Sorting

Sorting

Input: n number
Output: The n numbers in sorted order

Example:

$$6,2,9,4,5,1,4,3 \rightarrow 1,2,3,4,4,5,6,9$$

Sorting

Input: n number
Output: The n numbers in sorted order

Example:

$$6,2,9,4,5,1,4,3 \rightarrow 1,2,3,4,4,5,6,9$$

Many tasks are faster with sorted data (think of dictionaries, address lists in phones, etc.). This applies to both humans and computers. Sorting is often a building block in algorithms for other problems.

Sorting

Input: n number
Output: The n numbers in sorted order

Example:

6,2,9,4,5,1,4,3→1,2,3,4,4,5,6,9

Many tasks are faster with sorted data (think of dictionaries, address lists in phones, etc.). This applies to both humans and computers. Sorting is often a building block in algorithms for other problems

$\boxed{\text{Sorting is a fundamental and central task.}}$

Sorting

Input: n number
Output: The n numbers in sorted order

Example:

$$6,2,9,4,5,1,4,3 \rightarrow 1,2,3,4,4,5,6,9$$

Many tasks are faster with sorted data (think of dictionaries, address lists in phones, etc.). This applies to both humans and computers. Sorting is often a building block in algorithms for other problems.

Sorting is a fundamental and central task.

Many algorithms have been developed: Insertionsort, Selectionsort, Bubblesort, Mergesort, Quicksort, Heapsort, Radixsort, Countingsort, . . .

We will meet all of the above in this course.

Sorting

Comments:

- Sorted order can be either ascending or descending. In this course, we will always use ascending (more precisely: non-decreasing). If one needs to sort in descending order, all comparisons should simply be reversed.

Sorting

Comments:

- Sorted order can be either ascending or descending. In this course, we will always use ascending (more precisely: non-decreasing). If one needs to sort in descending order, all comparisons should simply be reversed.
- We will assume that the input is in an array (Java) / a list (Python).

Sorting

Comments:

- ► Sorted order can be either ascending or descending. In this course, we will always use ascending (more precisely: non-decreasing). If one needs to sort in descending order, all comparisons should simply be reversed.

- ► We will assume that the input is in an array (Java) / a list (Python).

- ► Elements are often sorted based on a sorting key along with additional information. The sorting key can be a number or anything that can be compared (e.g., strings/words). In this course, we will simply show elements as pure numbers.

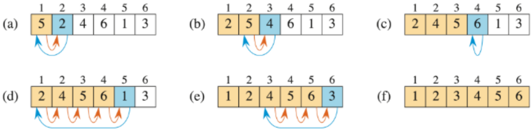
Insertion Sort

Used by many when sorting a hand of cards:

Insertion Sort

Used by many when sorting a hand of cards:

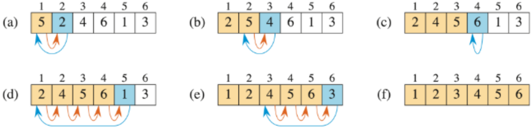
Same idea performed on numbers in an array:



Insertion Sort

Used by many when sorting a hand of cards:

Same idea performed on numbers in an array:

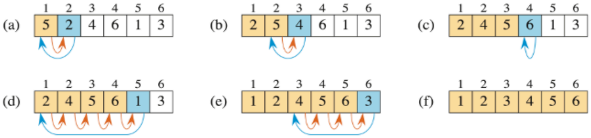


Argument for correctness: The yellow part of the array is always sorted. This part is extended by one all the time (⇒ the algorithm stops, and when it stops all elements are sorted).

Insertion Sort

As pseudo-code:

```
INSERTION-SORT(A, n)
for i = 2 to n                          i = the current card. n = amount of card we have
    key = A[i]                          value of the element i
    // Insert A[i] into the sorted subarray A[1 : i – 1].     length of the sorted sub array (start at 1, since 2-1 = 1)
    j = i – 1
    while j > 0 and A[j] > key          The while loop is the red arrow
        A[j + 1] = A[j]
        j = j – 1
    A[j + 1] = key                      blue arrow
```



Runtime Analysis of Insertion Sort

How many times each line is executed

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$\quad key = A[i]$	c_2	$n-1$
$\quad$ // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
$\quad j = i - 1$	c_4	$n-1$
$\quad$ **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$\quad\quad A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
$\quad\quad j = j - 1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
$\quad A[j+1] = key$	c_8	$n-1$

because the last time you evaluate the for loop, it will be false

Here is t_i the number of times the test in the inner **while**-loop is executed. t_i *-1* is how many times this loop runs (which is how many elements the *i-th* element has to pass during insertion).

