
- Now the search region is cut in half. Continue recursively on the appropriate side until a match is found of the search region becomes empty.

Algorithm (findpos):

unsigned findpos (unsigned i){
  $\ell := 1$
  $r := i$
  $pos := \lfloor (\ell + r)/2 \rfloor$
  while ($\ell < r$) do
    if ($A[pos] < A[i]$) then $\ell := pos + 1$
    else $r := pos$
    fi
    $pos := \lfloor (\ell + r)/2 \rfloor$
  od
  return ℓ
}

Complexity:

In general, the number of times a positive integer k has to be divided by 2 until it becomes ≤ 1 is $\Theta(\log k)$. Here, $k = (r - \ell + 1) = i$ is the initial size of the search region. Hence, findpos(i) requires $c_{fp}(i) = O(\log i)$ key comparisons.

(Why is it not accurate to claim that $c_{fp}(i) = \Theta(\log i)$?)

(Answer: If a match is found before the search region is exhausted, the number of key comparisons is less than $\log i$.)

The overall number of key comparisons in InsertionSort, using binary search, is

$$O\left(\sum_{i=2}^{n} c_{fp}(i)\right) = O(n \log n).$$

Note: all the methods above fall into the inductive framework mentioned at the beginning of this chapter. We shall now see how this can be taken further...

Complexity:

In general, the number of times a positive integer k has to be divided by 2 until it becomes ≤ 1 is $\Theta(\log k)$. Here, $k = (r - \ell + 1) = i$ is the initial size of the search region. Hence, findpos(i) requires $c_{fp}(i) = O(\log i)$ key comparisons.

(Why is it not accurate to claim that $c_{fp}(i) = \Theta(\log i)$?)

(Answer: If a match is found before the search region is exhausted, the number of key comparisons is less than $\log i$.)

The overall number of key comparisons in InsertionSort, using binary search, is

$$O\left(\sum_{i=2}^{n} c_{fp}(i)\right) = O(n \log n).$$

Note: all the methods above fall into the inductive framework mentioned at the beginning of this chapter. We shall now see how this can be taken further...

Complexity:

In general, the number of times a positive integer k has to be divided by 2 until it becomes ≤ 1 is $\Theta(\log k)$. Here, $k = (r - \ell + 1) = i$ is the initial size of the search region. Hence, findpos(i) requires $c_{fp}(i) = O(\log i)$ key comparisons.

(Why is it not accurate to claim that $c_{fp}(i) = \Theta(\log i)$?)

(Answer: If a match is found before the search region is exhausted, the number of key comparisons is less than $\log i$.)

The overall number of key comparisons in InsertionSort, using binary search, is

$$O\left(\sum_{i=2}^{n} c_{fp}(i)\right) = O(n \log n).$$

Note: all the methods above fall into the inductive framework mentioned at the beginning of this chapter. We shall now see how this can be taken further...

Complexity:

In general, the number of times a positive integer k has to be divided by 2 until it becomes ≤ 1 is $\Theta(\log k)$. Here, $k = (r - \ell + 1) = i$ is the initial size of the search region. Hence, findpos(i) requires $c_{fp}(i) = O(\log i)$ key comparisons.

(Why is it not accurate to claim that $c_{fp}(i) = \Theta(\log i)$?)

(Answer: If a match is found before the search region is exhausted, the number of key comparisons is less than $\log i$.)

The overall number of key comparisons in InsertionSort, using binary search, is

$$O\left(\sum_{i=2}^{n} c_{fp}(i)\right) = O(n \log n).$$

Note: all the methods above fall into the inductive framework mentioned at the beginning of this chapter. We shall now see how this can be taken further...

Complexity:

In general, the number of times a positive integer k has to be divided by 2 until it becomes ≤ 1 is $\Theta(\log k)$. Here, $k = (r - \ell + 1) = i$ is the initial size of the search region. Hence, findpos(i) requires $c_{fp}(i) = O(\log i)$ key comparisons.

(Why is it not accurate to claim that $c_{fp}(i) = \Theta(\log i)$?)

