Insertion Sort

Idea: by Ex.

Given the following sequence to be sorted

34 8 64 51 32 21

When the elements 1, … p are sorted, then the next element, p+1 is inserted within these to the right place after some comparisons.

We take the first element: 34 is sorted

We take the second element: 8<34 → this means that we are going to replace them

8 34 — all the element remained are unchanged

.
.
.

We take the p^{th} element:

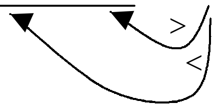
8 34 64 — compare the $p\text{-}1^{th}$ element

we leave 64 in place

8 34 64 — are already sorted

We take the $p+1^{th}$ element

8 34 64 51 — compare with the largest

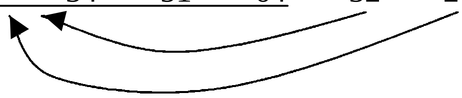


we move 1 position further and compare

we swap the elements

8 34 51 64 — we get a new element in the sequence sorted

8 34 51 64 32 21



8 21 32 34 51 64

Code:

```
FOR     p = 2   TO      N
        j = p   tmp=a(p)
        WHILE           tmp < a(j-1)
                        a(j) = a(j-1)           {swap}
                        j = j-1
        ENDWHILE
                        a(j) = tmp
ENDFOR
```

Bubble Sorting

$O(N^2)$ – Insertion Sort (we can see from the code)
$\Theta(N^2)$ tight bound
(number of comparisons, number of swaps)

when elements are „almost" sorted (many of them are already in the correct order) the Insertion Sort algorithm needs just a relatively small number of comparisons and swaps. Then it is effective.

Shell Sort

Idea: divide the sequence into groups

1. induces great order into the initial random sequence relatively fast
2. then apply insertion sort to get final order

to compare element which are relatively great distance from each other
1st step: half the seqiunece (4 elements/group), and compare corresponding elements (four groups, two elements/group)

7 1 3 5 | 0 9 2 10

Each groups contains 2 elements:

7	1	3	5	
0	9	2	10	→ 4 groups with two elements /group
↑	↑	↑	↑	
1st group	2nd group	3rd	4th	

Now apply insertion sort for each groups:

↓	↓	↓	↓
0	1	2	5
7	9	3	10

⇓

0 1 | 2 5 | 7 9 | 3 10 our new sequence after the first step

now apply the same idea again: divide the first half into 2 subhalves
form again the groups

0	1
2	5
7	9
3	10

→ 2 groups with 4 elements/group

Apply now insertion sort for each groups. We get

0	1
2	5
3	9
7	10

→ we are going to have 1 swap: 2 and 3 are swapt

0 | 1 | 2 | 5 ‖ 3 | 9 | 7 | 10 after the second step

We apply the same again : we divide the subhalves into subhalves until every subhalf contains just 1 element . So we get one group that contains every elements.

We get:

0 1 2 5 3 9 7 10 → now apply insertion sort

↓

effective, because this is the „almost" sorted sequence

↓

so insertion sort will be very fast

Code:

```
FOR ( gap = N/2; gap > 0; gap/=2 )          {N/2: the length of the half; gap is going to be halfed}
        FOR ( i = gap; i < N; i++ )
                FOR ( j = i – gap; j >= 0 && a [ j ] > a [ j + gap ]; j - = gap )
                {       temp = a [j ];
                        a[ j ] = a [ j + gap ];
                        a [ j + gap ] = temp;
                        …       }
```

proved very fast in practice
$O(N\log N) < O(N^2)$ the complexity

for certain values – better choice – of the gap $\Theta(N^{1,5})$, $O(N^{1,25})$

the difficult problem is evaluation of its complexity
not final

$\Theta(N) <$ complexity $< \Theta(N^2)$

A better choice of the gap:
…, 1093, 364, 121, 40, 13, 4, 1
$a_1, a_2, \ldots, a_n$ to be sorted

```
        find the largest first

        gap = 1
        REPEAT
                gap = 3* gap +1
        UNTIL gap > n
REPEAT
        gap = gap DIV 3
             { DO   INSERTION SORT …  }
             {      with              }

UNTIL gap = 1
END
```