Note that: $1 \leq t_i \leq i$. Set $c = c_1 + c_2 + \cdots + c_8$.

Runtime Analysis of Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$\quad key = A[i]$	c_2	$n-1$
$\quad$ // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
$\quad j = i - 1$	c_4	$n-1$
$\quad$ **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$\quad\quad A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
$\quad\quad j = j - 1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
$\quad A[j+1] = key$	c_8	$n-1$

Here is t_i the number of times the test in the inner **while**-loop is executed. $t_i - 1$ is how many times this loop runs (which is how many elements the *i-th* element has to pass during insertion).

Note that: $1 \leq t_i \leq i$. Set $c = c_1 + c_2 + \cdots + c_8$.

Best case: $t_i = 1$ for all i. Total time $\leq c.n$.

Runtime Analysis of Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $i = 2$ **to** n	c_1	n
$key = A[i]$	c_2	$n - 1$
// Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n - 1$
$j = i - 1$	c_4	$n - 1$
while $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
$A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
$j = j - 1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
$A[j+1] = key$	c_8	$n - 1$

Here is t_i the number of times the test in the inner **while**-loop is executed.
t_i *-1* is how many times this loop runs (which is how many elements the *i-th* element has to pass during insertion).
Note that: $1 \leq t_i \leq i$. Set $c = c_1 + c_2 + \cdots + c_8$.

Best case: $t_i = 1$ for all i. Total time $\leq$ c.n.
Worst case: $t_i = i$ for all i. Total time $\leq c \cdot n^2$, since

$$\sum_{i=2}^{n} i \leq (1 + 2 + 3 + \cdots + n) = \frac{(n+1)n}{2} = \frac{n^2 + n}{2} \leq \frac{2n^2}{2} = n^2.$$

The biggest term (here n^2) determine the rate of growth

Selection Sort

Another simple and natural sorting algorithm:

inList = input
outList = empty list
While *inList* not empty:
  find the smallest element x in *inList*
  move x from *inList* to the end of *outList*

Selection Sort

Another simple and natural sorting algorithm:

inList = input
outList = empty list
While *inList* not empty:
    find the smallest element x in *inList*
    move x from *inList* to the end of *outList*

Clearly **correct**, i.e., gives a sorted output (each element that is extracted must be at least as large as the previous one).

Selection Sort

Another simple and natural sorting algorithm:

inList = input
outList = empty list
While *inList* not empty:
    find the smallest element x in *inList*
    move x from *inList* to the end of *outList*

Clearly **correct**, i.e., gives a sorted output (each element that is extracted must be at least as large as the previous one).

Run time?

Selection Sort

Like picking the smallest of the cars

Another simple and natural sorting algorithm:

inList = input
outList = empty list
While *inList* not empty:
find the smallest element x in *inList*
move x from *inList* to the end of *outList*

Clearly **correct**, i.e., gives a sorted output (each element that is extracted must be at least as large as the previous one).

Run time?

Every time we have to check all the cards

In total, the smallest element in the input list is found n times.

A simple method to find the smallest element is linear search, which looks at each remaining element once.

Thus, the time becomes $\leq c.(n + (n-1) + (n-2) + \cdots + 1) \leq c.n^2$.

Merge

Input: Two sorted rows A and B
Output: The same elements in one sorted row C

Example:

A=2,4,5,7,8
B=1,2,3,6
C=1,2,2,3,4,5,6,7,8

Merge

Input: Two sorted rows A and B
Output: The same elements in one sorted row C

Example:

A=2,4,5,7,8
B=1,2,3,6
C=1,2,2,3,4,5,6,7,8

We can of course sort $A \cup B$.