(Answer: If a match is found before the search region is exhausted, the number of key comparisons is less than $\log i$.)

The overall number of key comparisons in InsertionSort, using binary search, is

$$O\left(\sum_{i=2}^{n} c_{fp}(i)\right) = O(n \log n).$$

Note: all the methods above fall into the inductive framework mentioned at the beginning of this chapter. We shall now see how this can be taken further...

2. Merge Sort

To derive this algorithm, we choose a different way of reducing the problem size n in the inductive step.

In this case we reduce n to $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ by cutting the unsorted array in two sections of (approx.) equal size.

This strategy makes MergeSort a Divide and Conquer algorithm.

Let us suppose we have a second array $B[]$ of the same size as $A[]$ to store intermediate results.

2. Merge Sort

To derive this algorithm, we choose a different way of reducing the problem size n in the inductive step.

In this case we reduce n to $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ by cutting the unsorted array in two sections of (approx.) equal size.

This strategy makes MergeSort a Divide and Conquer algorithm.

Let us suppose we have a second array $B[]$ of the same size as $A[]$ to store intermediate results.

2. Merge Sort

To derive this algorithm, we choose a different way of reducing the problem size n in the inductive step.

In this case we reduce n to $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ by cutting the unsorted array in two sections of (approx.) equal size.

This strategy makes MergeSort a Divide and Conquer algorithm.

I. Base case: trivial

II. Let an array of length n be given.

Let us suppose we have a second array $B[]$ of the same size as $A[]$ to store intermediate results.

2. Merge Sort

To derive this algorithm, we choose a different way of reducing the problem size n in the inductive step.

In this case we reduce n to $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ by cutting the unsorted array in two sections of (approx.) equal size.

This strategy makes MergeSort a Divide and Conquer algorithm.

I. Base case: trivial
II. Let an array of length n be given.
    - Divide it in the middle
    - Sort the two halves by induction, assuming we know how to sort arrays of length $< n$
    - Recombine the two sorted halves in such a way that the total array is sorted. This is a linear-time operation.

Let us suppose we have a second array $B[]$ of the same size as $A[]$ to store intermediate results.

2. Merge Sort

To derive this algorithm, we choose a different way of reducing the problem size n in the inductive step.

In this case we reduce n to $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ by cutting the unsorted array in two sections of (approx.) equal size.

This strategy makes MergeSort a Divide and Conquer algorithm.

I. Base case: trivial
II. Let an array of length n be given.
   - Divide it in the middle
   - Sort the two halves by induction, assuming we know how to sort arrays of length $< n$
   - Recombine the two sorted halves in such a way that the total array is sorted. This is a linear-time operation.

Let us suppose we have a second array $B[]$ of the same size as $A[]$ to store intermediate results.

2. Merge Sort

To derive this algorithm, we choose a different way of reducing the problem size n in the inductive step.

In this case we reduce n to $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ by cutting the unsorted array in two sections of (approx.) equal size.

This strategy makes MergeSort a Divide and Conquer algorithm.

I. Base case: trivial

II. Let an array of length n be given.
   - Divide it in the middle
   - Sort the two halves by induction, assuming we know how to sort arrays of length $< n$
   - Recombine the two sorted halves in such a way that the total array is sorted. This is a linear-time operation.

Let us suppose we have a second array $B[]$ of the same size as $A[]$ to store intermediate results.

2. Merge Sort

To derive this algorithm, we choose a different way of reducing the problem size n in the inductive step.

In this case we reduce n to $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ by cutting the unsorted array in two sections of (approx.) equal size.

This strategy makes MergeSort a Divide and Conquer algorithm.

I. Base case: trivial
II. Let an array of length n be given.
    - Divide it in the middle
    - Sort the two halves by induction, assuming we know how to sort arrays of length $< n$
    - Recombine the two sorted halves in such a way that the total array is sorted. This is a linear-time operation.

Let us suppose we have a second array $B[]$ of the same size as $A[]$ to store intermediate results.