complexity: $\Theta(N^{1,25})$

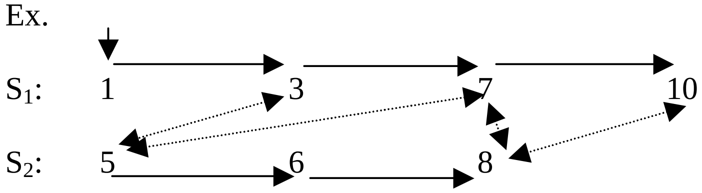
Merge Sort

idea: „divide et impera”
What does merging mean?
given two <u>sorted</u> sequences,
produce one sequence with the property:

- contains every element of the two given sequences, and
- sorted the same way (descendingly / ascendingly)

Ex. working with files

Ex.



may have different lengthes

S: 1 3 5 6 7 8 10

we take the first elements
simple, but very difficult to code in practice

Code Merging

- main memory S_1, S_2, S
- files on the disk

attention: when the end of a sequence is detected

Merge Sort Ex.

34 56 78 12 45 3 99 23

1. divide phase

34 56 78 12 | 45 3 99 23

2. we divide every half to subhalves

34 56 | 78 12 | 45 3 | 99 23

3. we divide every subhalf again to subhalves:

34 | 56 | 78 12 | 45 3 | 99 | 23

S_1 S_2

divide

Merge algorithm

34 56 | 12 78 | 3 45 | 23 99

we apply the same Merge alg.

12 34 56 78 | 3 23 46 99

finally we apply Merge alg. again

3 12 23 34 45 56 78 99

conquer

the final order

Code

Merge Sort

```
mid = (first + last )
mergesort (first, mid )
mergesort (mid + 1, last )
merge
```

Merge Sort:

- recursive algorithm
- complexity

  T (N) = 2 T (N/2) + N

  telescoping

  T (N) = O (NlogN)

very tricky another algorithm

HEAPSORT

heap: an array represented as a binary tree obeying the <u>heap property</u>

every node X (in tree)

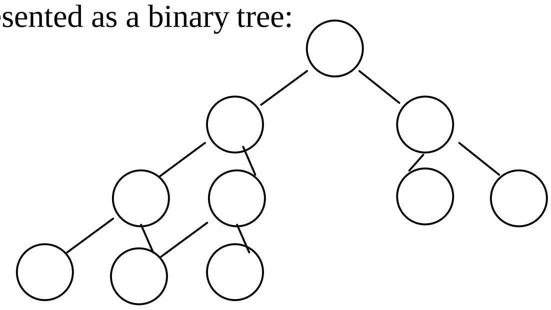
the key value Key (Parent(X)) ≥ Key (X)

≥

Ex. for a heap:

13 21 16 24 31 19 68 65 26 32

represented as a binary tree:



the heaps property is satisfied, because every node is less or equal to their children

<u>1. very interesting property</u> of the heap from a programming point of view:

every element i:

left child 2i — right child 2i+1 → we need no pointers! we simply need +, *

this yields a fast algorithm

<u>2. property:</u> A binary tree with height h has between 2^h and 2^{h+1}-1 nodes

justification: for this property:

h = 0 N = 1 just contains the root

$1 = 2^0 = 2^h$

h = 1 N = 3 (root + at least one child)

$3 = 2^1 + 1 = 2^h + 1 = 2^h + (2^1 - 1)$

h = 2 N = 7 $7 = 2^h + 3 = 2^h + (2^2-1)$

N ≤ number of the leaves + number of the nodes of the previous tree

$N \leq 2^h + 2^h - 1$ ($= 2*2^h = 2^{h+1}$)

the max. number of nodes: $2^{h+1} - 1$

the min.number of nodes: (if every level contains just 1 node) 2^h

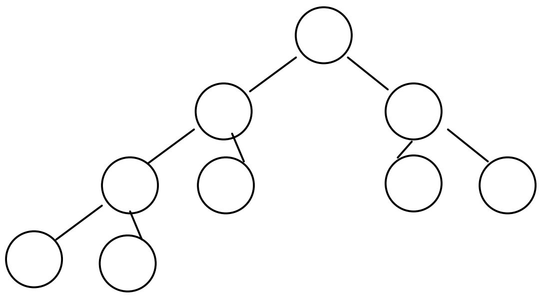
$2^h \leq N \leq 2^{h+1} - 1$

⇓

$h = \lfloor \log N \rfloor = O(\log N)$

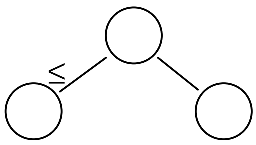
Heapify:

Given an arbitrary array —heapify→ make it a heap!

