Politecnico di Torino

Iterative Quadratic Sorting Algorithms

Paolo Camurati

Edited by Josie E. Rodriguez

$O(n^2)$

Outline

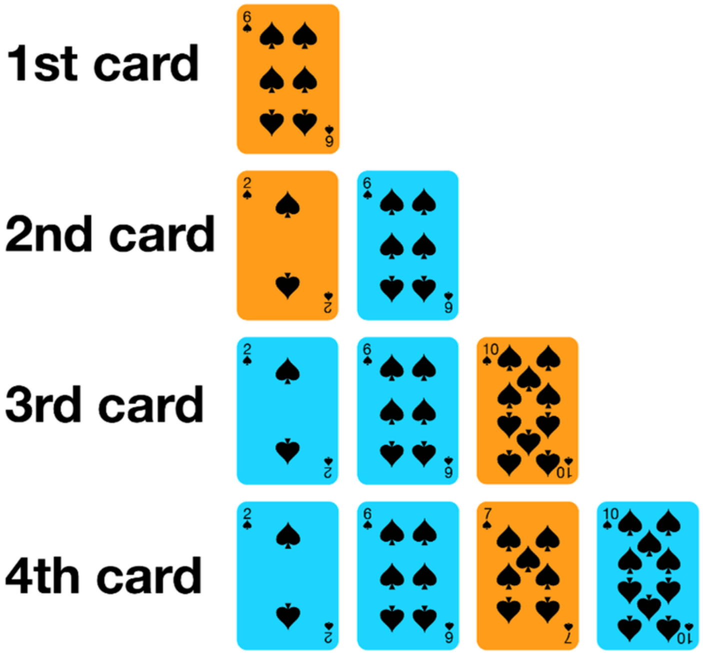
- Insertion sorting
- Bubble/exchange Sorting
- Selection sorting
- Shell sorting

Insertion Sort

Sorting algorithm that builds the final sorted **array** (or list) one item at a time by **comparisons**

Insertion Sort

- Start with one card in your hand,
- **Pick** the next card and insert it into its proper **sorted** order,
- Repeat previous step for all cards.



Insertion Sort

Let **N** integers be stored in an array **A** whose indices range from **i**=0 and **r**=**N**-1.

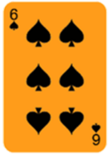
Conceptually array **A** consists of **2 subarrays**:

- ***the left one***: already sorted
- ***the right one***: not yet sorted

Initially the left subarray contains **1 item**, the right subarray contains **N-1** items.

An array with just **1 item** is sorted by definition.

1st card



Approach

Incremental paradigm:

- At each step, we expand the **already sorted** left subarray inserting an item taken from the still **unsorted right subarray**
- Insertion must guarantee that the left subarray remains sorted after inserting the item (*invariance of the sorting property*)
- **termination:** all items have been **properly inserted**, the **left subarray** contains **N** items, the **right subarray** is **empty**

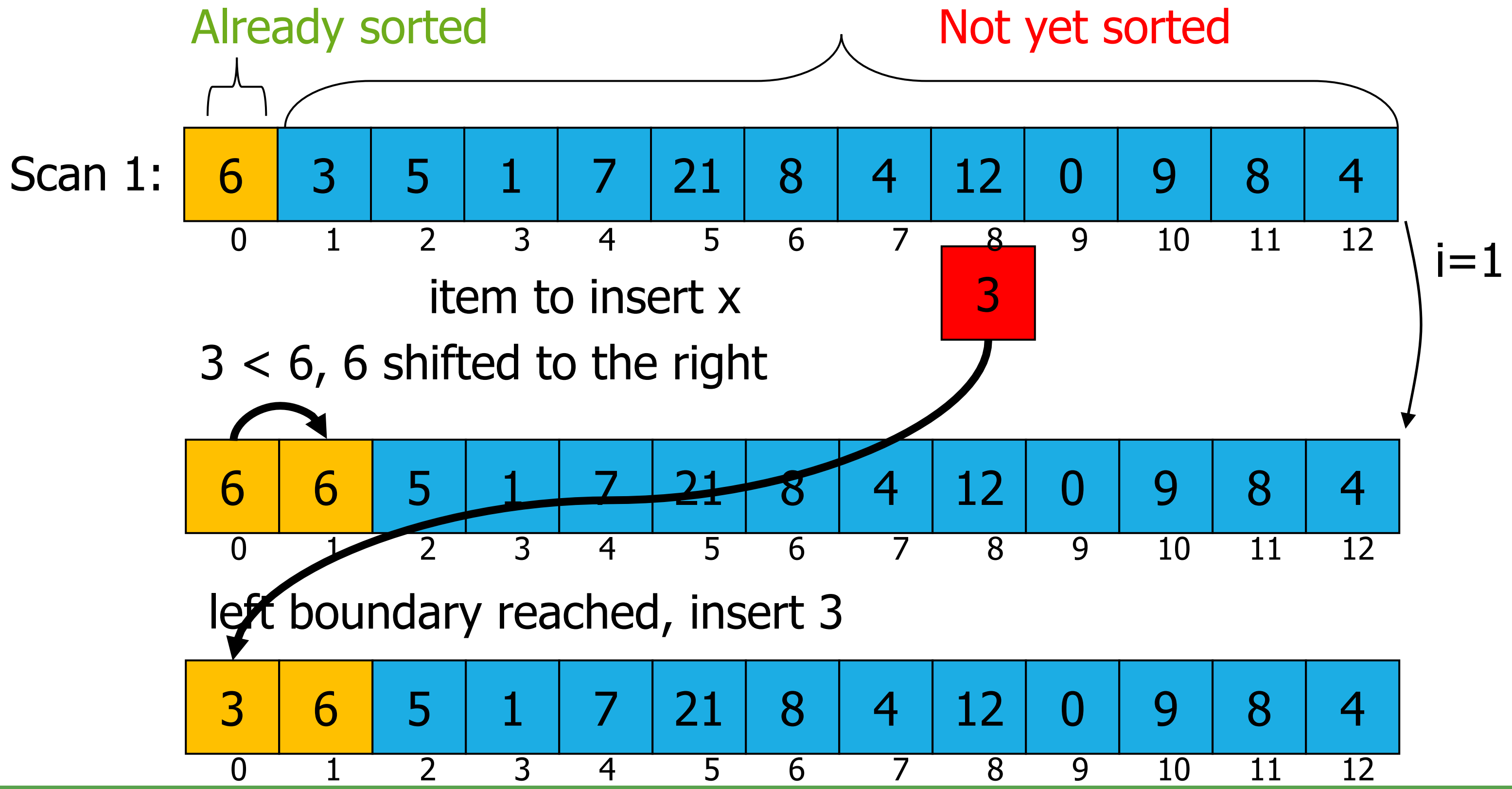
Step *i*: sorted insertion

At step *i* item **x** = $\mathbf{A_i}$ is stored at the correct position in the **left subarray:**

- the **left subarray** (already sorted and ranging from $\mathbf{A_{i-1}}$ downto $\mathbf{A_0}$) is **scanned** until an item is found such that $\mathbf{A_j} > \mathbf{A_i}$
- during the scan, items in the range from $\mathbf{A_j}$ to $\mathbf{A_{i-1}}$ are **shifted to the right** by **1** position

When the **loop** terminates, the correct position for inserting $\mathbf{A_i}$ has been found

Example



Already sorted

Not yet sorted

Scan 1:

6	3	5	1	7	21	8	4	12	0	9	8	4
0	1	2	3	4	5	6	7	8	9	10	11	12

item to insert x

3

i=1

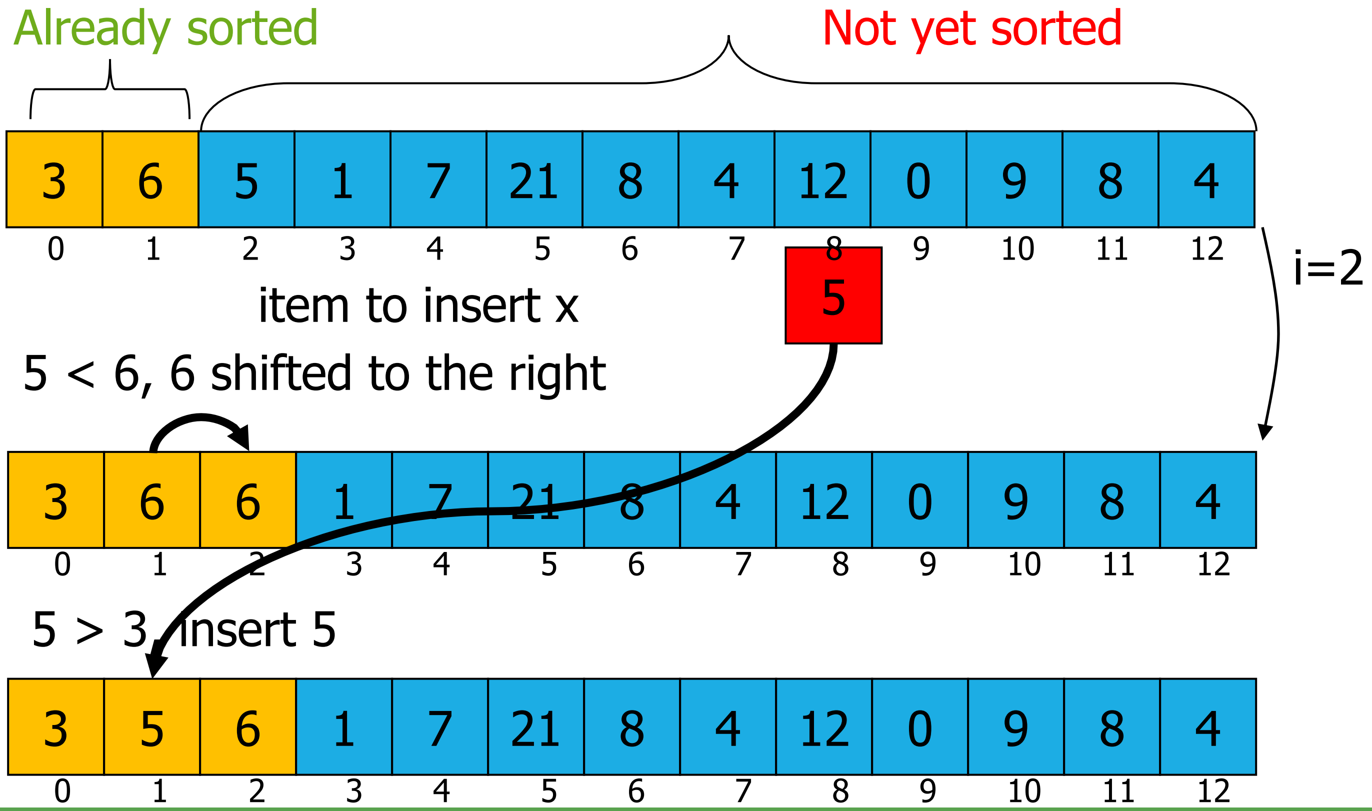
3 < 6, 6 shifted to the right

6	6	5	1	7	21	8	4	12	0	9	8	4
0	1	2	3	4	5	6	7	8	9	10	11	12

left boundary reached, insert 3

3	6	5	1	7	21	8	4	12	0	9	8	4
0	1	2	3	4	5	6	7	8	9	10	11	12

Example



Already sorted

Not yet sorted

3	6	5	1	7	21	8	4	12	0	9	8	4
0	1	2	3	4	5	6	7	8	9	10	11	12

i=2

item to insert x

5

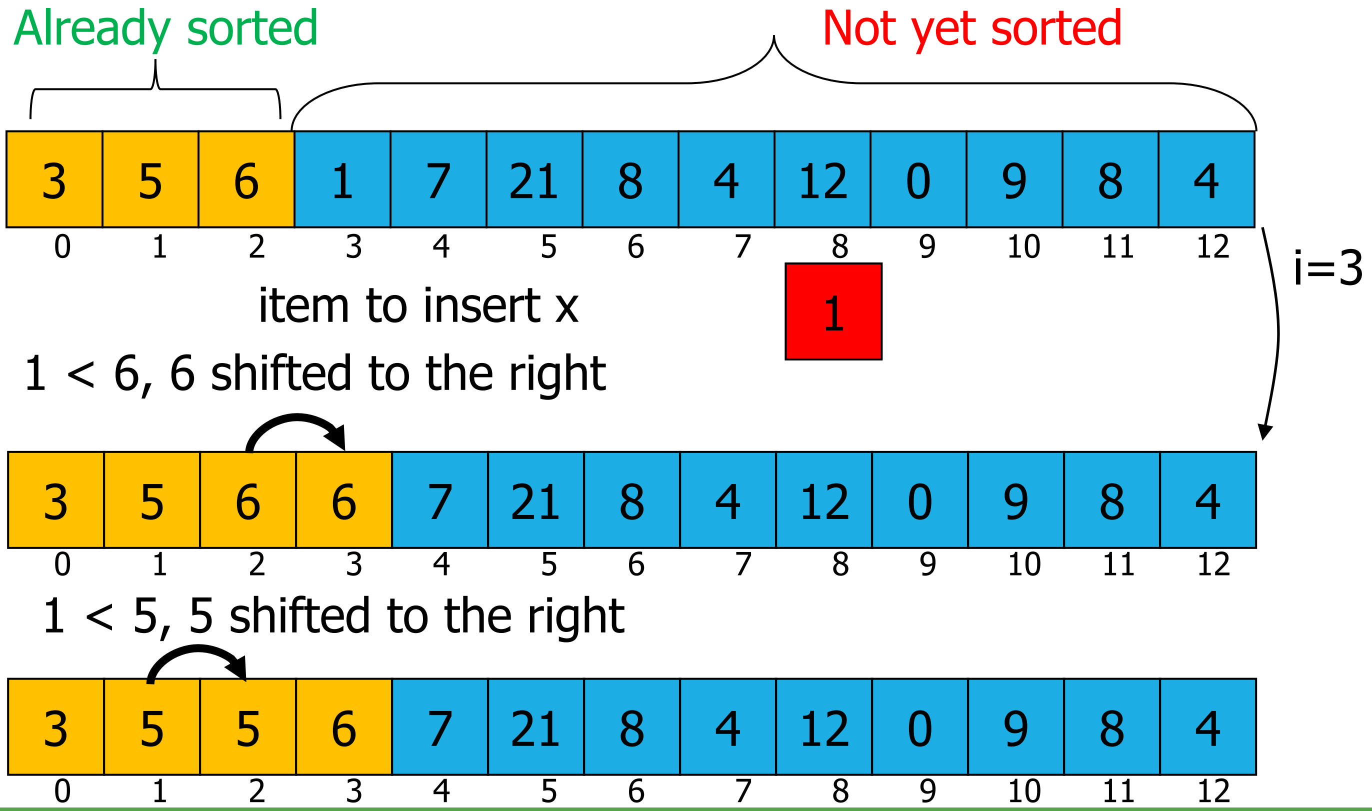
5 < 6, 6 shifted to the right

3	6	6	1	7	21	8	4	12	0	9	8	4
0	1	2	3	4	5	6	7	8	9	10	11	12

5 > 3, insert 5

3	5	6	1	7	21	8	4	12	0	9	8	4
0	1	2	3	4	5	6	7	8	9	10	11	12

Example



Already sorted

Not yet sorted

3	5	6	1	7	21	8	4	12	0	9	8	4
0	1	2	3	4	5	6	7	8	9	10	11	12

i=3

item to insert x 1

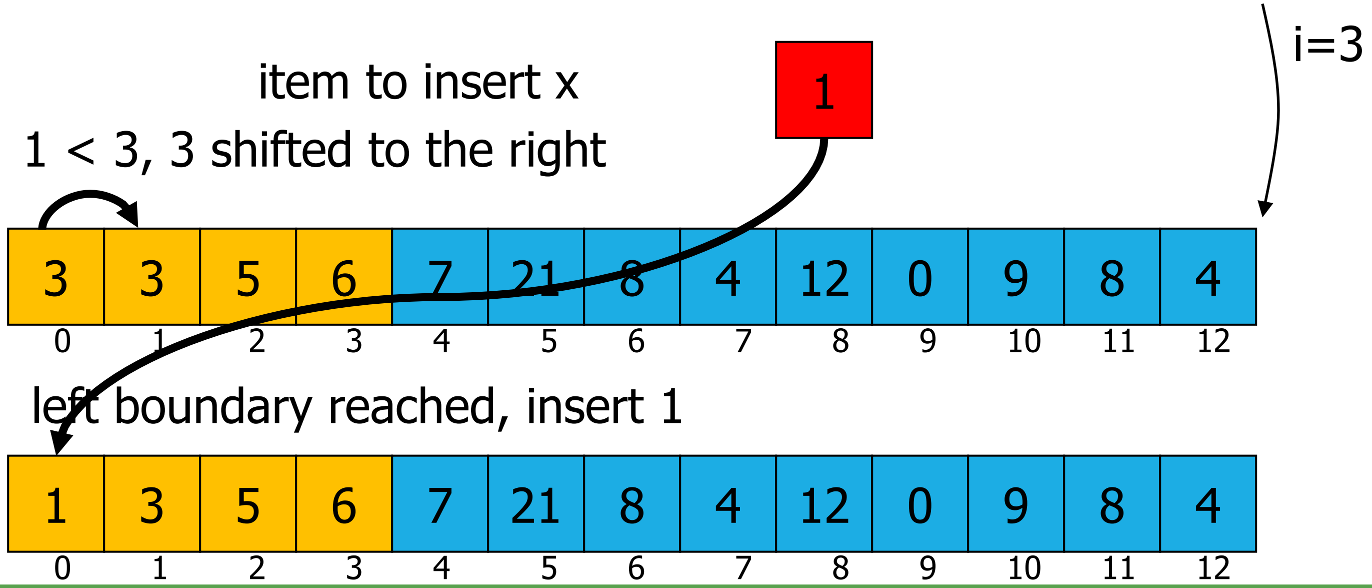
1 < 6, 6 shifted to the right

3	5	6	6	7	21	8	4	12	0	9	8	4
0	1	2	3	4	5	6	7	8	9	10	11	12

1 < 5, 5 shifted to the right

3	5	5	6	7	21	8	4	12	0	9	8	4
0	1	2	3	4	5	6	7	8	9	10	11	12

Example



i=3

item to insert x

1

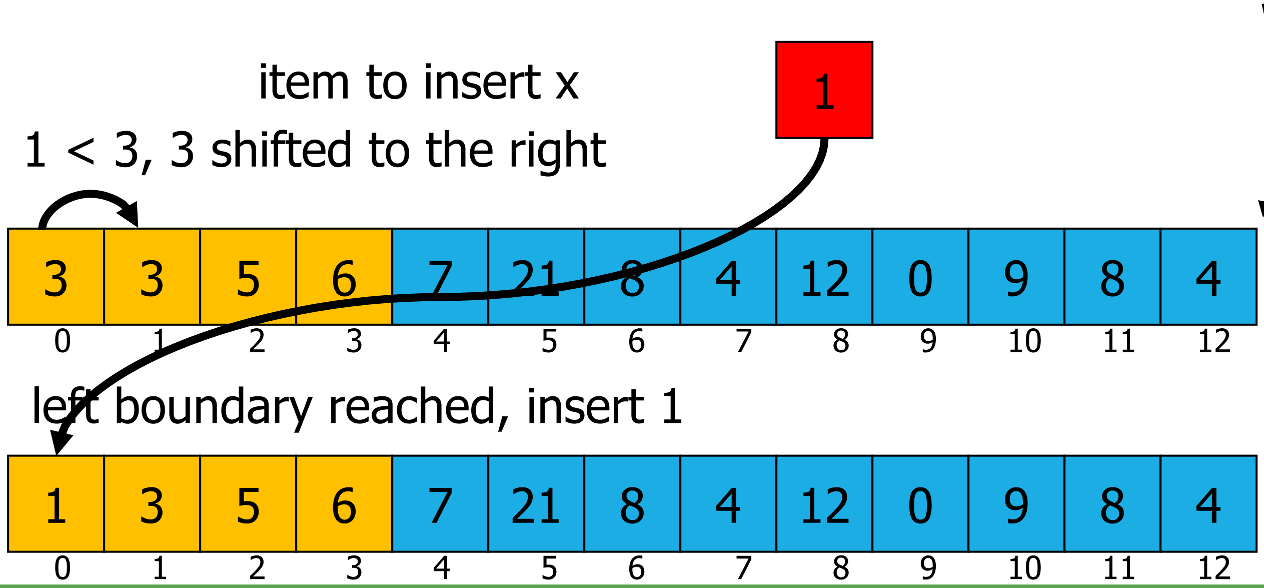
1 < 3, 3 shifted to the right

3	3	5	6	7	21	8	4	12	0	9	8	4
0	1	2	3	4	5	6	7	8	9	10	11	12

left boundary reached, insert 1

1	3	5	6	7	21	8	4	12	0	9	8	4
0	1	2	3	4	5	6	7	8	9	10	11	12

Example



i=3

item to insert x

1 < 3, 3 shifted to the right

3	3	5	6	7	21	8	4	12	0	9	8	4
0	1	2	3	4	5	6	7	8	9	10	11	12

left boundary reached, insert 1

1	3	5	6	7	21	8	4	12	0	9	8	4
0	1	2	3	4	5	6	7	8	9	10	11	12

How many times is the cycle executed?

Example

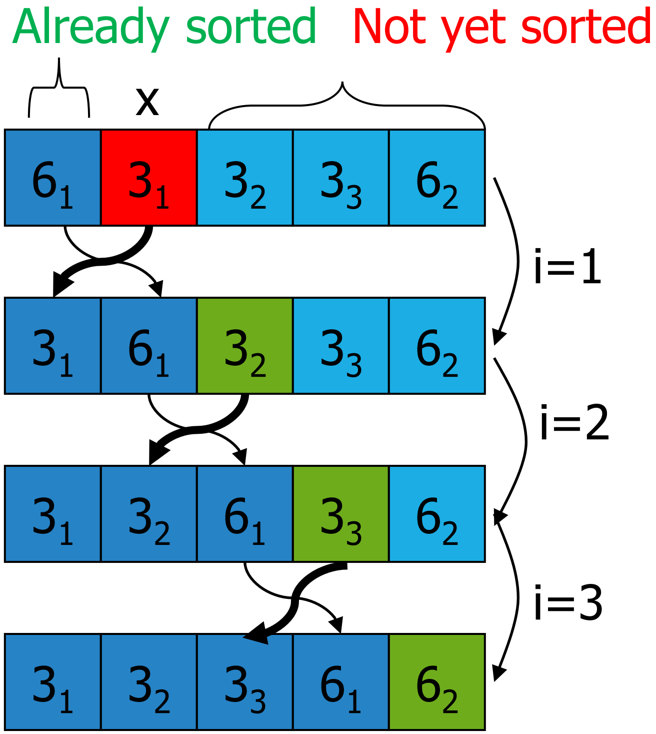
Example of functions implementing the insertion sorting algorithm

```
void InsertionSort(int A[], int N)
{
  int i, j, l=0, r=N-1, x;
   for (i = l+1; i <= r; i++) {
     x = A[i];
     j = i - 1;
     while (j >= l && x < A[j]){
       A[j+1] = A[j];
       j--;
    }
    A[j+1] = x;
  }
}
```

Is insertion sorting an algorithm or a procedure?

Insertion sort Features

- **in place:** used only array A and variable x
- **stable:** if the item to insert is a duplicate key, it can't jump over to the left a previous occurrence of the same key



Complexity Analysis of Insertion sort

Worst-case asymptotic analysis:

- Based on each **step** of the statements used
- Based on the high-level behaviour of the statements used

to estimate:

- The number of **comparisons**
- The number of **swaps**

Detailled Analysis of the # of comparisons

Assumption:

- All instructions have unit cost
- **i** starts from **1** and increases to **N-1** (for simplicity no use of **l** and **r**)

instructions # executions

```
for(i=1; i<N; i++) {
```

2N

- 2 instructions at each step (< and +)
- **N** executions taking into account the last one as well when the condition **i<N** fails

Detailled Analysis of the # of comparisons

instructions	# executions
`for(i=1; i<N; i++) {`	**2N**
`  x = A[i];`	N-1
`  j = i-1;`	N-1

N-1 executions only when condition i<N holds

Detailled Analysis of the # of comparisons

instructions	# executions
`for(i=1; i<N; i++) {`	**2N**
`x = A[i];`	N-1
`j = i-1;`	N-1
`while (j>=0 && x<A[j]){`	$2\sum_{j=2}^{N} j$

- 2 instructions at each step (>=and <)
- N-1 executions of the loop in the worst case (array already sorted in descending order)
- j operations at each loop execution taking into account as well the last one when condition j>=0 fails

Detailled Analysis of the # of comparisons

instructions	# executions
`for(i=1; i<N; i++) {`	**2N**
`  x = A[i];`	N-1
`  j = i-1;`	N-1
`  while (j>=0 && x<A[j]){`	$2\sum_{j=2}^{N} j$
`    A[j+1] = A[j];`	$\sum_{j=2}^{N}(j-1)$
`    j--;`	$\sum_{j=2}^{N}(j-1)$
`  }`	

- N-1 executions of the loop in the worst case (array already sorted in descending order)
- j-1 operations at each loop execution only when the loop condition holds

Detailled Analysis of the # of comparisons

instructions	# executions
`for(i=1; i<N; i++) {`	**2N**
`x = A[i];`	N-1
`j = i-1;`	N-1
`while (j>=0 && x<A[j]){`	$2\sum_{j=2}^{N} j$
`A[j+1] = A[j];`	$\sum_{j=2}^{N}(j-1)$
`j--;`	$\sum_{j=2}^{N}(j-1)$
`}`	
`A[j+1] = x;`	N - 1
`}`	

N-1 executions only when condition i<N holds

Detailled Analysis of the # of comparisons

Thus:

$T(N) = 2N + (N\text{-}1) + (N\text{-}1) + 2\sum_{j=2}^{N} j + \sum_{\mathrm{j}=2}^{\mathrm{N}}(j\text{-}1) + \sum_{\mathrm{j}=2}^{\mathrm{N}}(j\text{-}1) + (N\text{-}1)$

Recalling that:

$\sum_{\mathrm{j}=2}^{\mathrm{N}} \mathrm{j} = 2+3+\ldots N = N(N+1)/2\ \text{-}1$

$\sum_{\mathrm{j}=2}^{\mathrm{N}}(\mathrm{j}\text{-}1) = N(N\text{-}1)/2$

$T(N) = 2N + 3(N\text{-}1) + 2(N(N+1)/2\ \text{-}1) + 2(N(N\text{-}1)/2) = 2N^2 + 6N\ \text{-}6$

$T(N) = O(N^2)$: the number of comparisons in the worst case grows quadratically.

High-level Complexity Analysis

Two nested loops:

- **Outer loop:** N-1 executions
- **Inner loop in the worst case:** *i* executions at the *i-th* iteration of the outer loop

Complexity:

$T(N) = 1+2+3+ \dots +(N-2)+(N-1)$

$= \Sigma_{1 \leq i < N}\, i = N(N-1)/2$

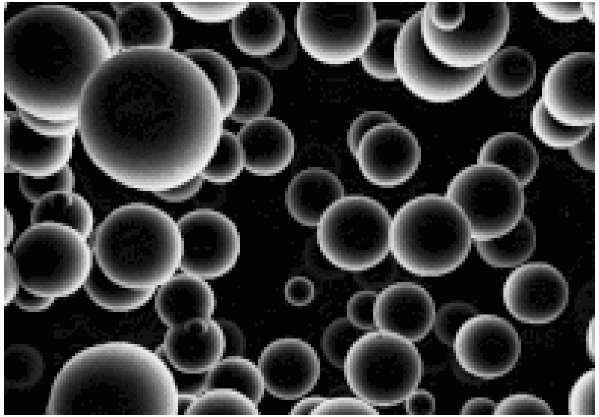
T(N) grows quadratically with N.

Finite arithmetic progression with ratio 1 (Gauss, end of XVII cent.)

$T(N) = O(N^2)$: the number of comparisons and of swaps grows quadratically.

Bubble/Exchange sort

Also known as ***sinking sort***. It is a sorting algorithm that is used to sort the elements in **an ascending order**. It uses less storage space.



Bubble/Exchange sort

Let **N** integers be stored in an array **A** whose indices range from **l**=0 and **r**=**N**-1. Conceptually array **A** consists of 2 **subarrays**:

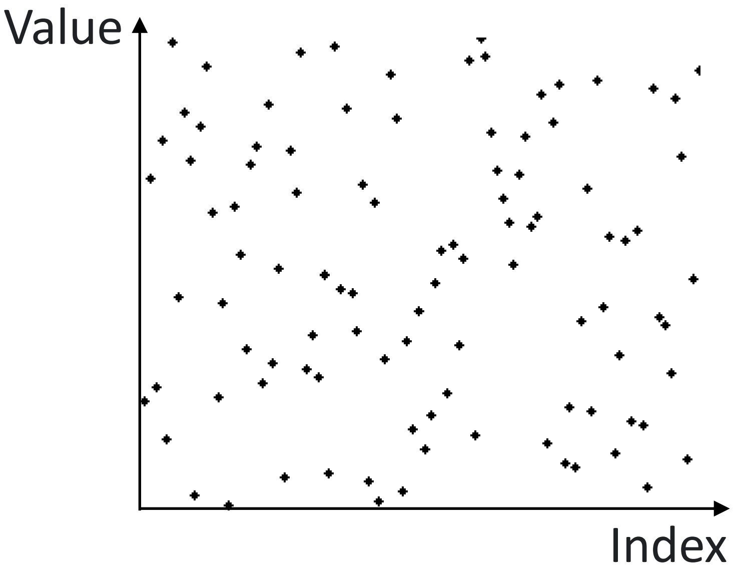
- the **right one**: **already sorted**
- the **left one: not yet sorted**

Initially the **right subarray** is empty, the left one contains **N** items.

The basic operation is a comparison between adjacent items of array **A** (**A[j]** and **A[j+1]**), swapping them if **A[j]** > **A[j+1]**.

Bubble/Exchange sort

6 5 3 1 8 7 2 4



Approach

Incremental paradigm:

- At each **step** expand the already sorted **right subarray** by **inserting** an item taken from the **not yet sorted** left subarray
- Insertion must guarantee that the **right subarray** remains sorted after insertion (*invariance of the sorting property*)
- At iteration **i** insert the **largest item** of the **left subarray** (A_l … A_{r-i+l}) into the **leftmost position** of **the right subarray** A[r-i+l]. The sorted **right subarray** grows by **1** in size to the left, dually the unsorted left one decreases in size by **1**
- **Termination:** all items have been properly inserted, the **right subarray** contains **N** items, the **left one** is empty.

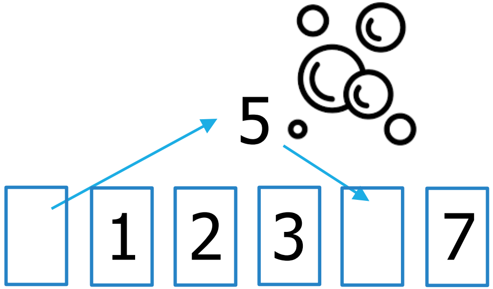
6 5 3 1 8 7 2 4

Step *i*: identifying the maximum

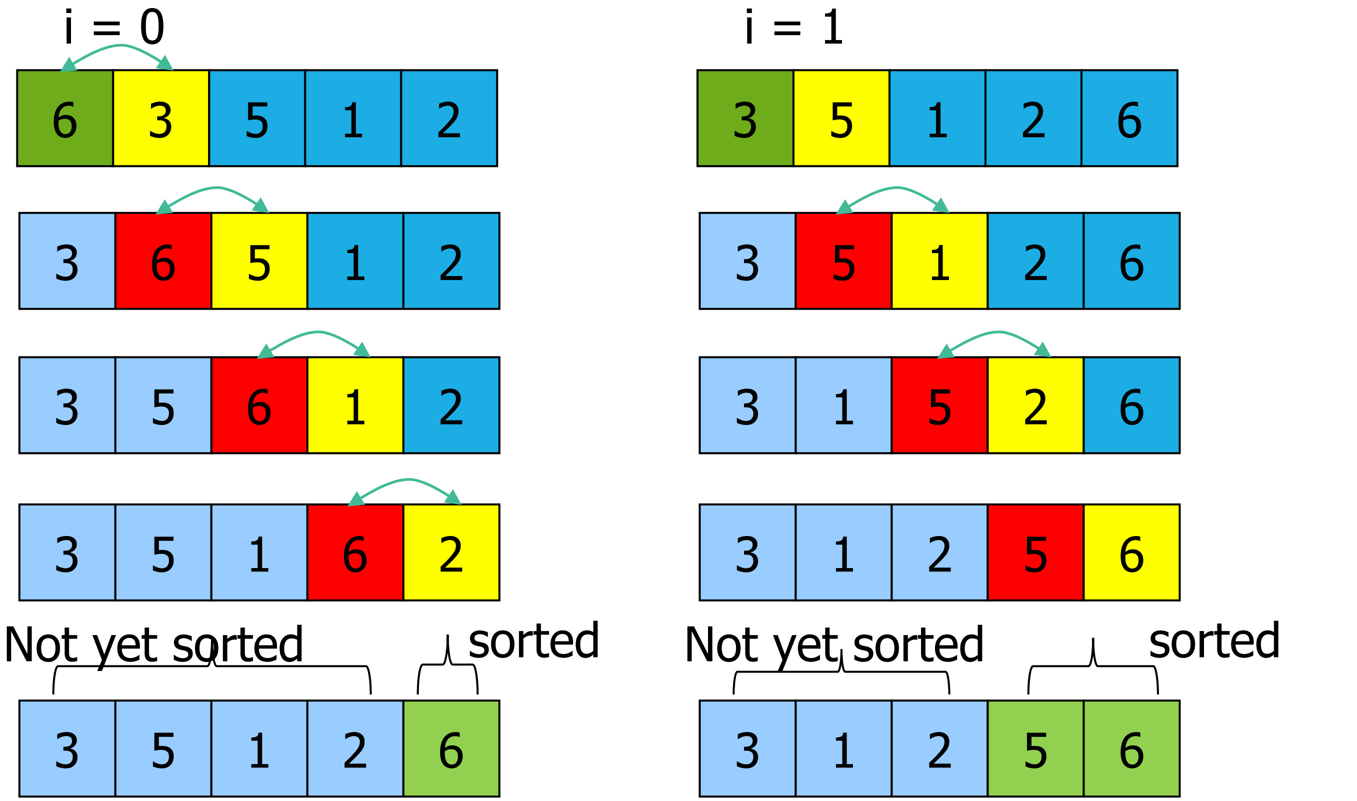
Identification of the **largest item** in the left subarray at step **i**

- scan the left subarray comparing item pairs **A[j]** and **A[j+1]** and swapping them if **A[j]** > **A[j+1].**

When the loop ends, the largest item has «**floated**» like a «**bubble**» to the correct position at the **leftmost** location of the sorted right subarray.



Example


i = 0
6 3 5 1 2
3 6 5 1 2
3 5 6 1 2
3 5 1 6 2
Not yet sorted
sorted
3 5 1 2 6
i = 1
3 5 1 2 6
3 5 1 2 6
3 1 5 2 6
3 1 2 5 6
Not yet sorted
sorted
3 1 2 5 6

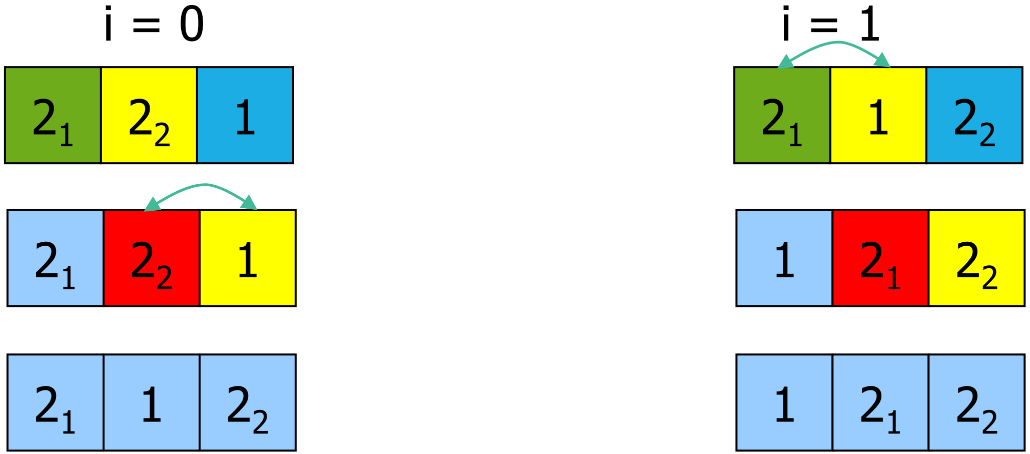
Example of functions implementing the bubble sorting algorithm

```
void BubbleSort(int A[], int N){
  int i, j, l=0, r=N-1;
  int temp;
  for (i = l; i < r; i++) {
    for (j = l; j < r - i +l; j++)
      if (A[j] > A[j+1])
      {
        temp = A[j];
        A[j] = A[j+1];
        A[j+1] = temp;
      }
  }
  return;
}
```

i-l is the size of the already sorted right subarray

Bubble/Exchange sort Features

- **in place:** used only array **A** and variable **temp**
- **stable:** if there are several duplicate keys the rightmost none takes the rightmost position and no other identical key jumps over it to the right:



High-level Complexity Analysis

Two nested loops:

- **Outer loop:** always executed **N-1** times
- **Inner loop**: at the **i-th** iteration executed always **N-1-i** times

T(N) = (N-1) + (N-2) +… 2 + 1

= $\Sigma_{1 \leq i < N}$ i = N(N-1)/2

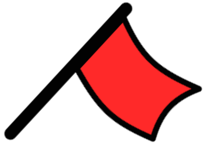
T(N) = $\Theta(N^2)$.

Finite arithmetic progression with ratio 1 (Gauss, end of XVII cent.)

- **Number of swaps** in the worst case: $O(N^2)$: a swap doesn't necessarily occur
- **Number of comparisons** in the worst case: $\Theta(N^2)$: a comparison always takes place

Optimization

- Use **a flag** to mark whether there has been **swaps**. Execution **continues** only if there have been **swaps**



  - The **outer loop** is executed at most **N-1** times
  - The **inner loop** at the **i-th** iteration is executed always **N-1-i** times

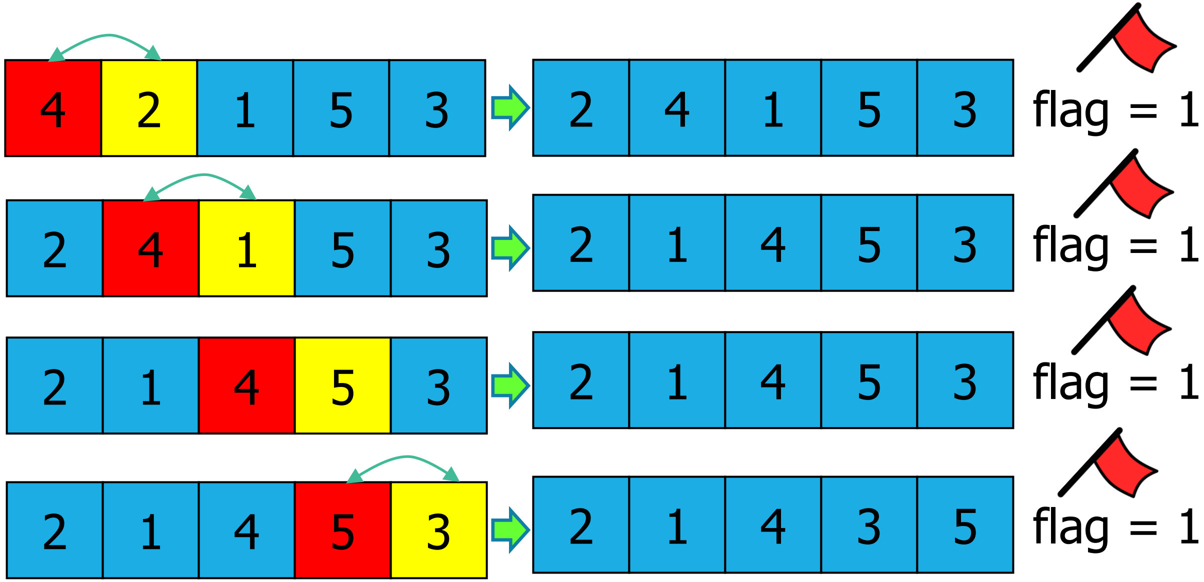
- Number of comparisons in the worst case: $O(N^2)$: a comparison doesn't necessarily take place
- Average case complexity is improved, no change in worst case complexity.

Example

flag = 1

i = 0 outer loop executed

flag = 0



j = 0 inner loop executed

4	2	1	5	3	→	2	4	1	5	3	flag = 1

j = 1 inner loop executed

2	4	1	5	3	→	2	1	4	5	3	flag = 1

j = 2 inner loop executed

2	1	4	5	3	→	2	1	4	5	3	flag = 1

j = 3 inner loop executed

2	1	4	5	3	→	2	1	4	3	5	flag = 1

Example

i = 1 outer loop executed

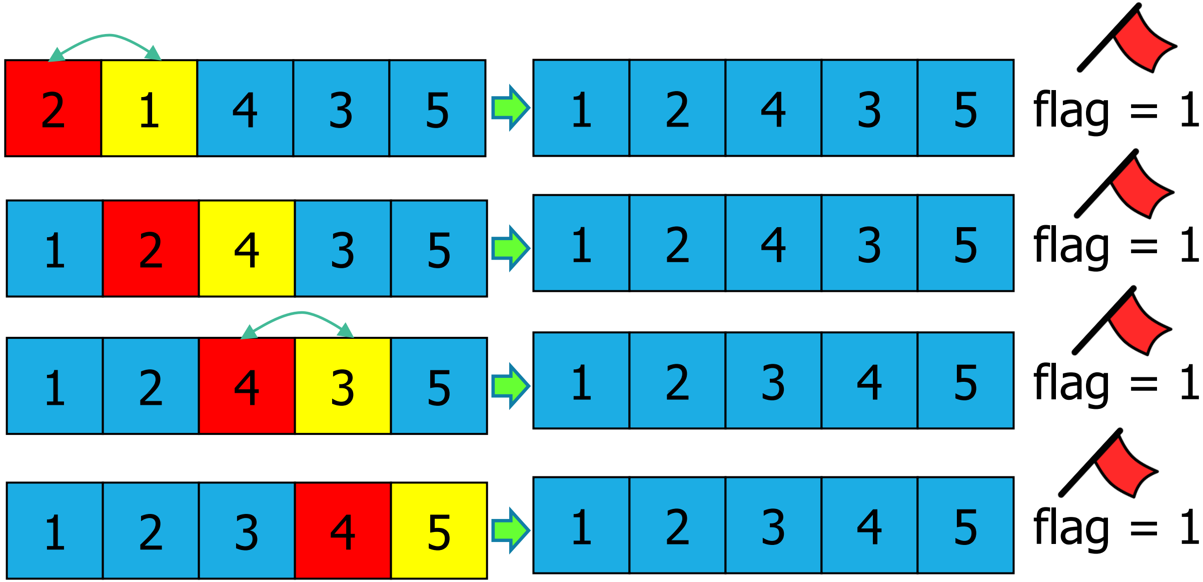
flag = 0

j = 0 inner loop executed

j = 1 inner loop executed

j = 2 inner loop executed

j = 3 inner loop executed



2 1 4 3 5 → 1 2 4 3 5 flag = 1

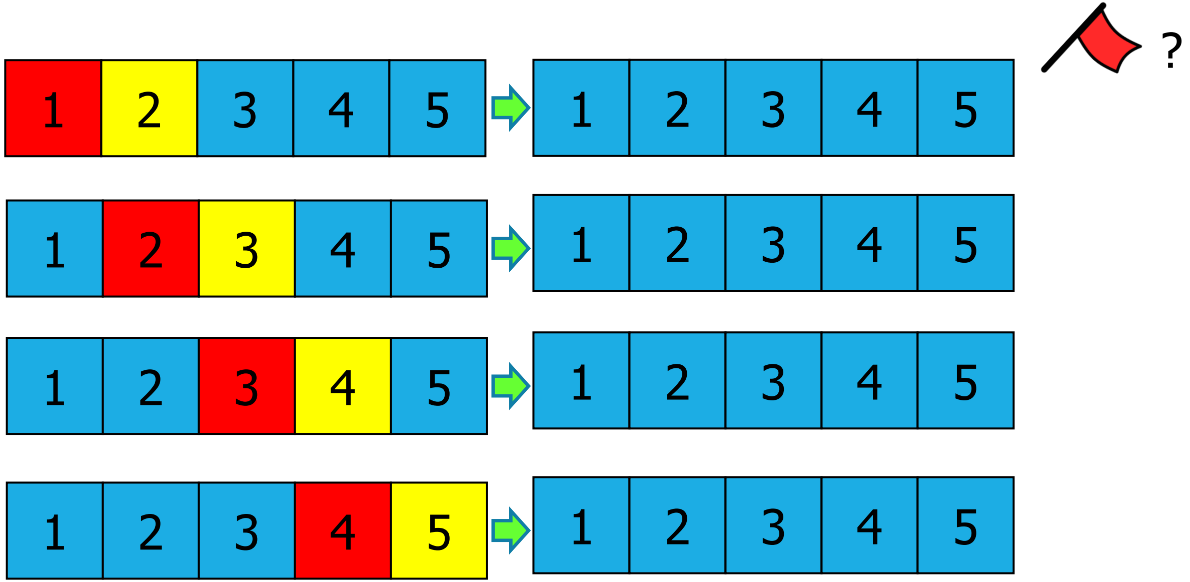
1 2 4 3 5 → 1 2 4 3 5 flag = 1

1 2 4 3 5 → 1 2 3 4 5 flag = 1

1 2 3 4 5 → 1 2 3 4 5 flag = 1

i = 2 outer loop executed

flag = 0



j = 0 inner loop executed

1	2	3	4	5	⇨	1	2	3	4	5

j = 1 inner loop executed

1	2	3	4	5	⇨	1	2	3	4	5

j = 2 inner loop executed

1	2	3	4	5	⇨	1	2	3	4	5

j = 3 inner loop executed

1	2	3	4	5	⇨	1	2	3	4	5

i = 3 but **flag = 0** as there has been **no swaps**, outer loop not executed

Example of functions implementing the optimized bubble sorting algorithm

```
void OptBubbleSort(int A[], int N) {
   int i, j, l=0, r=N-1, flag=1;
   int temp;

   for (i = 1; i < r && flag==1; i++)
   {
        flag = 0;
        for (j = 1; j < r – i + 1; j++) {
             if (A[j] > A[j+1])
             {
                  flag = 1;
                  temp = A[j];
                  A[j] = A[j+1];
                  A[j+1] = temp;
             }
        }
   }
        return;
}
```

Question?

Whats the best way to sort one million 32-bit integers?

Question?

Former Google CEO Eric Schmidt asked Barack Obama during an interview:

Whats the best way to sort one million 32-bit integers?

Question?

Former Google CEO Eric Schmidt asked Barack Obama during an interview:

Whats the best way to sort one million 32-bit integers?

Obama paused for a moment and replied:

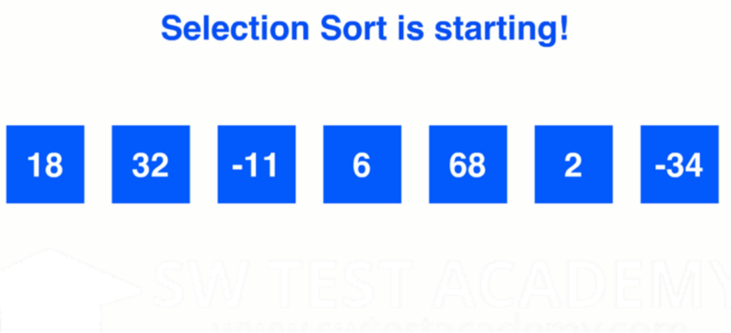
"I think the bubble sort would be the wrong way to go"

The Internet Sort

It’s just a ***bubble sort***, but perform every comparison by searching the internet. For example, “*Which is greater – 0.211 or 0.75?*”.

Selection sort

It is a **simple** and **efficient** sorting algorithm that works by repeatedly selecting the **smallest** (***or largest***) **element** from the **unsorted** portion of the list and moving it to the **sorted portion** of the list.



Selection Sort is starting!

Selection sort

Let **N** integers be stored in an array **A** whose indices range from **l**=0 and **r**=**N**-1.

Conceptually array **A** consists of 2 subarrays:

- **Left subarray:** already sorted
- **Right subarray:** **not yet sorted**

Initially the **left subarray is empty**, the right one contains **N** items.

Approach

Incremental paradigm:

- At each step **expand** the **already sorted** left subarray inserting an item taken from the **not yet sorted** right subarray

- Insertion must guarantee that the **left subarray remains sorted** after insertion (*invariance of the sorting property*). This is guaranteed by identifying the **smallest item** in the right subarray (**A_i … A_r**) and assigning it to **A[i].** The sorted left subarray **grows** by **1** in size to the right, dually the not yet sorted right one **decreases** in size by **1**

- **Termination:** all items have been properly inserted, the left subarray contains **N** items, the **right one is empty**.

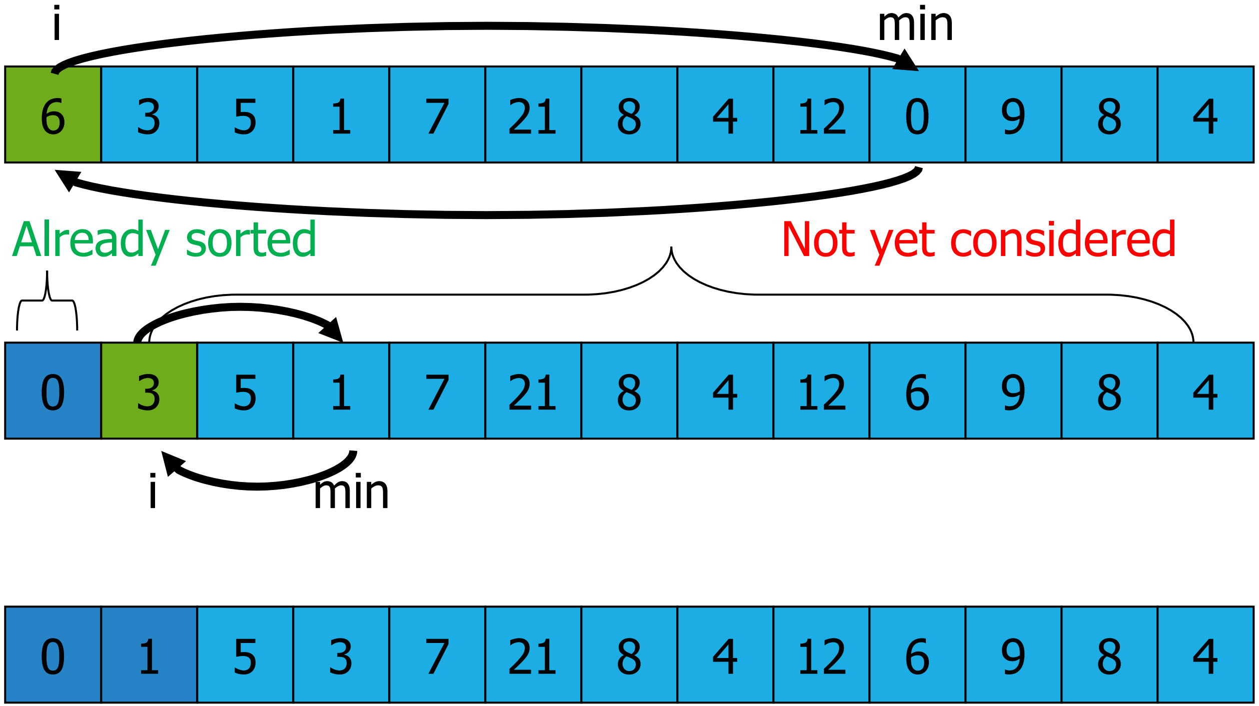
Step *i*: identifying the minimum

Identification of **the smallest item** of the right subarray at the **i-th** step:

- Scan the **right subarray**, assuming that the smallest item is in **A[i]** and updating the smallest item at each comparison with the following items

When the **loop** ends, the smallest item has been identified by means of its position (***index in A***) and **swapped** with **A[i].**

Example



i

min

6	3	5	1	7	21	8	4	12	0	9	8	4

Already sorted

Not yet considered

0	3	5	1	7	21	8	4	12	6	9	8	4

i

min

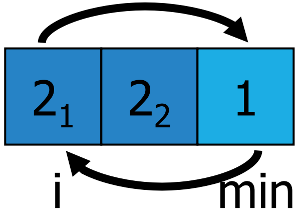
0	1	5	3	7	21	8	4	12	6	9	8	4

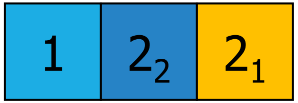
Example of function implementing the selection sorting algorithm

```
void SelectionSort(int A[], int N) {
  int i, j, l=0, r=N-1, min;
  int temp;
  for (i = l; i < r; i++) {
    min = i;
    for (j = i+1; j <= r; j++)
      if (A[j] < A[min])
         min = j;
    if (min != i) {
      temp = A[i];
      A[i] = A[min];
      A[min] = temp;
    }
  }
  return;
}
```

Selection sort Features

- **in place:** only array **A** and variable **temp** are used
- **non stable:** a swap of **2** «far away» items may allow a **duplicate key** to jump to the left of a previous identical key:





High-level Complexity Analysis

Two nested loops:

- **Outer loop**: always executed **N**-1 times
- **Inner loop:** at the **i-th** iteration always executed **N**-1-i times

$T(N) = (N-1) + (N-2) + \dots 2 + 1$

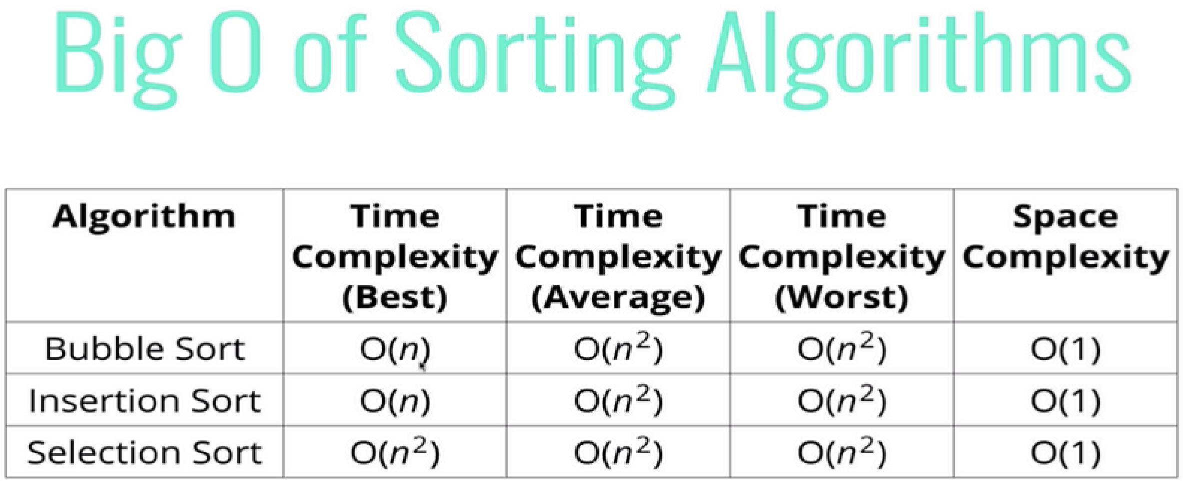
$= \Sigma_{1 \leq i < N}\, i = N(N-1)/2$

$T(n) = \Theta(N^2)$.

Finite arithmetic progression with ratio 1 (Gauss, end of XVII cent.)

- **Number of swaps in the worst case:** O(N): it may happen to always swap the current item with the current smallest one, but this occurs at most **N** times
- **Number of comparisons in the worst case:** $\Theta(N^2)$: comparisons always occur
- **The algorithm is quadratic**, as complexity depends on the **number of comparisons**, not on the **number of swaps**.

Quick summary



Big O of Sorting Algorithms

Algorithm	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$

Shell sort (Shell, 1959)

Donald Shell

D. L. Shell. 1959. "A high-speed sorting procedure". Commun. ACM 2, 7 (July 1959), 30–32.

Shell sort (Shell, 1959)

It is a **generalized** version of the insertion sort algorithm. It first sorts elements that are **far apart** from each other and successively **reduces** the interval between the elements to be sorted.

Limit of Insertion sort: only adjacent items are compared and possibly swapped.

Shell sort approach:

- **Compare** and possibly **swap** items at distance **h**
- Define a decreasing sequence of integers **h** that end at 1. The last step is Insertion sort.

73
67
56
32
52
41
83
37
32
10

D. L. Shell. 1959. "A high-speed sorting procedure". Commun. ACM 2, 7 (July 1959), 30-32.

Linear Sequences

- Finite set of consecutive items. Each item is uniquely associated to an index

$$\mathbf{a_0, a_1, \ldots , a_i, \ldots, a_{n-1}}$$

- A predecessor/successor relationship is defined on pairs of consecutive items:

$$\mathbf{a_{i+1}} = \mathrm{succ}(\mathbf{a_i}) \qquad \mathbf{a_i} = \mathrm{pred}(\mathbf{a_{i+1}})$$

- **Storage and access:**
  - Array: **contiguous** data in memory with direct access:
    - Given index **i**, it is possible to access item $\mathbf{a_i}$ without scanning the linear sequence
    - Cost of access doesn't depend on the position of the item in the linear sequence, thus it is O(1)
  - list: **non contiguous** data in memory with sequential access. Topic of the second year course.

Subarrays and Subsequences

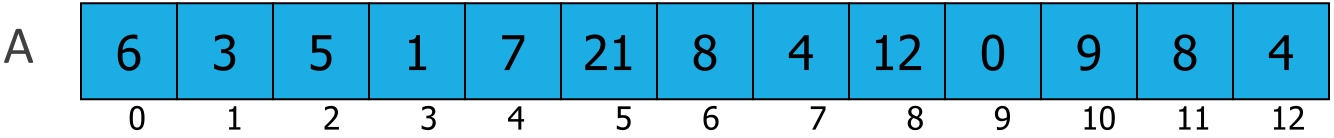
Given a linear sequence of **N** integers stored in array **A**

$$\mathbf{A} = (a_0, a_1, \ldots a_{N-1})$$

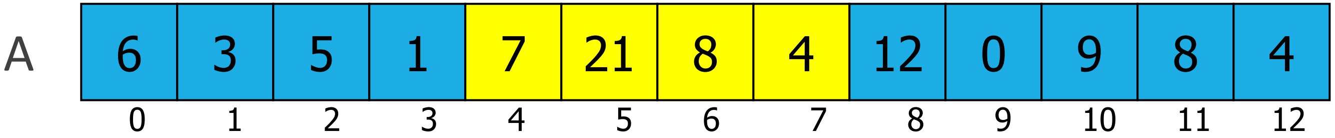
- a **subsequence** of **A** of length **k** (**k** ≤**N**) is any tuple **Y** of **k** items of **A** with increasing and **not necessarily** contiguous indices $i_0, i_1, \cdots, i_{k-1}$

- a **subarray** of **A** of length **k** (**k** ≤**N**) is any tuple Y of **k** items of A with increasing and contiguous indices $i_0, i_1, \cdots, i_{k-1}$.

Example

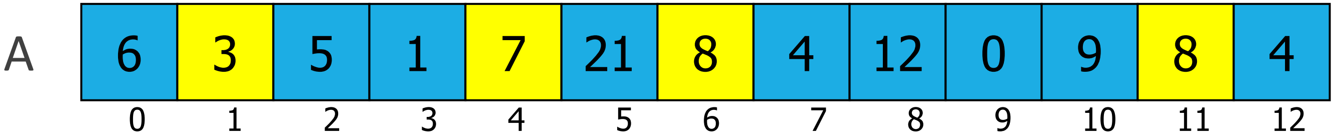


A	6	3	5	1	7	21	8	4	12	0	9	8	4
	0	1	2	3	4	5	6	7	8	9	10	11	12



A	6	3	5	1	7	21	8	4	12	0	9	8	4
	0	1	2	3	4	5	6	7	8	9	10	11	12

Subarray of **A** with **k**=4 items with increasing and contiguous indices $\mathbf{i_4}$, $\mathbf{i_5}$, $\mathbf{i_6}$, $\mathbf{i_7}$



A	6	3	5	1	7	21	8	4	12	0	9	8	4
	0	1	2	3	4	5	6	7	8	9	10	11	12

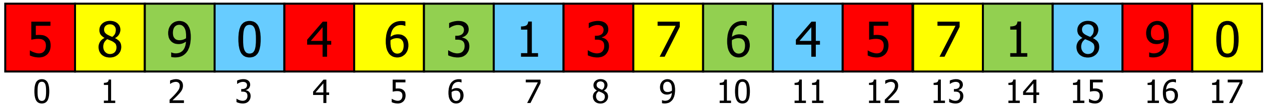
Subsequence of **A** with **k**=4 items

with increasing and non contiguous indices $\mathbf{i_1}$, $\mathbf{i_4}$, $\mathbf{i_6}$, $\mathbf{i_{11}}$

Subsequences in Shell sort

Shell sort works with **subsequences** composed by array items whose indices at distance **h** from each other.

Example: **h**=4

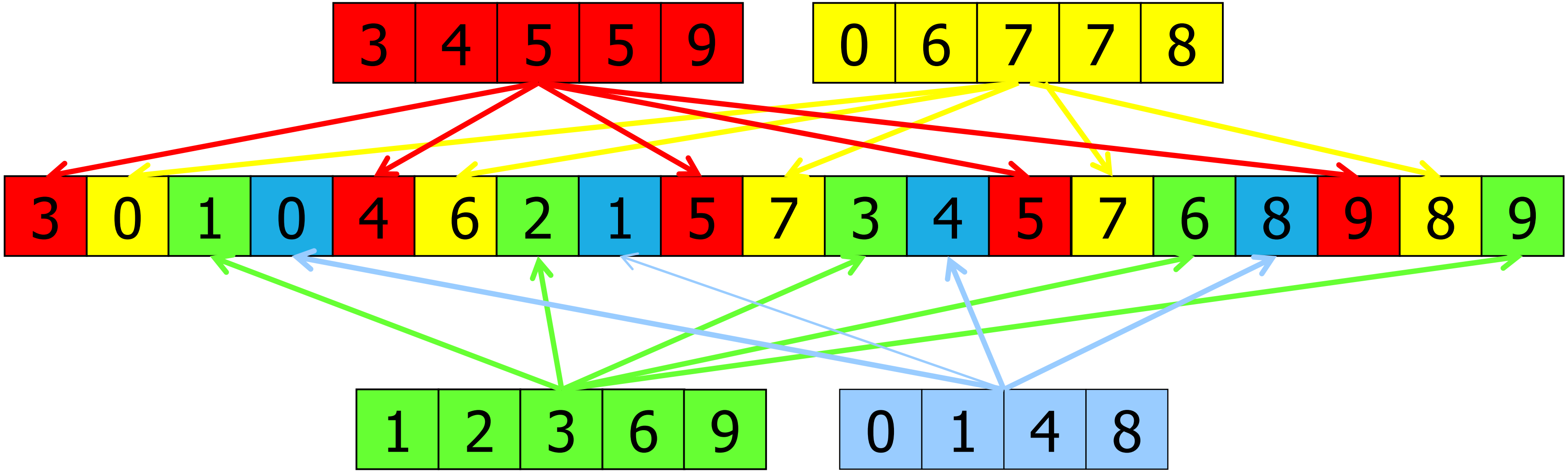


5	8	9	0	4	6	3	1	3	7	6	4	5	7	1	8	9	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Subsequences in Shell sort

If the **subsequences** containing items at distance **h** are sorted, the array is **h-sorted**.

Example: **h**=4



Subsequences in Shell sort

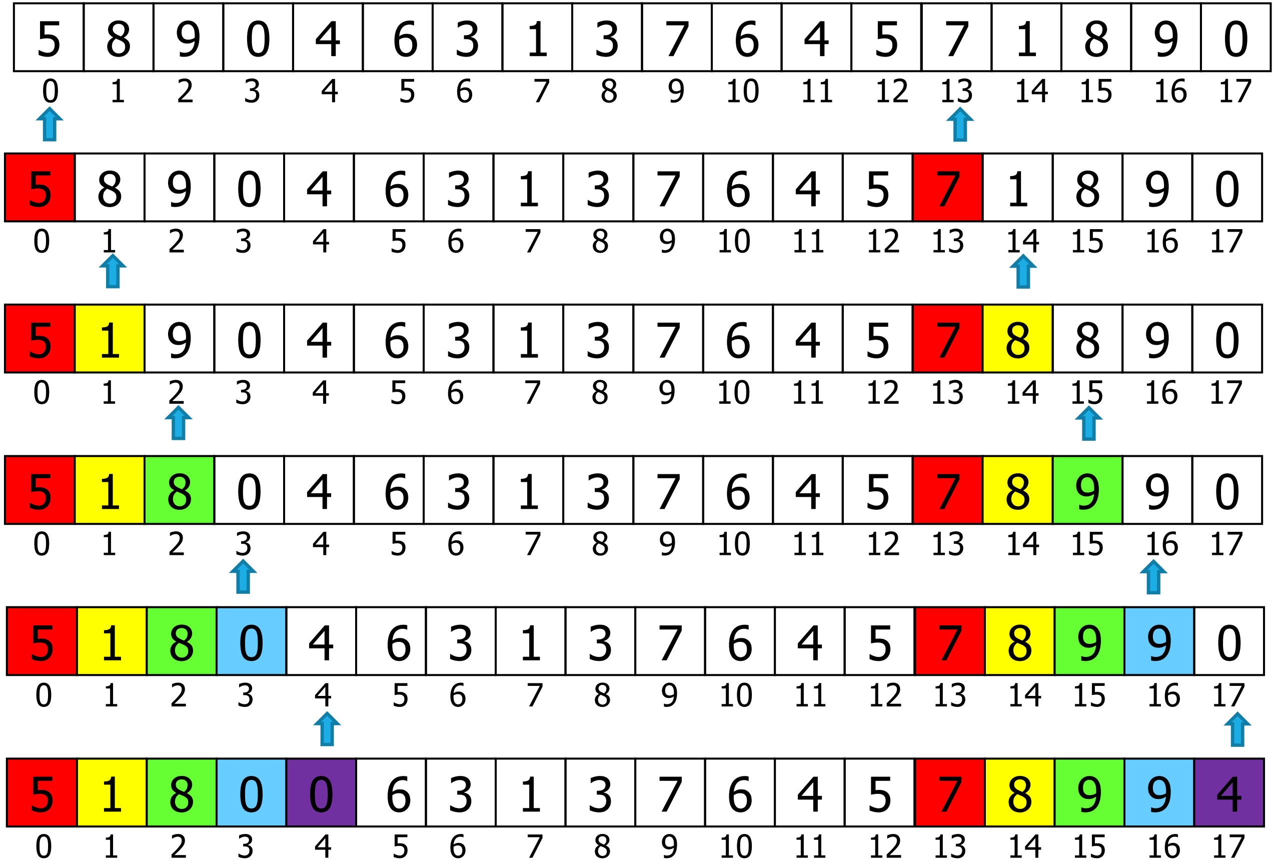
For each **subsequence** apply Insertion sort. The items of the **subsequence** are the items at distance **h** from the current one.

```
    for i = l+h; i <= r; i++) {
      j = i; x = A[i];
      while (j>= l+h && x < A[j-h]) {
        A[j] = A[j-h];
        j -=h;
      }
      A[j] = x;
    }
```

Example

sequence h: 13, 4, 1

Step 1: h=13



5	8	9	0	4	6	3	1	3	7	6	4	5	7	1	8	9	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

5	8	9	0	4	6	3	1	3	7	6	4	5	7	1	8	9	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

5	1	9	0	4	6	3	1	3	7	6	4	5	7	8	8	9	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

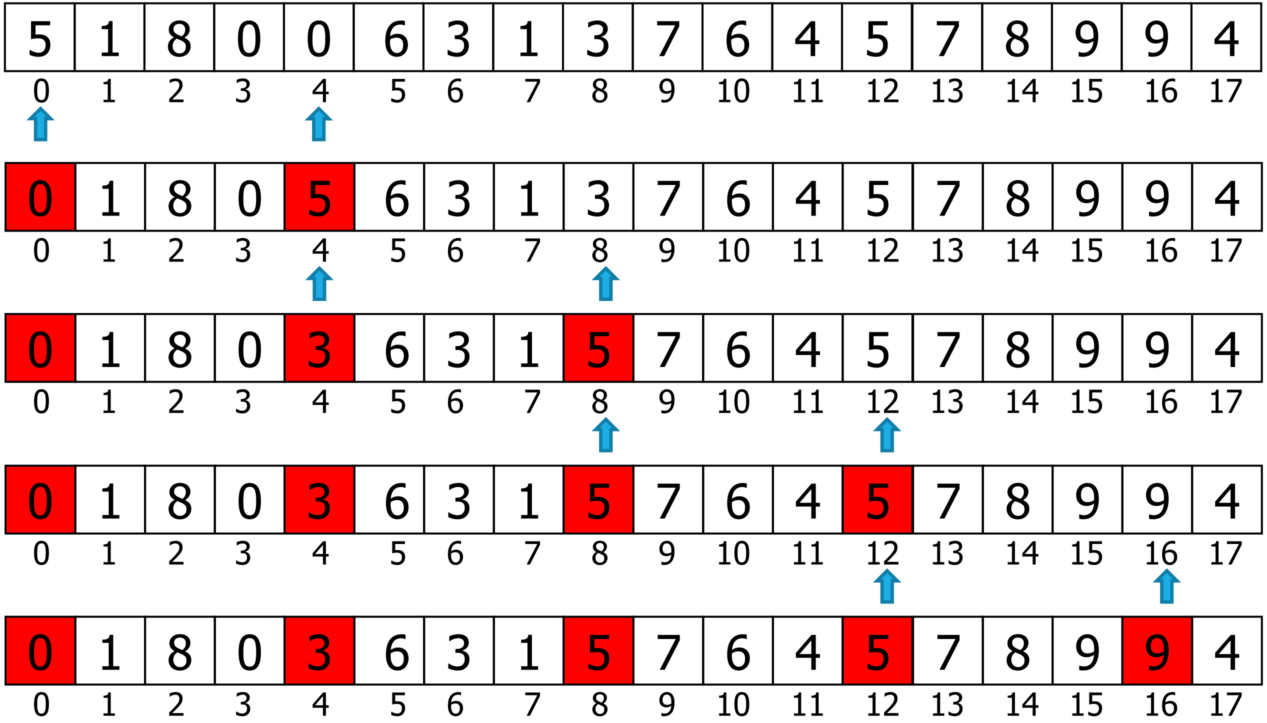
5	1	8	0	4	6	3	1	3	7	6	4	5	7	8	9	9	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

5	1	8	0	4	6	3	1	3	7	6	4	5	7	8	9	9	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

5	1	8	0	0	6	3	1	3	7	6	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

sequence h: 13, 4, 1

Step 2: h=4



5	1	8	0	0	6	3	1	3	7	6	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
⇧				⇧													

0	1	8	0	**5**	6	3	1	3	7	6	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
				⇧				⇧									

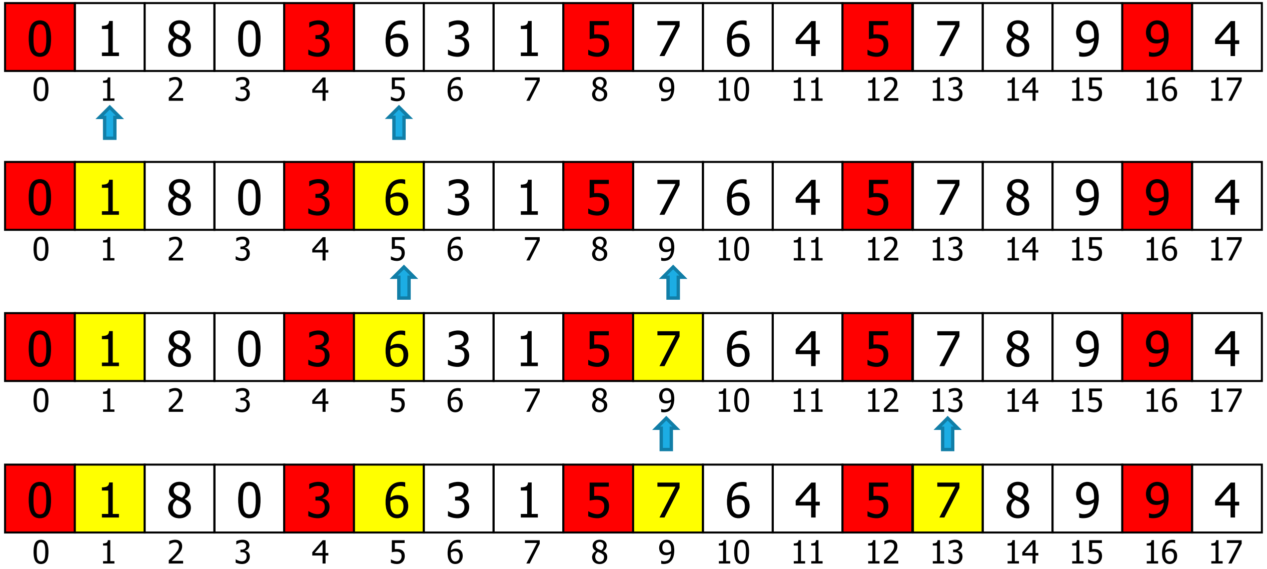
0	1	8	0	**3**	6	3	1	**5**	7	6	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
								⇧				⇧					

0	1	8	0	**3**	6	3	1	**5**	7	6	4	**5**	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
												⇧				⇧	

0	1	8	0	**3**	6	3	1	**5**	7	6	4	**5**	7	8	9	**9**	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

sequence h: 13, 4, 1

Step 2: h=4



0	1	8	0	3	6	3	1	5	7	6	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

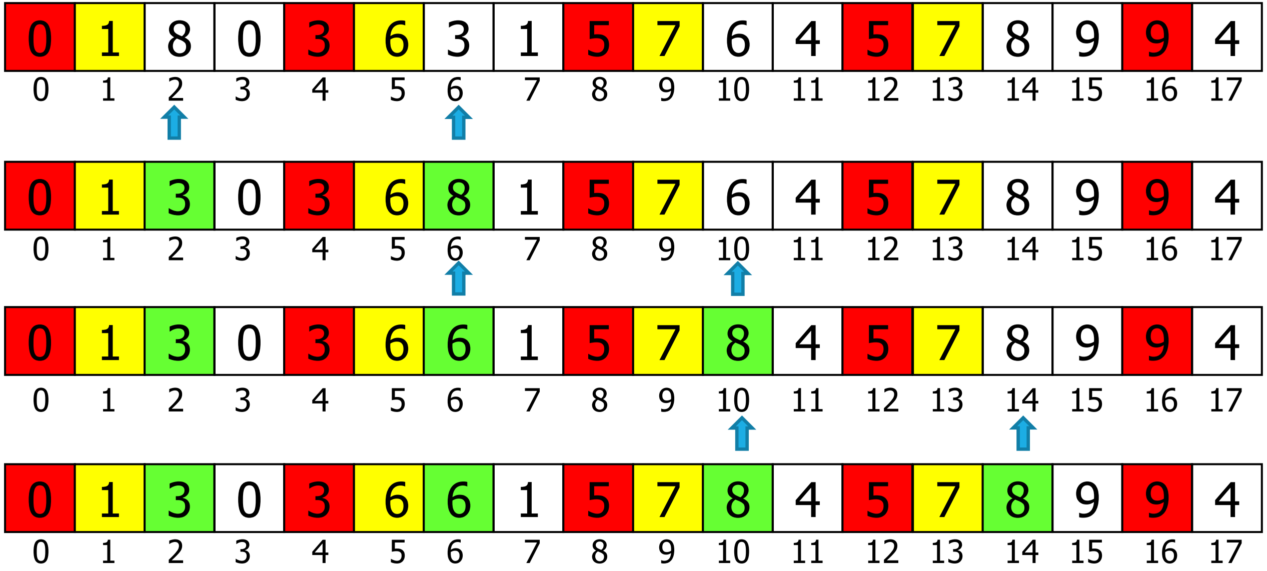
0	1	8	0	3	6	3	1	5	7	6	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

0	1	8	0	3	6	3	1	5	7	6	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

0	1	8	0	3	6	3	1	5	7	6	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

sequence h: 13, 4, 1

Step 2: h=4



0	1	8	0	3	6	3	1	5	7	6	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
		⇧				⇧											

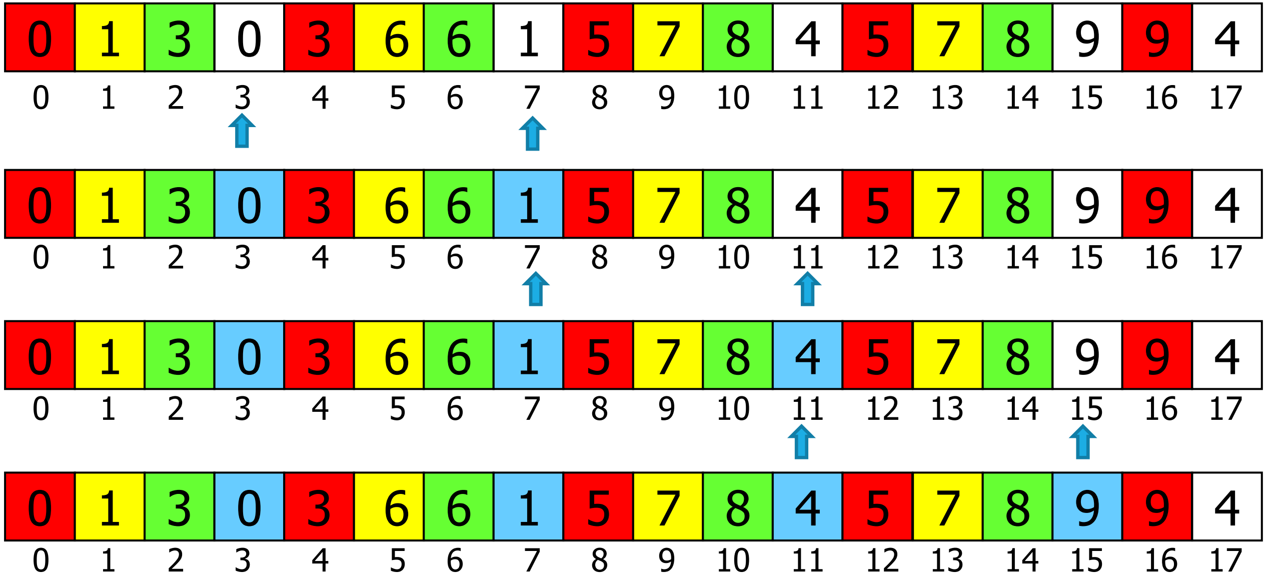
0	1	3	0	3	6	8	1	5	7	6	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
						⇧				⇧							

0	1	3	0	3	6	6	1	5	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
										⇧				⇧			

0	1	3	0	3	6	6	1	5	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

sequence h: 13, 4, 1

Step 2: h=4



0	1	3	0	3	6	6	1	5	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

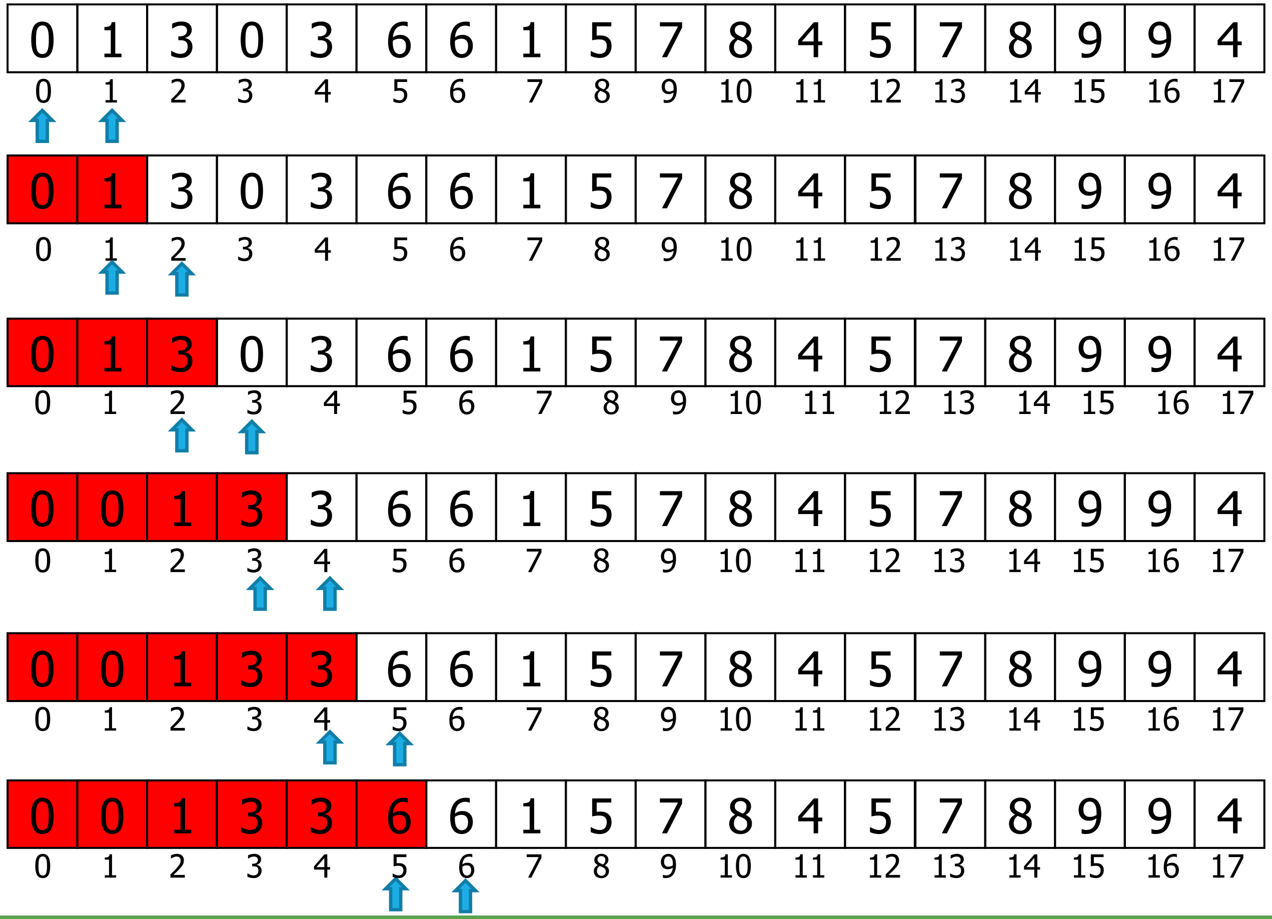
0	1	3	0	3	6	6	1	5	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

0	1	3	0	3	6	6	1	5	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

0	1	3	0	3	6	6	1	5	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

sequence h: 13, 4, 1

Step 3: h=1



0	1	3	0	3	6	6	1	5	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

0	1	3	0	3	6	6	1	5	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

0	1	3	0	3	6	6	1	5	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

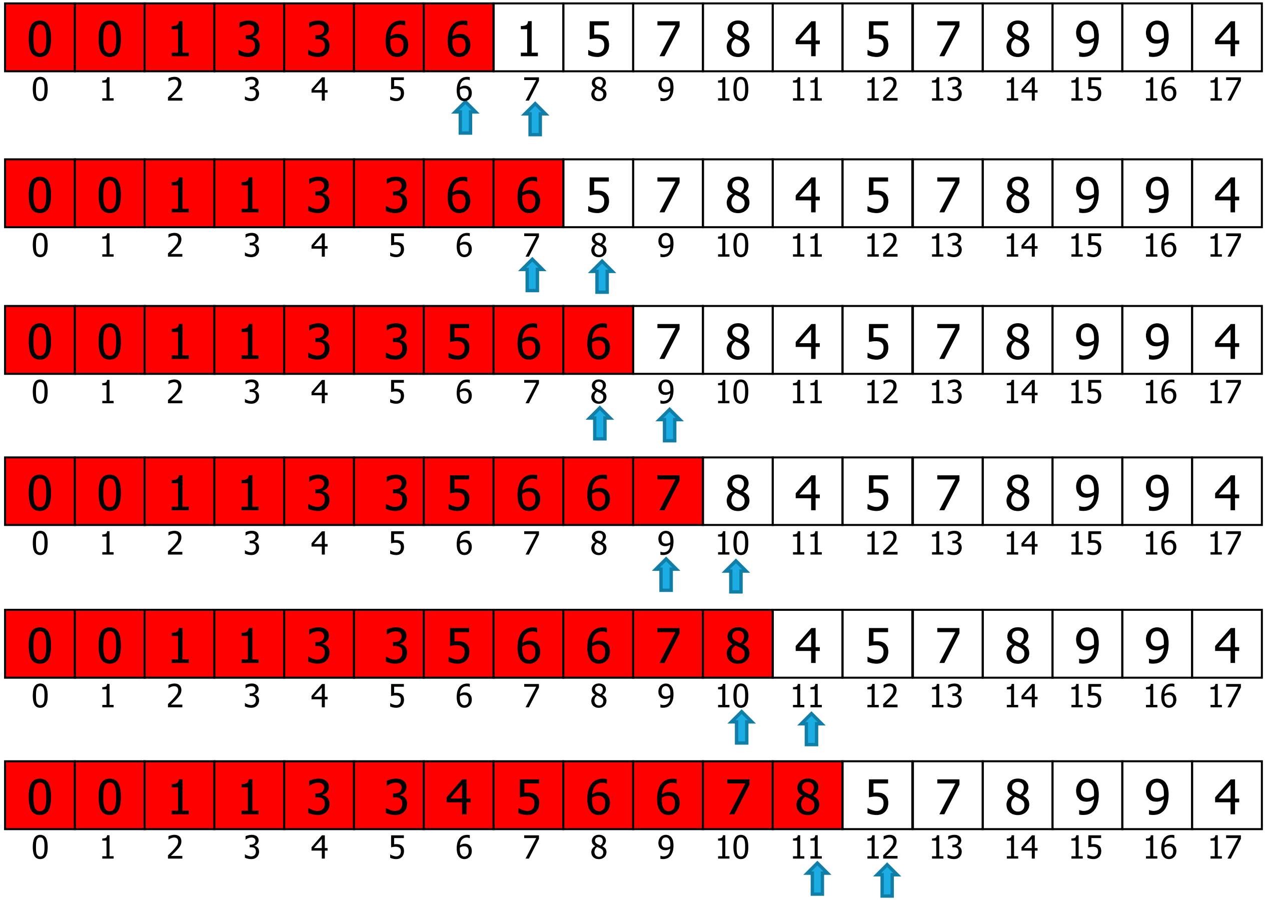
0	0	1	3	3	6	6	1	5	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

0	0	1	3	3	6	6	1	5	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

0	0	1	3	3	6	6	1	5	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

sequence h: 13, 4, 1

Step 3: h=1



0	0	1	3	3	6	6	1	5	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
						⬆	⬆										

0	0	1	1	3	3	6	6	5	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
							⬆	⬆									

0	0	1	1	3	3	5	6	6	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
								⬆	⬆								

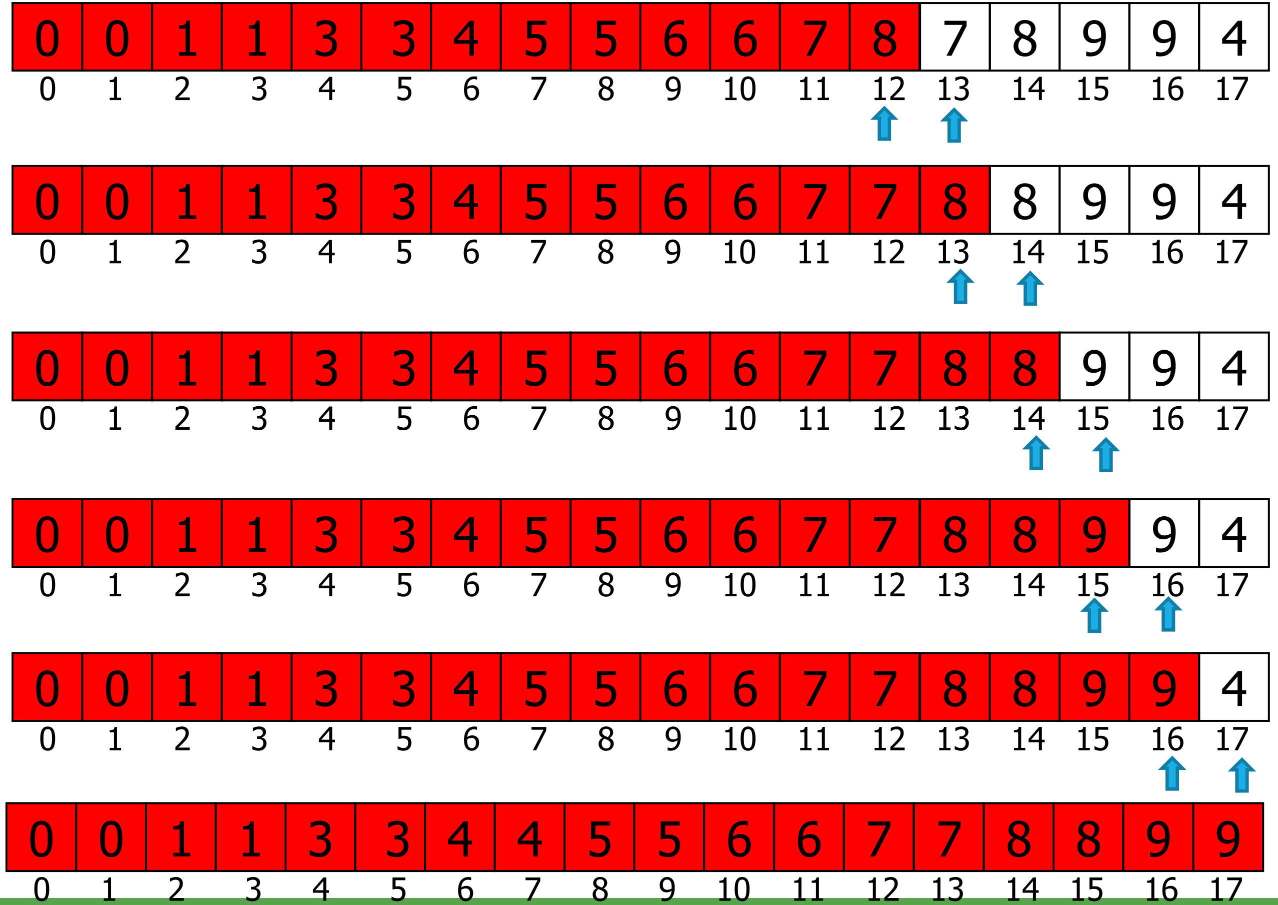
0	0	1	1	3	3	5	6	6	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
									⬆	⬆							

0	0	1	1	3	3	5	6	6	7	8	4	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
										⬆	⬆						

0	0	1	1	3	3	4	5	6	6	7	8	5	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
											⬆	⬆					

sequence h: 13, 4, 1

Step 3: h=1



0	0	1	1	3	3	4	5	5	6	6	7	8	7	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

0	0	1	1	3	3	4	5	5	6	6	7	7	8	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

0	0	1	1	3	3	4	5	5	6	6	7	7	8	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

0	0	1	1	3	3	4	5	5	6	6	7	7	8	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

0	0	1	1	3	3	4	5	5	6	6	7	7	8	8	9	9	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

0	0	1	1	3	3	4	4	5	5	6	6	7	7	8	8	9	9
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Sequences

- **Shell's sequence** (1959) $h_i = 2^{i-1}$

$h_1 = 1, h_2 = 2, h_3 = 4, h_4 = 8, h_5 = 16, \dots$

- **Hibbard's sequence** (1963): $h_i = 2^i - 1$

$h_1 = 1, h_2 = 3, h_3 = 7, h_4 = 15, h_5 = 31, \dots$

- **Pratt's sequence** (1971):

$1, 2, 3, 4, 6, 8, 9, 12, 16, \dots, 2^p3^q, \dots$

- **Knuth's** (1973): initially $h_0=0$ then $h_i = 3h_{i-1}+1$

$h_1 = 1, h_2 = 4, h_3 = 13, h_4 = 40, h_5 = 121, \dots$

- **Sedgewick's sequence** (1986):

1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, ...

Hibbard, T. N. (1963). An empirical study of minimal storage sorting. Communications of the ACM, 6(5), 206-213.

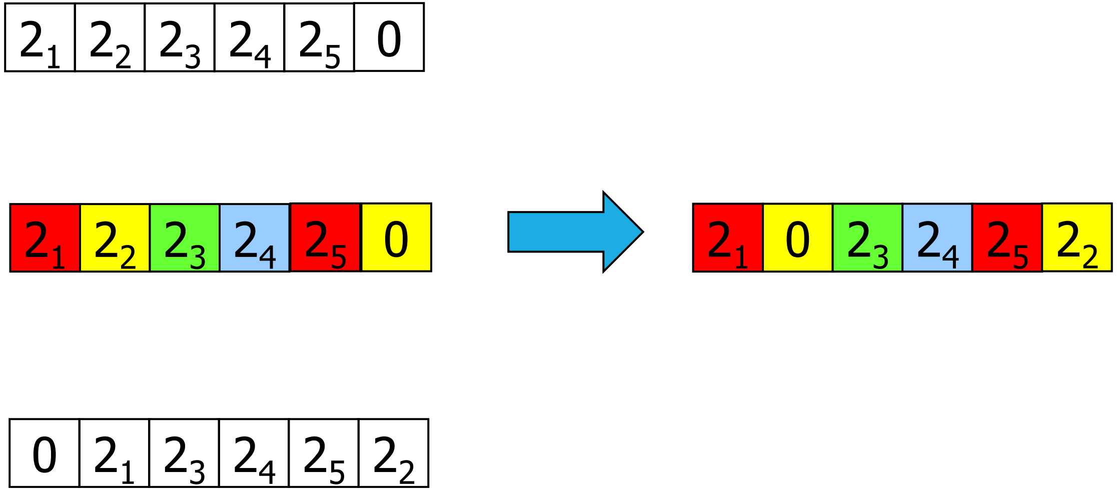
Pratt, V. R. (1972). Shellsort and sorting networks (Vol. 72, No. 260). C. S. Department, Stanford University.

```
void ShellSort(int A[], int N) {
  int i, j, x, l=0, r=N-1, h=1;
  while (h <= N/3)
    h = 3*h+1;
  while(h >= 1) {
    for (i = l + h; i <= r; i++) {
      j = i;
      x = A[i];
      while(j >= l + h  && x < A[j-h]) {
        A[j] = A[j-h];
        j -=h;
      }
      A[j] = x;
    }
    h = h/3;
  }
}
```

Knuth’s sequence

Shell sort Features

- **in place:** apart from array **A** only variable **x** is used
- **non stable:** a **swap between** «distant» items **can force a duplicate key** to jump to the lest of a previous occurrence:



2_1	2_2	2_3	2_4	2_5	0

2_1	2_2	2_3	2_4	2_5	0

2_1	0	2_3	2_4	2_5	2_2

0	2_1	2_3	2_4	2_5	2_2

High-level Complexity Analysis

- With **Shell’s sequence:** 1 2 4 8 16 … T(N) = $O(N^2)$
- With **Hibbard’s sequence:** 1 3 7 15 31 … T(N) = $O(N^{3/2})$
- With **Pratt’s sequence:** 1 2 3 4 6 8 9 12 … T(N) = $O(N\log^2 N)$
- With **Knuth’s sequence:** 1 4 13 40 121 … T(n) = $O(N^{3/2})$
- With **Sedgewick’s sequence:** 1, 5, 19, 41, 109, 209, … T(N) = $O(N^{4/3})$