2. Merge Sort

To derive this algorithm, we choose a different way of reducing the problem size n in the inductive step.

In this case we reduce n to $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ by cutting the unsorted array in two sections of (approx.) equal size.

This strategy makes MergeSort a Divide and Conquer algorithm.

I. Base case: trivial

II. Let an array of length n be given.
   - Divide it in the middle
   - Sort the two halves by induction, assuming we know how to sort arrays of length $< n$
   - Recombine the two sorted halves in such a way that the total array is sorted. This is a linear-time operation.

Let us suppose we have a second array $B[]$ of the same size as $A[]$ to store intermediate results.

2. Merge Sort

To derive this algorithm, we choose a different way of reducing the problem size n in the inductive step.

In this case we reduce n to $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ by cutting the unsorted array in two sections of (approx.) equal size.

This strategy makes MergeSort a Divide and Conquer algorithm.

I. Base case: trivial

II. Let an array of length n be given.
   - Divide it in the middle
   - Sort the two halves by induction, assuming we know how to sort arrays of length $< n$
   - Recombine the two sorted halves in such a way that the total array is sorted. This is a linear-time operation.

Let us suppose we have a second array $B[]$ of the same size as $A[]$ to store intermediate results.

Algorithm (framework):

// sort $A[\ell \cdots r]$

void MergeSort (key $A[]$, key $B[]$, unsigned ℓ, r){
  if ($l == r$) then return
  else
    unsigned $m := \lfloor(\ell + r)/2\rfloor$
    MergeSort(A, B, ℓ, m)
    MergeSort($A, B, m + 1, r$)
    merge(A, B, ℓ, r)
  fi
}

Algorithm (framework):

```
// sort A[ℓ···r]
void MergeSort (key A[], key B[], unsigned ℓ, r){
  if (l == r) then return
  else
    unsigned m := ⌊(ℓ + r)/2⌋
    MergeSort(A, B, ℓ, m)
    MergeSort(A, B, m + 1, r)
    merge(A, B, ℓ, r)
  fi
}
```

- We will implement the merge-function such that the sorted result is stored in $A[]$
- Hence, the two recursive calls yield two sorted subarrays: $A[\ell \cdots m]$ and $A[m+1 \cdots r]$
- Here is how to implement the merge step:

Algorithm (framework):

```
// sort A[ℓ···r]
void MergeSort (key A[], key B[], unsigned ℓ, r){
   if (l == r) then return
   else
      unsigned m := ⌊(ℓ + r)/2⌋
      MergeSort(A, B, ℓ, m)
      MergeSort(A, B, m + 1, r)
      merge(A, B, ℓ, r)
   fi
}
```

- We will implement the merge-function such that the sorted result is stored in $A[]$
- Hence, the two recursive calls yield two sorted subarrays: $A[\ell \cdots m]$ and $A[m+1 \cdots r]$
- Here is how to implement the merge step:

Algorithm (framework):

```
// sort A[ℓ···r]
void MergeSort (key A[], key B[], unsigned ℓ, r){
  if (l == r) then return
  else
    unsigned m := ⌊(ℓ + r)/2⌋
    MergeSort(A, B, ℓ, m)
    MergeSort(A, B, m + 1, r)
    merge(A, B, ℓ, r)
  fi
}
```

- We will implement the merge-function such that the sorted result is stored in $A[]$
- Hence, the two recursive calls yield two sorted subarrays: $A[\ell \cdots m]$ and $A[m+1 \cdots r]$
- Here is how to implement the merge step:

Merge function:

void merge(key $A[]$, key $B[]$, unsigned ℓ, m, r){
  unsigned $i := \ell$ //pointer into the left half
  unsigned $j := m + 1$ //pointer into the right half
  unsigned $k := \ell$//pointer into $B[]$
  while $k \leq r$ do
    if $(j > i) \vee (i \leq m \wedge A[i] \leq A[j])$ then
      // if 2nd half exhausted or elt from 1st half smaller
      $B[k++] := A[i++]$
    else
      $B[k++] := A[j++]$
    fi
  od
  for $(i := \ell$ to $r)$ do
    $A[i] := B[i]$
  od
}

Complexity: As for the previous algorithms, we analyze the number $c(n)$ of key comparisons that have to be carried out by MergeSort, when applied to an array of length n.