Merge

Input: Two sorted rows A and B
Output: The same elements in one sorted row C

Example:

A=2,4,5,7,8
B=1,2,3,6
C=1,2,2,3,4,5,6,7,8

We can of course sort $A \cup B$.

But it is faster to **merge**:

Repeat:
Move the smaller of the two front elements

Merge

= uneven

Input: Two sorted rows A and B
Output: The same elements in one sorted row C

Example:

A=2,4,5,7,8
B=1,2,3,6
C=1,2,2,3,4,5,6,7,8

We can of course sort $A \cup B$.

But it is faster to **merge**:

Repeat:
Move the smaller of the two front elements

Running time: $\leq c \cdot n$ where n = total number of elements in $A \cup B$.

Merge

Input: Two sorted rows A and B
Output: The same elements in one sorted row C

Example:

A=2,4,5,7,8
B=1,2,3,6
C=1,2,2,3,4,5,6,7,8

We can of course sort $A \cup B$.

But it is faster to **merge**:

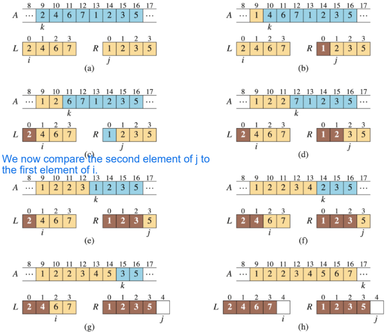
Repeat:
Move the smaller of the two front elements

Running time: $\leq c \cdot n$ where n = total number of elements in $A \cup B$.
Correctness: Merge can be seen as a version of Selection sort that takes advantage of A and B being sorted, so the smallest in (the rest of) A and B can be found by looking only at the first two in A and B, which takes constant time.

Example of merge

Here, it is assumed that the two input lists are neighboring parts of the same array/ list.A, namely A[p . . . q] and A[q+1…r]. They are first moved to L and R.

(**Note** that in the book A[p . . . q] is equal to the elements A[p],A[p+1],. . . , A[q], which is one more element than almost the same notation in Python.)



We now compare the second element of j to the first element of i.

You copy the original list and split up th the list,
and compare the first element
of the first splittet list to the second splitted list.
The lowest of the values go in front of the original list.

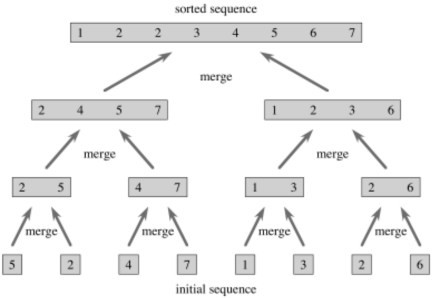
Pseudo-code for Merge

```
MERGE(A, p, q, r)
n_L = q - p + 1               // length of A[p : q]
n_R = r - q                   // length of A[q + 1 : r]
let L[0 : n_L - 1] and R[0 : n_R - 1] be new arrays
for i = 0 to n_L - 1          // copy A[p : q] into L[0 : n_L - 1]
    L[i] = A[p + i]
for j = 0 to n_R - 1          // copy A[q + 1 : r] into R[0 : n_R - 1]
    R[j] = A[q + j + 1]
i = 0                         // i indexes the smallest remaining element in L
j = 0                         // j indexes the smallest remaining element in R
k = p                         // k indexes the location in A to fill
// As long as each of the arrays L and R contains an unmerged element,
//     copy the smallest unmerged element back into A[p : r].
while i < n_L and j < n_R
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
    k = k + 1
// Having gone through one of L and R entirely, copy the
//     remainder of the other to the end of A[p : r].
while i < n_L
    A[k] = L[i]
    i = i + 1
    k = k + 1
while j < n_R
    A[k] = R[j]
    j = j + 1
    k = k + 1
```

what does the two whiles loops?
When the list is split, but the two list are with different length

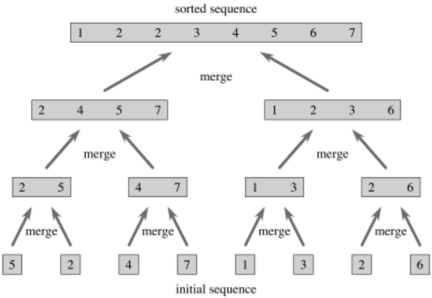
Merge sort

Mergesort: Build longer and longer sorted parts of the input by repeatedly using merge.