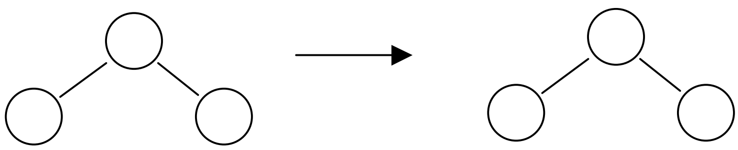


Does it satisfy the heap property?

i = 1 • 1st level
i = 2 •
i = 3 → heap property is not satisfied
we have to swap the element according to the property



•



.
.
. we continue analysing

Heapify code:

```
Heapify (A, i)
    MAX = max (A (i), Left (A (i)), Right (A (i)))
        IF MAX ≠ A (i) swap (A(i), max (Left(A(i)), Right (A(i))))
    Heapify (A, MAX)
    { recursively }
```

the selection of max. (time) : Θ (1)
the heapify : Θ (h)

$\Theta (1) + \Theta (h) = \Theta (h) = O (h) = O (\log N)$

Convert an array into a heap

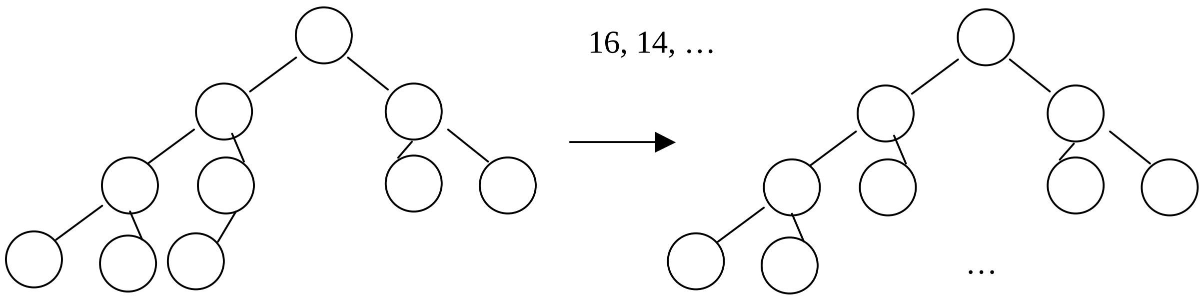
Build_Heap (A)
FOR i = $\lfloor$ length (A) / 2 $\rfloor$ DOWNTO 1 DO Heapify (A, i)

O (N) * O (logN) = O (NlogN) time

```
HeapSort ( A )
        Build_Heap ( A )                                  { O (NlogN) }
                FOR i = length (A) DOWNTO 2               { n-1 calls     constant }
                        DO swap ( A(1), A(i))
                                Heapsize (A) = Heapsize (A) – 1
                                Heapify (A,1)             { O (logN ) }
```

$O(N \log N) + (N-1) * O(\log N) = O(N \log N)$

after executing Build_Heap we have a heap: we will get the array in ascending order



Bubble Sort: $O(N^2)$
Insertion Sort: $O(N^2)$
Quick Sort: $O(N \log N)$
Merge Sort: $O(N \log N)$
Shell Sort:: $O(N^{1,5})$
Heap Sort: $O(N \log N)$
in average

$\forall$S comparisons $\Omega(N \log N)$ lower bound
$\forall$: internal sorting methods the elements to be sorted are all in the main memory.
When sorting in real time on-line : different algorithm would be needed (so sorting on disk diff. algorithm would be required, for example Merge can de used)

Special cases:

pre-defined requirements, some certain properties

↓

only then: sorting algorithm : LINEAR TIME