All comparisons of key values take place in the merge-function. Calling merge() for two (sub)-arrays of lengths n' and n'' then costs $(n' + n'' - 1)$ comparisons.

As MergeSort was formulated as a recursive function, $c(n)$ can be expressed most easily in the form of a recurrence relation. The number of key comparisons needed to sort n numbers is equal to the number of key comparisons needed to independently sort the two halves of the input array, plus the number of key comparisons needed to recombine the two sorted subarrays:

$$c(n) = \begin{cases} 0 & \text{for } n = 1 \\ c\left(\lceil \frac{n}{2} \rceil\right) + c\left(\lfloor \frac{n}{2} \rfloor\right) + (n-1) & \text{for } n \geq 2 \end{cases}$$

Complexity: As for the previous algorithms, we analyze the number $c(n)$ of key comparisons that have to be carried out by MergeSort, when applied to an array of length n.

All comparisons of key values take place in the merge-function. Calling merge() for two (sub)-arrays of lengths n' and n'' then costs $(n' + n'' - 1)$ comparisons.

As MergeSort was formulated as a recursive function, $c(n)$ can be expressed most easily in the form of a recurrence relation. The number of key comparisons needed to sort n numbers is equal to the number of key comparisons needed to independently sort the two halves of the input array, plus the number of key comparisons needed to recombine the two sorted subarrays:

$$c(n) = \begin{cases} 0 & \text{for } n = 1 \\ c\left(\lceil \frac{n}{2} \rceil\right) + c\left(\lfloor \frac{n}{2} \rfloor\right) + (n-1) & \text{for } n \geq 2 \end{cases}$$

Complexity: As for the previous algorithms, we analyze the number $c(n)$ of key comparisons that have to be carried out by MergeSort, when applied to an array of length n.

All comparisons of key values take place in the merge-function. Calling merge() for two (sub)-arrays of lengths n' and n'' then costs $(n' + n'' - 1)$ comparisons.

As MergeSort was formulated as a recursive function, $c(n)$ can be expressed most easily in the form of a recurrence relation. The number of key comparisons needed to sort n numbers is equal to the number of key comparisons needed to independently sort the two halves of the input array, plus the number of key comparisons needed to recombine the two sorted subarrays:

$$c(n) = \begin{cases} 0 & \text{for } n = 1 \\ c\left(\lceil \frac{n}{2} \rceil\right) + c\left(\lfloor \frac{n}{2} \rfloor\right) + (n-1) & \text{for } n \geq 2 \end{cases}$$

Lemma 2

Consider the recurrence relation for the real-valued parameter x

$$c'(x) = \begin{cases} 0 & \text{for } x \in \mathbf{R}, 1 \leq x < 2 \\ 2 \cdot c'\left(\left\lceil \frac{x+1}{2} \right\rceil\right) + (x-1) & \text{for } x \in \mathbf{R}, x \geq 2. \end{cases}$$

For all $n \in \mathbf{N}$ and any $\varepsilon > 0$, it holds that $c(n) \leq c'(n+\varepsilon)$.

(Proof omitted)

Lemma 3

Moreover, it holds that

$$c'(n) = \Theta(n \log n).$$

(Proof: homework)

Theorem 4

The MergeSort algorithm shown above takes $O(n \log n)$ comparisons between key values to sort n numbers.

3. Heap Sort

HeapSort is an efficient sorting algorithm based on an efficient method for storing keys, i.e. a data structure known as heap. To describe this data structure we need to introduce some basic terminology of graph theory.

3.1 Introduction to Graphs, Trees and Heaps

Definition 5

A graph is a pair $G = (V, E)$ of a set $V = \{v_1, v_2, \ldots, v_n\}$ of n vertices and a set $E = \{e_1, e_2, \ldots, e_m\}$ of m edges. In the case of an undirected graph, an edge is a set $e_i = \{v_j, v_k\} \subset V$. In case of a directed graph, an edge is a vertex pair $e_i = (v_j, v_k) \in V^2$.

3. Heap Sort

HeapSort is an efficient sorting algorithm based on an efficient method for storing keys, i.e. a data structure known as heap. To describe this data structure we need to introduce some basic terminology of graph theory.

3.1 Introduction to Graphs, Trees and Heaps

Definition 5

A graph is a pair $G = (V, E)$ of a set $V = \{v_1, v_2, \ldots, v_n\}$ of n vertices and a set $E = \{e_1, e_2, \ldots, e_m\}$ of m edges. In the case of an undirected graph, an edge is a set $e_i = \{v_j, v_k\} \subset V$. In case of a directed graph, an edge is a vertex pair $e_i = (v_j, v_k) \in V^2$.

3. Heap Sort

HeapSort is an efficient sorting algorithm based on an efficient method for storing keys, i.e. a data structure known as heap. To describe this data structure we need to introduce some basic terminology of graph theory.

3.1 Introduction to Graphs, Trees and Heaps

Definition 5

A graph is a pair $G = (V, E)$ of a set $V = \{v_1, v_2, \ldots, v_n\}$ of n vertices and a set $E = \{e_1, e_2, \ldots, e_m\}$ of m edges. In the case of an undirected graph, an edge is a set $e_i = \{v_j, v_k\} \subset V$. In case of a directed graph, an edge is a vertex pair $e_i = (v_j, v_k) \in V^2$.

Definition 6

An undirected tree an be recursively defined as follows: A graph $T = (\{v\}, \emptyset)$ consisting of only one vertex is an undirected tree. And a graph $T = (V, E)$ in which some vertex $v \in V$ is connected by one edge to any number of undirected trees $T_1, T_2, \ldots, T_k$ is an undirected tree.

Definition 7

- In the latter case, v can be considered the root of T, and $T_1, T_2, \ldots, T_k$ can be considered its subtrees.
- Let $w_1, w_2, \ldots, w_k$ be the roots of the subtrees. Then v is called the father of the w_i, and each w_i is v's son.
- A vertex without children is called a leaf. All others are internal vertices.
- The father, grandfather, etc. of a vertex are called ancestors of this vertex.
- If v is an ancestor of w then w is descendent of v.

Definition 6

An undirected tree an be recursively defined as follows: A graph $T = (\{v\}, \emptyset)$ consisting of only one vertex is an undirected tree. And a graph $T = (V, E)$ in which some vertex $v \in V$ is connected by one edge to any number of undirected trees $T_1, T_2, \ldots, T_k$ is an undirected tree.

Definition 7

- In the latter case, v can be considered the root of T, and $T_1, T_2, \ldots, T_k$ can be considered its subtrees.
- Let $w_1, w_2, \ldots, w_k$ be the roots of the subtrees. Then v is called the father of the w_i, and each w_i is v's son.
- A vertex without children is called a leaf. All others are internal vertices.
- The father, grandfather, etc. of a vertex are called ancestors of this vertex.
- If v is an ancestor of w then w is descendent of v.

Definition 6

An undirected tree an be recursively defined as follows: A graph $T = (\{v\}, \emptyset)$ consisting of only one vertex is an undirected tree. And a graph $T = (V, E)$ in which some vertex $v \in V$ is connected by one edge to any number of undirected trees $T_1, T_2, \ldots, T_k$ is an undirected tree.

Definition 7

- In the latter case, v can be considered the root of T, and $T_1, T_2, \ldots, T_k$ can be considered its subtrees.
- Let $w_1, w_2, \ldots, w_k$ be the roots of the subtrees. Then v is called the father of the w_i, and each w_i is v's son.
- A vertex without children is called a leaf. All others are internal vertices.
- The father, grandfather, etc. of a vertex are called ancestors of this vertex.
- If v is an ancestor of w then w is descendent of v.

Definition 6

An undirected tree an be recursively defined as follows: A graph $T = (\{v\}, \emptyset)$ consisting of only one vertex is an undirected tree. And a graph $T = (V, E)$ in which some vertex $v \in V$ is connected by one edge to any number of undirected trees $T_1, T_2, \ldots, T_k$ is an undirected tree.

Definition 7

- In the latter case, v can be considered the root of T, and $T_1, T_2, \ldots, T_k$ can be considered its subtrees.
- Let $w_1, w_2, \ldots, w_k$ be the roots of the subtrees. Then v is called the father of the w_i, and each w_i is v's son.
- A vertex without children is called a leaf. All others are internal vertices.
- The father, grandfather, etc. of a vertex are called ancestors of this vertex.
- If v is an ancestor of w then w is descendent of v.

Definition 6

An undirected tree an be recursively defined as follows: A graph $T = (\{v\}, \emptyset)$ consisting of only one vertex is an undirected tree. And a graph $T = (V, E)$ in which some vertex $v \in V$ is connected by one edge to any number of undirected trees $T_1, T_2, \ldots, T_k$ is an undirected tree.

Definition 7

- In the latter case, v can be considered the root of T, and $T_1, T_2, \ldots, T_k$ can be considered its subtrees.
- Let $w_1, w_2, \ldots, w_k$ be the roots of the subtrees. Then v is called the father of the w_i, and each w_i is v's son.
- A vertex without children is called a leaf. All others are internal vertices.
- The father, grandfather, etc. of a vertex are called ancestors of this vertex.
- If v is an ancestor of w then w is descendent of v.

Definition 8

- Let T be a tree with root w. Then the level of w is 1. If v is some vertex in the tree whose father is v' then the level of v is equal to the the level of v' plus one. Hence, the level of a vertex gives its depth in tree, or its distance from the root.
- The depth $d(T)$ of tree T is defined as the maximum of all its nodes' levels.

Definition 9

An undirected binary tree is an undirected tree in which each vertex has at most 2 children.

Lemma 10

Let T be an undirected binary tree of depth $d = d(T)$. Then

Definition 8

- Let T be a tree with root w. Then the level of w is 1. If v is some vertex in the tree whose father is v' then the level of v is equal to the the level of v' plus one. Hence, the level of a vertex gives its depth in tree, or its distance from the root.
- The depth $d(T)$ of tree T is defined as the maximum of all its nodes' levels.

Definition 9

An undirected binary tree is an undirected tree in which each vertex has at most 2 children.

Lemma 10

Let T be an undirected binary tree of depth $d = d(T)$. Then

Definition 8

- Let T be a tree with root w. Then the level of w is 1. If v is some vertex in the tree whose father is v' then the level of v is equal to the the level of v' plus one. Hence, the level of a vertex gives its depth in tree, or its distance from the root.
- The depth $d(T)$ of tree T is defined as the maximum of all its nodes' levels.

Definition 9

An undirected binary tree is an undirected tree in which each vertex has at most 2 children.

Lemma 10

Let T be an undirected binary tree of depth $d = d(T)$. Then

Definition 8

- Let T be a tree with root w. Then the level of w is 1. If v is some vertex in the tree whose father is v' then the level of v is equal to the the level of v' plus one. Hence, the level of a vertex gives its depth in tree, or its distance from the root.
- The depth $d(T)$ of tree T is defined as the maximum of all its nodes' levels.

Definition 9

An undirected binary tree is an undirected tree in which each vertex has at most 2 children.

Lemma 10

Let T be an undirected binary tree of depth $d = d(T)$. Then

Definition 8

- Let T be a tree with root w. Then the level of w is 1. If v is some vertex in the tree whose father is v' then the level of v is equal to the the level of v' plus one. Hence, the level of a vertex gives its depth in tree, or its distance from the root.
- The depth $d(T)$ of tree T is defined as the maximum of all its nodes' levels.

Definition 9

An undirected binary tree is an undirected tree in which each vertex has at most 2 children.

Lemma 10

Let T be an undirected binary tree of depth $d = d(T)$. Then

- *the maximum number of vertices on level ℓ within T is $2^{\ell-1}$,*
- *the total number of vertices in T is $2^d - 1$,*
- *the number of leaves in T is at most 2^{d-1}.*

Definition 8

- Let T be a tree with root w. Then the level of w is 1. If v is some vertex in the tree whose father is v' then the level of v is equal to the the level of v' plus one. Hence, the level of a vertex gives its depth in tree, or its distance from the root.
- The depth $d(T)$ of tree T is defined as the maximum of all its nodes' levels.

Definition 9

An undirected binary tree is an undirected tree in which each vertex has at most 2 children.

Lemma 10

Let T be an undirected binary tree of depth $d = d(T)$. Then

- *the maximum number of vertices on level ℓ within T is $2^{\ell-1}$,*
- *the total number of vertices in T is $2^d - 1$,*
- *the number of leaves in T is at most 2^{d-1}.*

Definition 8

- Let T be a tree with root w. Then the level of w is 1. If v is some vertex in the tree whose father is v' then the level of v is equal to the the level of v' plus one. Hence, the level of a vertex gives its depth in tree, or its distance from the root.
- The depth $d(T)$ of tree T is defined as the maximum of all its nodes' levels.

Definition 9

An undirected binary tree is an undirected tree in which each vertex has at most 2 children.

Lemma 10

Let T be an undirected binary tree of depth $d = d(T)$. Then

- *the maximum number of vertices on level ℓ within T is $2^{\ell-1}$,*
- *the total number of vertices in T is $2^d - 1$,*
- *the number of leaves in T is at most 2^{d-1}.*

Definition 11

A binary tree is complete if all its leaves have the same depth and all levels are filled with vertices to capacity. An almost complete binary tree is a binary tree satisfying the following conditions:

- All internal vertices, with at most one exception, have exactly two children.
- All vertices having less than two children are on the deepest two levels of the tree.
- In the tree's graphical representation, the vertices on the deepest level are filled up from left to right.

Definition 11

A binary tree is complete if all its leaves have the same depth and all levels are filled with vertices to capacity. An almost complete binary tree is a binary tree satisfying the following conditions:

- All internal vertices, with at most one exception, have exactly two children.
- All vertices having less than two children are on the deepest two levels of the tree.
- In the tree's graphical representation, the vertices on the deepest level are filled up from left to right.

Definition 11

A binary tree is complete if all its leaves have the same depth and all levels are filled with vertices to capacity. An almost complete binary tree is a binary tree satisfying the following conditions:

- All internal vertices, with at most one exception, have exactly two children.
- All vertices having less than two children are on the deepest two levels of the tree.
- In the tree's graphical representation, the vertices on the deepest level are filled up from left to right.

Definition 12

A heap is an almost complete binary tree whose vertices are annotated with key values such that the heap condition is satisfied in each vertex v: The key value stored in v is at most as large as the key values stored in v's children.
Hence, the root of the heap is annotated with a minimum key value. And each path of vertices from the root to a leaf is annotated with increasing sequence of keys.

A data structure is a structued method of storing data elements (typically permitting efficient access to its contents), along with a set of operations that allow access to the data structure and manipulation of the structure in such a way that the storage organization remains intact. We shall now see how to define a set of operations on heaps which will help us write down the HeapSort algrithm in just 4 lines of code.

Definition 12

A heap is an almost complete binary tree whose vertices are annotated with key values such that the heap condition is satisfied in each vertex v: The key value stored in v is at most as large as the key values stored in v's children.
Hence, the root of the heap is annotated with a minimum key value. And each path of vertices from the root to a leaf is annotated with increasing sequence of keys.

A data structure is a structued method of storing data elements (typically permitting efficient access to its contents), along with a set of operations that allow access to the data structure and manipulation of the structure in such a way that the storage organization remains intact. We shall now see how to define a set of operations on heaps which will help us write down the HeapSort algrithm in just 4 lines of code.