Merge sort

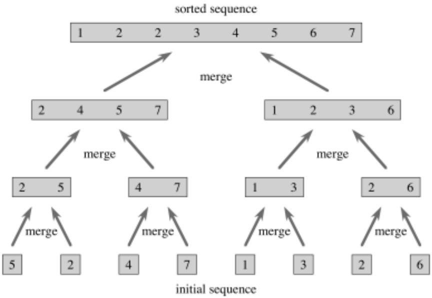
Mergesort: Build longer and longer sorted parts of the input by repeatedly using merge.



Time:

Merge sort

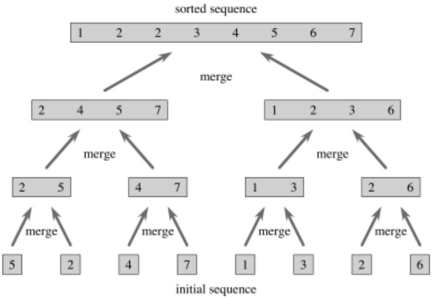
Mergesort: Build longer and longer sorted parts of the input by repeatedly using merge.



Time: Each merge takes at most **$c \cdot n_1$** time when **n_1** is the number of elements that are merged.

Merge sort

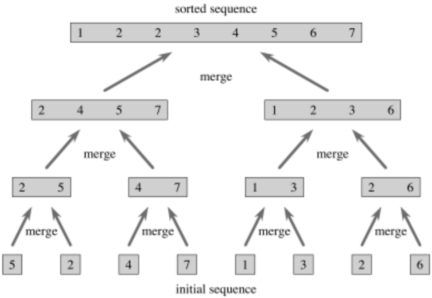
Mergesort: Build longer and longer sorted parts of the input by repeatedly using merge.



Time: Each merge takes at most **$c \cdot n_1$** time when **n_1** is the number of elements that are merged. So all merge operations in one layer together take at most $c. (n_1 + n_2 + \ldots) = c \,.\, n$ This applies to all layers.

Merge sort

Mergesort: Build longer and longer sorted parts of the input by repeatedly using merge.



log_x(y) = b <=> x^b=y

Time: Each merge takes at most **$c \cdot n_1$** time when **n_1** is the number of elements that are merged. So all merge operations in one layer together take at most $c. (n_1 + n_2 + \dots) = c . n$ This applies to all layers. There are a total of $\log_2 n$ layers so that the total time is at most $c.n.\log_2 .n.$

C = cost = specifc amount of time it needs to run every statement in the code

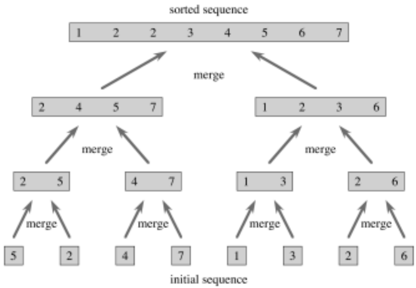
Merge sort

Why are there $\log_2 n$ merge layers ?

Merge sort

Why are there $\log_2 n$ merge layers ?

Number of sorted lists after k merge layers (assuming n is a power of 2):

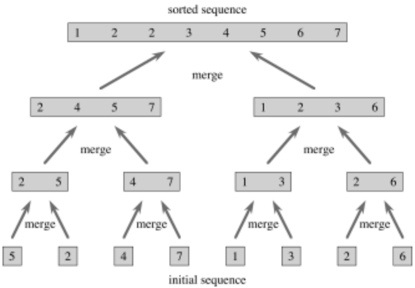


k	Antal lister	
$\vdots$	$\vdots$	
k	$n/2^k$	
$\vdots$	$\vdots$	
3	$n/2^3$	
2	$n/2^2$	
1	$n/2$	second upper level on the right
0	n	upper level on the right

Merge sort

Why are there $\log_2 n$ merge layers ?

Number of sorted lists after k merge layers (assuming n is a power of 2):



k	Antal lister
$\vdots$	$\vdots$
k	$n/2^k$
$\vdots$	$\vdots$
3	$n/2^3$
2	$n/2^2$
1	$n/2$
0	n

The algorithm stops when there is one sorted list.

$$n/2^k = 1 \quad \Leftrightarrow \quad n = 2^k \quad \Leftrightarrow \quad \log_2 n = k$$

No matter what we need to divide and conque, so the best case is equal to the worst case

Merge sort if n is not a power of 2? = uneven

The algorithm merges as many pairs as possible in each layer, and there may be one list that is not merged (this list is carried over to the next layer).

For example, 12 lists become 12/2 = 6 lists while 13 (= 12 + 1) lists become 12/2 + 1 = 6 + 1 = 7 lists.

Merge sort if n is not a power of 2?

The algorithm merges as many pairs as possible in each layer, and there may be one list that is not merged (this list is carried over to the next layer).

For example, 12 lists become 12/2 = 6 lists while 13 (= 12 + 1) lists become 12/2 + 1 = 6 + 1 = 7 lists.

In general: If there is x lists before a merge layer, there are $\lceil x/2 \rceil$ lists after.

Merge sort if n is not a power of 2?

The algorithm merges as many pairs as possible in each layer, and there may be one list that is not merged (this list is carried over to the next layer).

For example, 12 lists become 12/2 = 6 lists while 13 (= 12 + 1) lists become 12/2 + 1 = 6 + 1 = 7 lists.

In general: If there is x lists before a merge layer, there are $\lceil x/2 \rceil$ lists after.

Look at two input sizes n and n′, with n≤n′. Since $\lceil x/2 \rceil$ is an increasing function of x, the number of lists in each layer cannot be less than n′ than for n Therefore, the number of layers (before the algorithm reaches one list) cannot be less than n′ than for n.

Merge sort if n is not a power of 2? = uneven

The algorithm merges as many pairs as possible in each layer, and there may be one list that is not merged (this list is carried over to the next layer).

For example, 12 lists become 12/2 = 6 lists while 13 (= 12 + 1) lists become 12/2 + 1 = 6 + 1 = 7 lists.

In general: If there is x lists before a merge layer, there are $\lceil x/2 \rceil$ lists after.

Look at two input sizes n and n′, with n≤n′. Since $\lceil x/2 \rceil$ is an increasing function of x, the number of lists in each layer cannot be less than n′ than for n Therefore, the number of layers (before the algorithm reaches one list) cannot be less than n′ than for n.

Set n to the smallest power of two that is greater than or equal to n. There are exactly $\log_2 n'$ layers for n', and thus at most as many layers for n.

Conversely, it is easy to see that for $n=2^k+1$ is there $k+1 = \lceil \log_2 n \rceil$ layer. So there is $\lceil \log_2 n \rceil$ layer for general n.

n	7	$8 = 2^3$	9	10	11	12	13	14	15	$16 = 2^4$	17
$\log_2(n)$	2.807	3	3.169	3.321	3.459	3.584	3.700	3.807	3.906	4	4.087
Antal lag	3	3	4	4	4	4	4	4	4	4	5

Merge sort

Mergesort as pseudocode, in a variant formulated with recursion:

```
MERGE-SORT(A, p, r)
if p ≥ r                          // zero or one element?
    return
q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
MERGE-SORT(A, p, q)               // recursively sort A[p : q]
MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
// Merge A[p : q] and A[q + 1 : r] into A[p : r].
MERGE(A, p, q, r)
```

A call to Merge-Sort(A, p, r) is responsible for sorting the elements in A[p...r] in sorted order.

The first call is Merge-Sort(A, 1, n), which is tasked with sorting the entire A.

A call to Merge(A, p, q, r) merges the two sorted subarrays/lists A[p...q] and A[q + 1...r] together into A[p...r].

Merge sort

Example of execution: