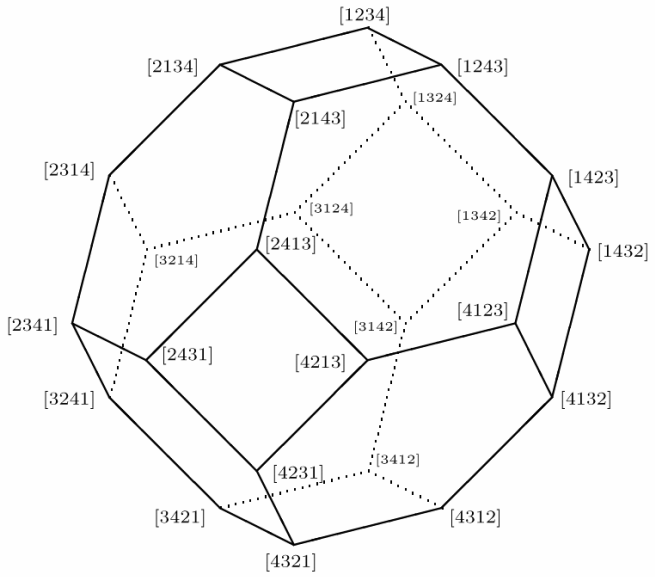
CS3334 Data Structures

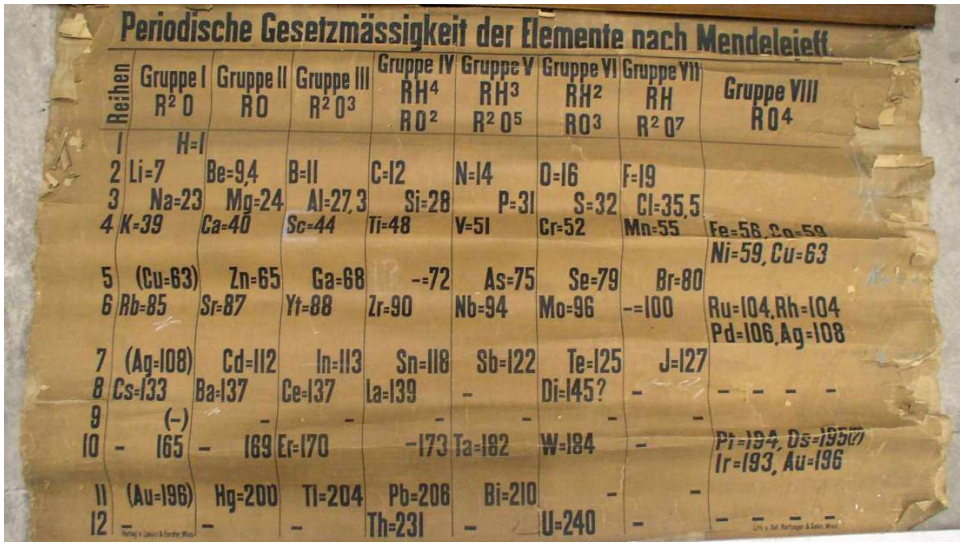
Lecture 4: Bubble Sort & Insertion Sort



Chee Wei Tan

Sorting Since Time Immemorial

Plimpton 322 Tablet: Sorted Pythagorean Triples
https://www.maa.org/sites/default/files/pdf/news/monthly105-120.pdf

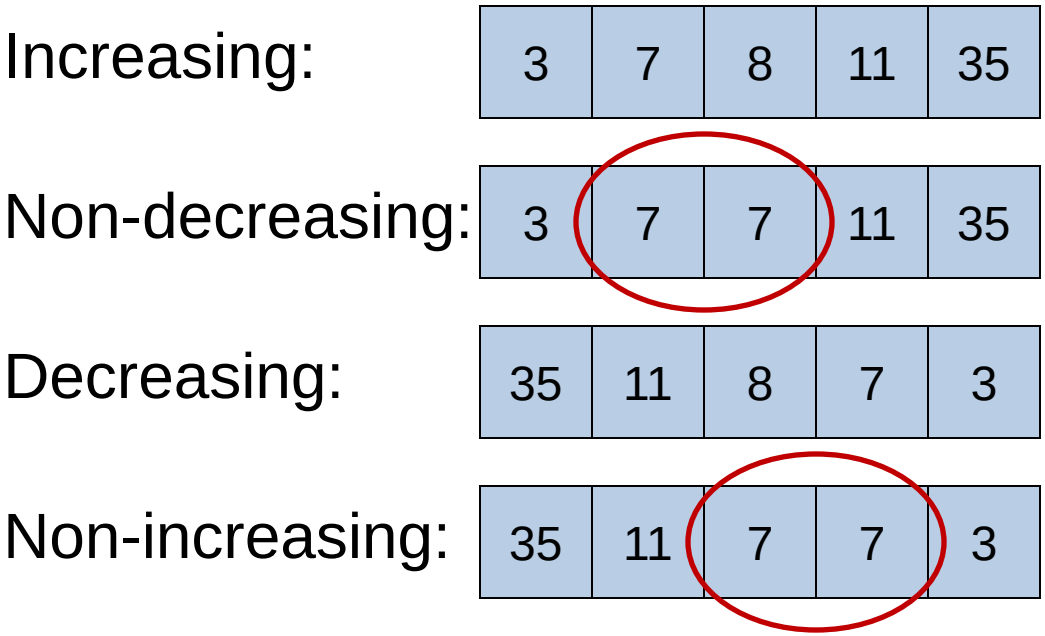

Periodische Gesetzmässigkeit der Elemente nach Mendeleieff

Reihen	Gruppe I R^2O	Gruppe II RO	Gruppe III R^2O^3	Gruppe IV RH^4 RO^2	Gruppe V RH^3 R^2O^5	Gruppe VI RH^2 RO^3	Gruppe VII RH R^2O^7	Gruppe VIII RO^4
1	H=1							
2	Li=7	Be=9,4	B=11	C=12	N=14	O=16	F=19	
3	Na=23	Mg=24	Al=27,3	Si=28	P=31	S=32	Cl=35,5	
4	K=39	Ca=40	Sc=44	Ti=48	V=51	Cr=52	Mn=55	Fe=56, Co=59 Ni=59, Cu=63
5	(Cu=63)	Zn=65	Ga=68	--=72	As=75	Se=79	Br=80	
6	Rb=85	Sr=87	Yt=88	Zr=90	Nb=94	Mo=96	--=100	Ru=104, Rh=104 Pd=106, Ag=108
7	(Ag=108)	Cd=112	In=113	Sn=118	Sb=122	Te=125	J=127	
8	Cs=133	Ba=137	Ce=137	La=139	–	Di=145?	–	– – – –
9	(–)	–	–	–	–	–	–	
10	–	165	– 169	Er=170	–173	Ta=182	W=184	– Pt=194, Os=195(?) Ir=193, Au=196
11	(Au=196)	Hg=200	Tl=204	Pb=206	Bi=210	–	–	
12	–	–	–	Th=231	–	U=240	–	– – – –

Oldest Periodic Table of Elements: Sorted elements in order of increasing atomic number (atomic mass)
https://www.bbc.com/bitesize/guides/z36cfcw/revision/1
https://www.theguardian.com/science/2019/jan/17/st-andrews-mldest-surviving-wall-chart-of-periodic-table-university

Introduction (1/2)

- To arrange the items in an array in increasing / decreasing order.
- E.g.,

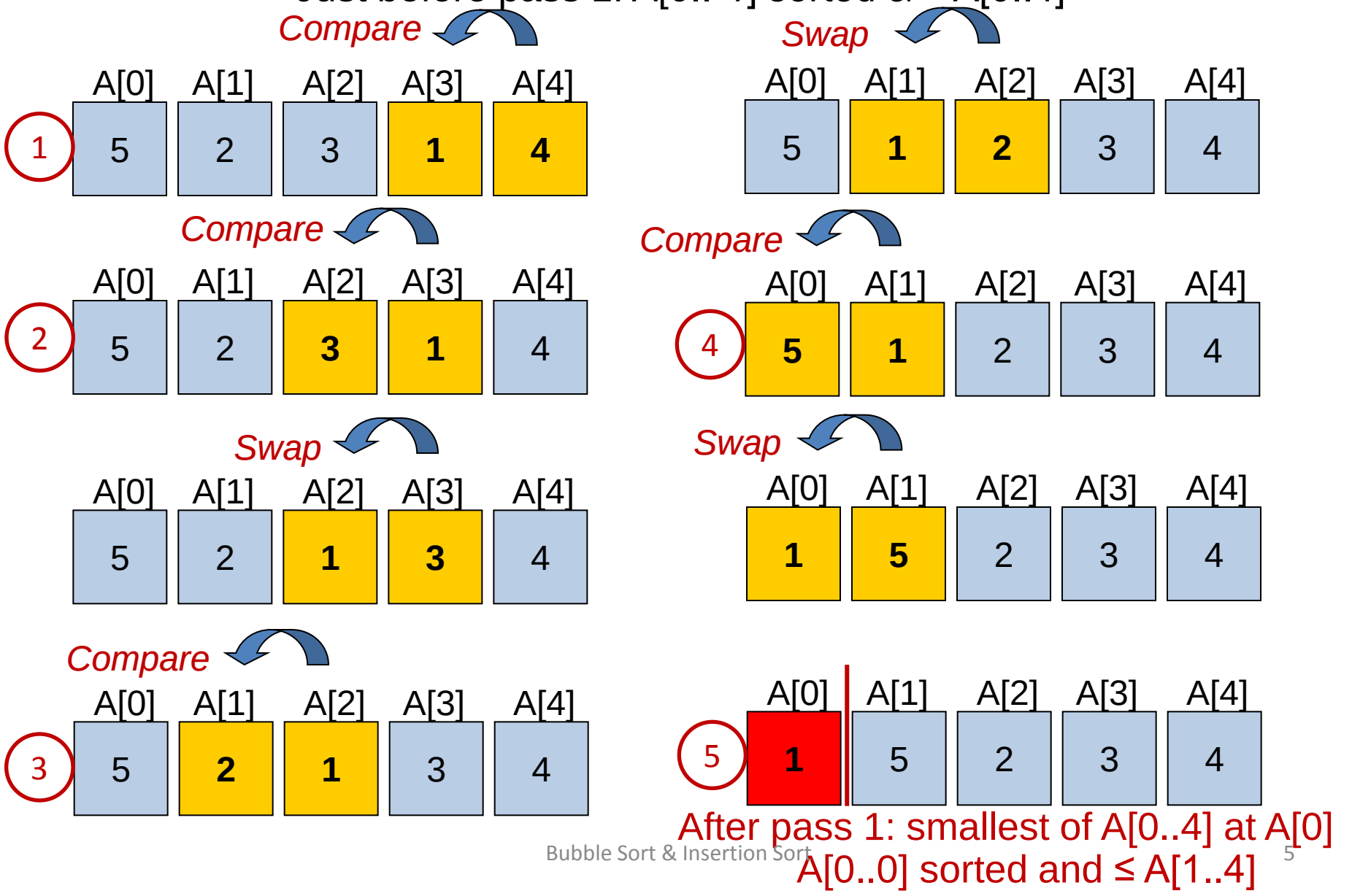


Introduction (2/2)

- One of the first studied computational problems; appears almost everywhere
- Algorithms:
  - Bubble sort, Insertion sort
  - Merge sort, Quick sort, Heap sort
  - Bucket & Radix sort
- Specific requirements:
  - Original array: int A[0..n-1]
  - After sorting, A[0..n-1] is a permutation of original A[0..n-1] in non-decreasing order

E.g. Bubble Sort (Pass 1)

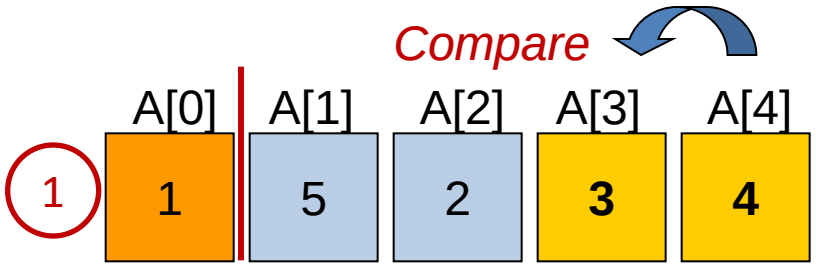
Just before pass 1: A[0..-1] sorted & ≤ A[0..4]

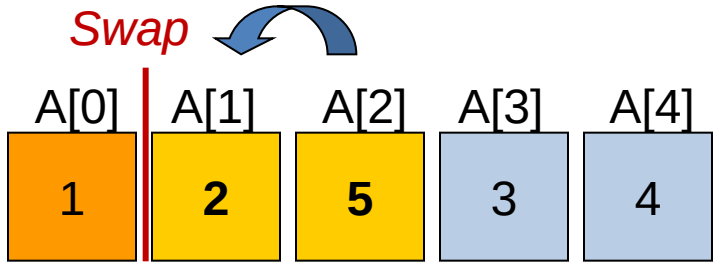


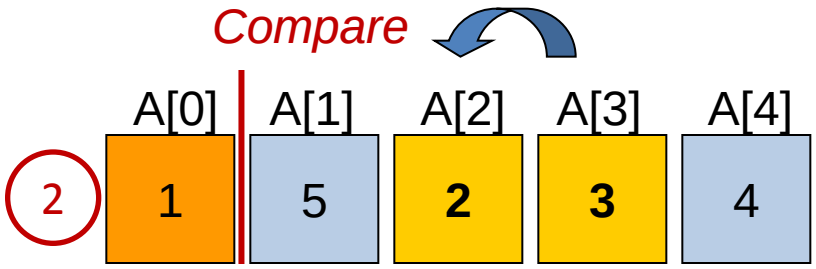
After pass 1: smallest of A[0..4] at A[0]
A[0..0] sorted and ≤ A[1..4]

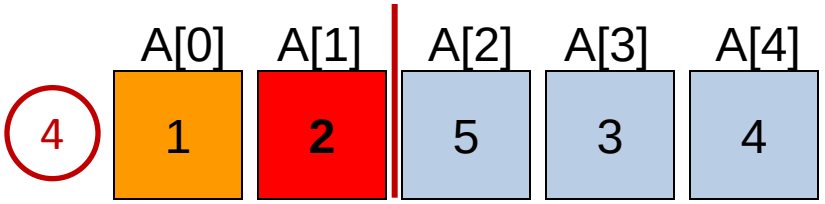
E.g. Bubble Sort (Pass 2)

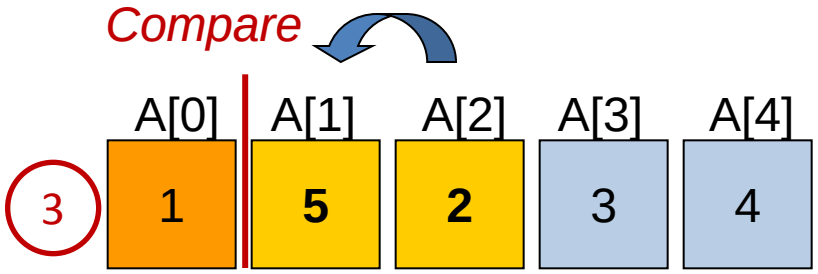
Just before pass 2: A[0..0] sorted & ≤ A[1..4]









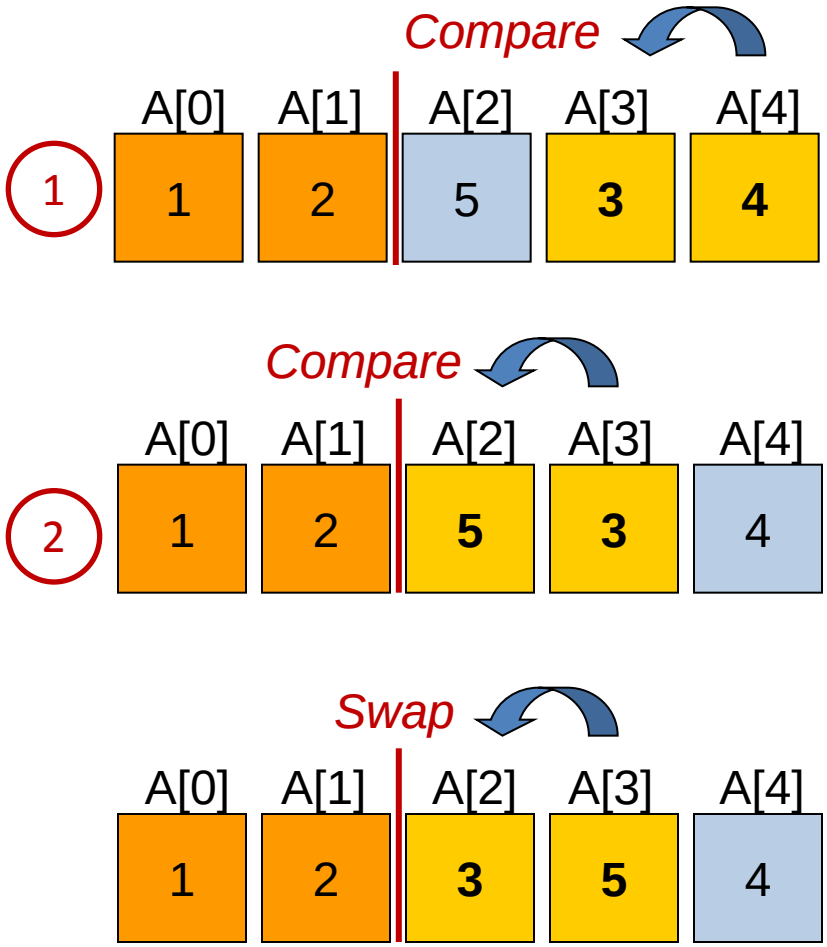


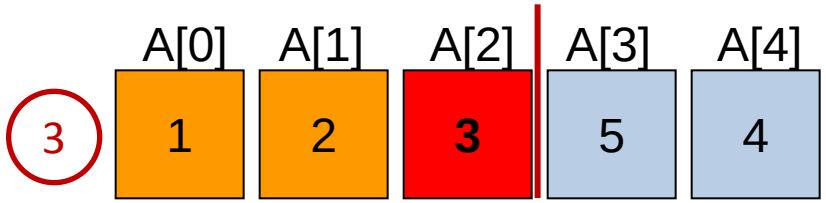
After pass 2: smallest of A[1..4] at A[1]

A[0..1] sorted and ≤ A[2..4]

E.g. Bubble Sort (Pass 3)

Just before pass 3: A[0..1] sorted & ≤ A[2..4]

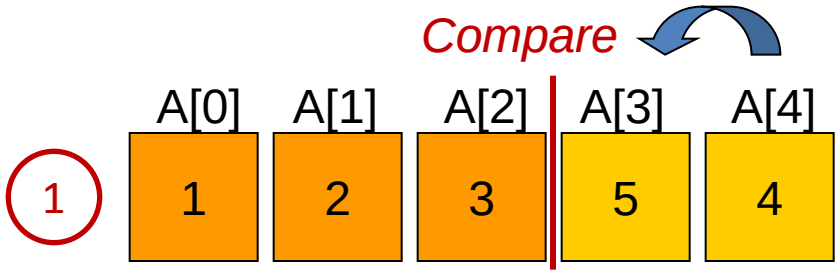


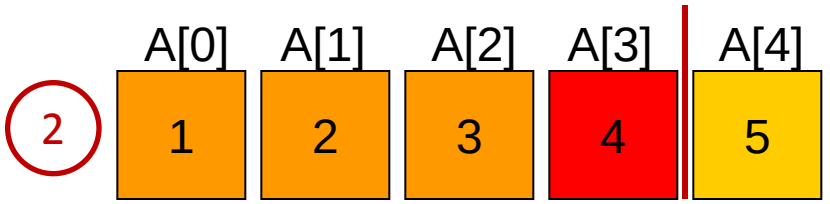


After pass 3: smallest of A[2..4] at A[2]
A[0..2] sorted and ≤ A[3..4]

E.g. Bubble Sort (Pass 4)

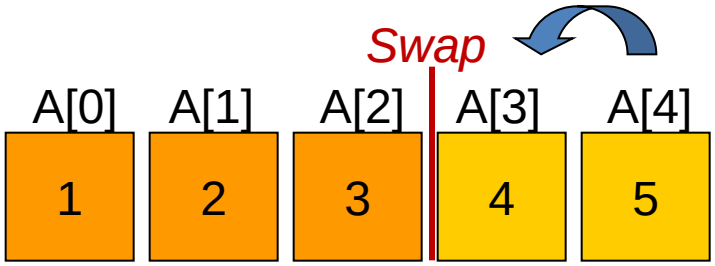
Just before pass 4: A[0..2] sorted & ≤ A[3..4]





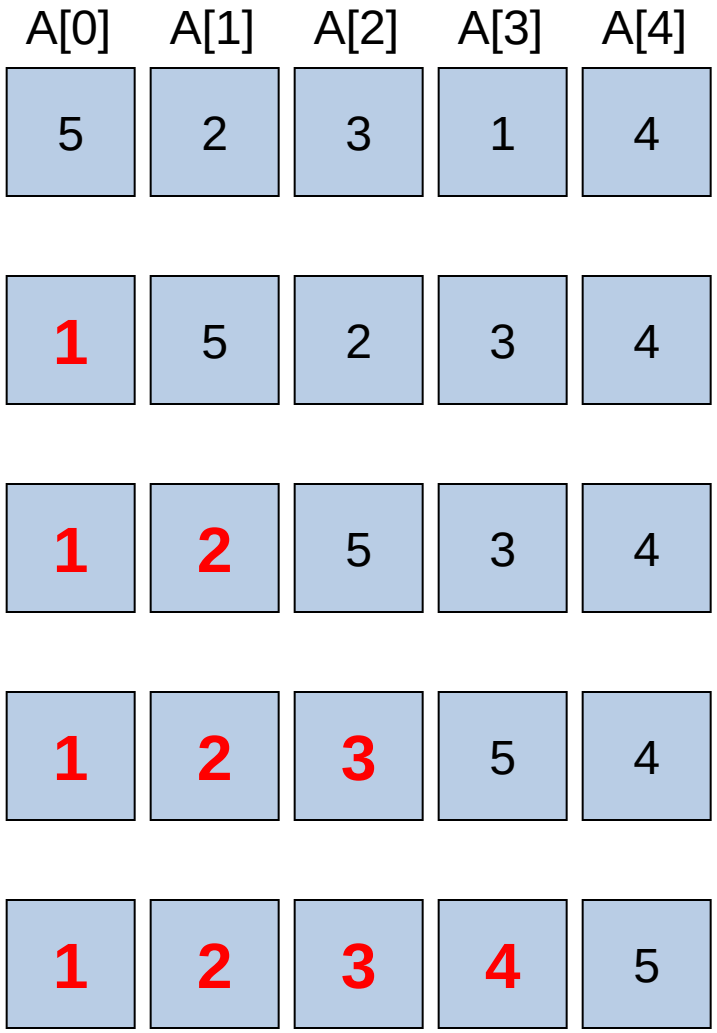
After pass 4: smallest of A[3..4] at A[3]

A[0..3] sorted and ≤ A[4]



Summary of the Example (1/2)

- Initially:
- After pass 1:
- After pass 2:
- After pass 3:
- After pass 4:



Summary of the Example (2/2)

- What do you observe?
  - Before pass i: (i.e., i=1, …, n-1 and A[0…-1] indicates an empty array)

    A[0…i-2] ≤ A[i-1…n-1]
    (Sorted) (Unsorted)

  - During pass i:

(Rearrange the smallest element in A[i-1..n-1] to A[i-1] by adjacent swaps)

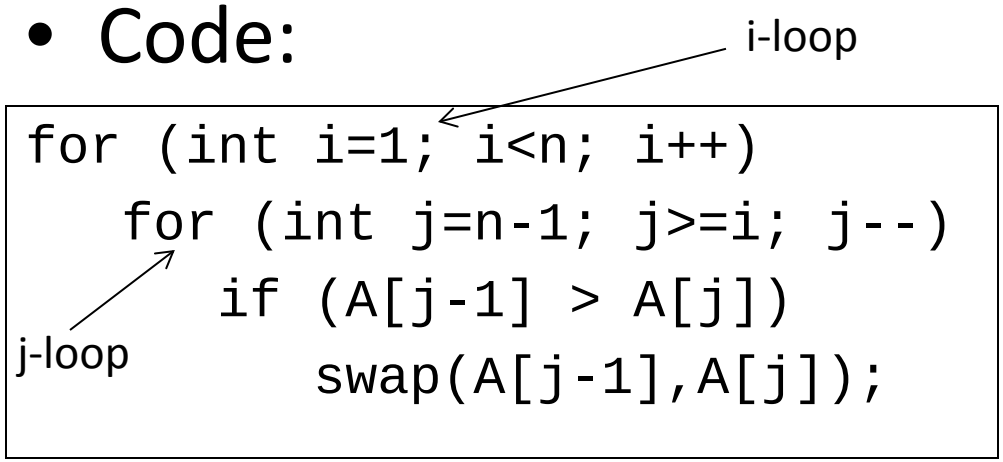
A[0…i-2] ≤ A[i-1] ≤ A[i…n-1]
(Sorted) (Unsorted)

  - After pass i:

A[0…i-1] ≤ A[i…n-1]
(Sorted) (Unsorted)

Bubble Sort

- Idea:
  - n-1 passes
  - For i = 1 through n-1,
    - Pass i scans A[i-1..n-1] from back to front, swapping adjacent pair A[j-1] & A[j] if A[j-1] > A[j]
- Code:



i	No. of iterations for j-loop
1	n-1 (i.e., j=n-1, …, 1)
2	n-2 (i.e., j=n-1, …, 2)
3	n-3 (i.e., j=n-1, …, 3)
…	…
n-1	1 (i.e., j=n-1)

Time Analysis

- Operations:
  - i-loop bookkeeping: init, loop test, increase i
  - j-loop bookkeeping: init, loop test, decrease j
  - Comparison of elements
  - Swapping elements
  - Assume each operation takes O(1) (i.e., constant) time
- Observations:
  - No. of loop body executions
    = no. of loop tests -1

Time Analysis

- Consider the j-loop:
  - j-loop body iterates n-i times
  - Each iteration performs a loop test (j>=i), an if-statement and decrement of j (j--)
  - So, each iteration uses constant time, denoted by c
  - j-loop initialization and last j-loop test take constant time, denoted by c'
  - So, j-loop uses ≤ c(n-i)+c' time

Operations	**No. of times**	**Unit cost**	**Cost**
Loop test, if-statement, j--, & swap	n-i	c	c(n-i)
j-loop initialization & last j-loop test	1	c'	c'
		Total cost=	**c(n-i) + c'**

- Consider the i-loop:
  - i-loop body iterates n-1 times
  - Each iteration performs a loop test (i<n), a j-loop and increment of i (i++)
  - The i-loop test and i++ together take constant time, denoted by b
  - So, each iteration uses ≤ c(n-i)+c'+b time
  - i-loop initialization and last i-loop test takes constant time b'

Operations	**No. of times**	**Unit cost**	**Cost**
Loop test, i++ & j-loop	n-1	b + [c(n-i)+c']	$\sum_{i=1}^{n-1}[c(n-i)+c']+b$
i-loop initialization & last i-loop test	1	b'	b'
	Total cost=	$\left(\sum_{i=1}^{n-1} c(n-i)+c'+b\right)+b'$	

Time Analysis

- Let the worst case time complexity of our bubble sort be $T_W(n)$:
  - Note: i goes from 1 to n-1. So,

$$T_w(n) \leq \left[\sum_{i=1}^{n-1} c(n-i) + c' + b\right] + b'$$

$$= \sum_{i=1}^{n-1} c(n-i) + \sum_{i=1}^{n-1} (c' + b) + b'$$

$$= c[(n-1) + \cdots + 1] + (c' + b)(n-1) + b'$$

$$= c\left[\frac{(n-1+1)(n-1)}{2}\right] + (c' + b)n + (b' - c' - b)$$

$$= \frac{c(n)(n-1)}{2} + (c' + b)n + (b' - c' - b)$$

$$= \frac{cn^2}{2} - \frac{cn}{2} + (c' + b)n + (b' - c' - b)$$

$$= O(n^2)$$

Arithmetic Progression

Note: Arithmetic Progression

- By definition, an arithmetic series is evenly distributed (i.e., the difference between the consecutive terms is constant), you could think of the average term as being the half way between the first term and the last one, so
  - Average term = (first term + last term)/2
- The sum of all the terms =

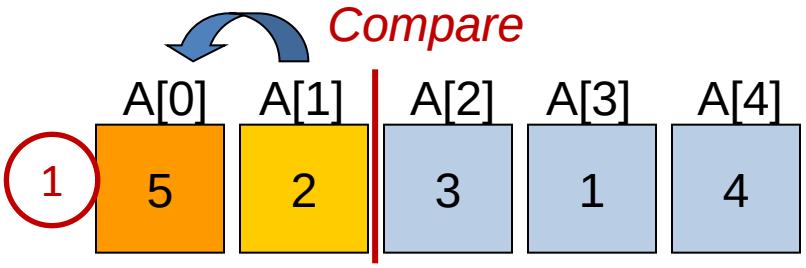
  (Number of terms)(Average term)
- $(a+0d)+(a+1d)+(a+2d)+\ldots+[a+(n-1)d]$, where $n \geq 0$

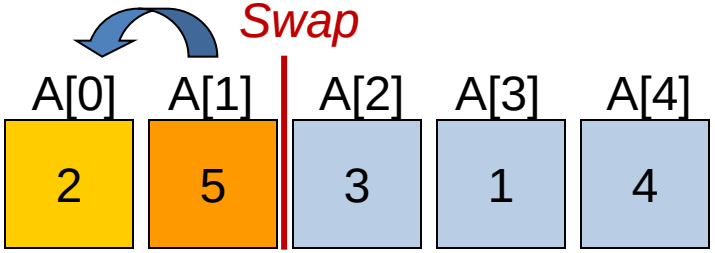
  Average term = $\{a+[a+(n-1)d]\}/2$

  The Sum = $(\{a+[a+(n-1)d]\}/2)n$

Insertion Sort (Pass 1)

Just before pass 1: A[0..0] is sorted version of old A[0..0]

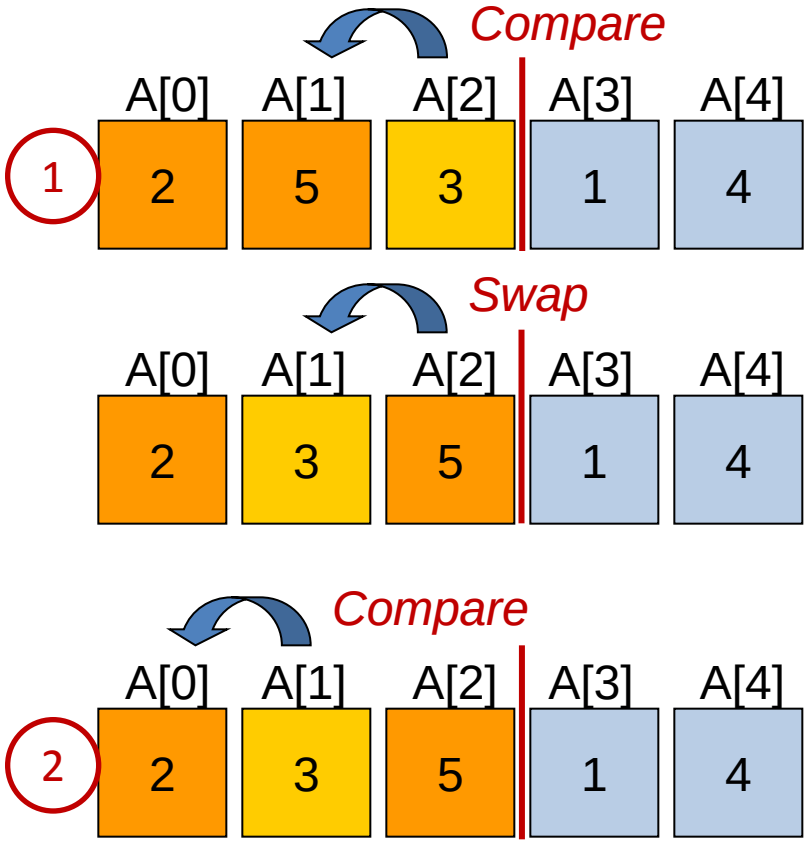




After pass 1: A[0..1] is sorted version of old A[0..1]
Pass 2 will consider A[2]

Insertion Sort (Pass 2)

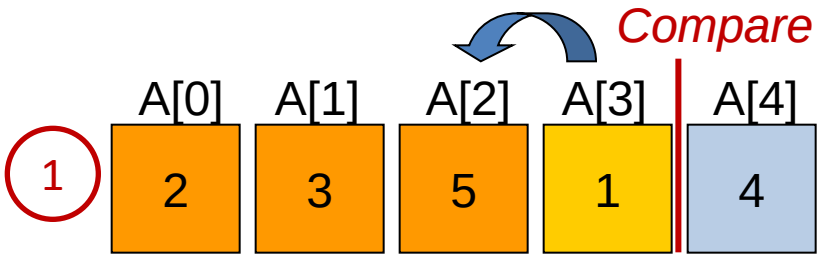
Just before pass 2: A[0..1] is sorted version of old A[0..1]

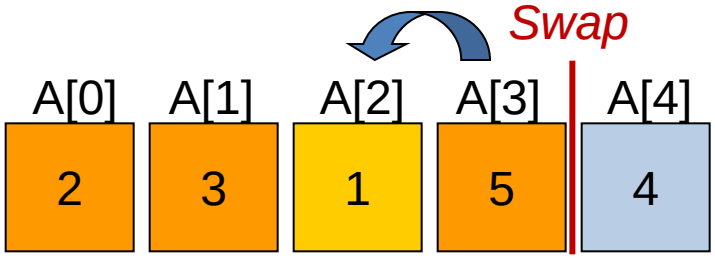


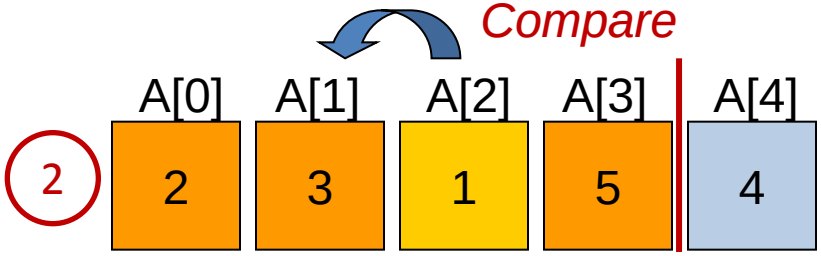
After pass 2: A[0..2] is sorted version of old A[0..2]
Pass 3 will consider A[3]

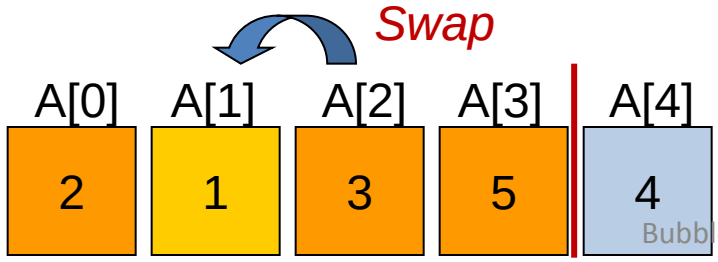
Insertion Sort (Pass 3)

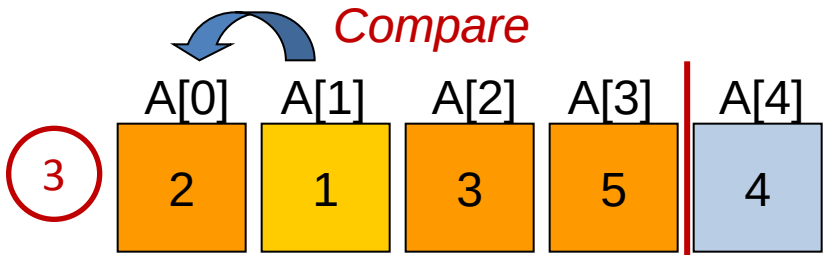
Just before pass 3: A[0..2] is sorted version of old A[0..2]

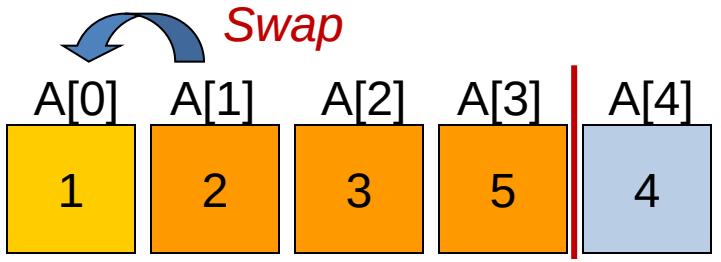








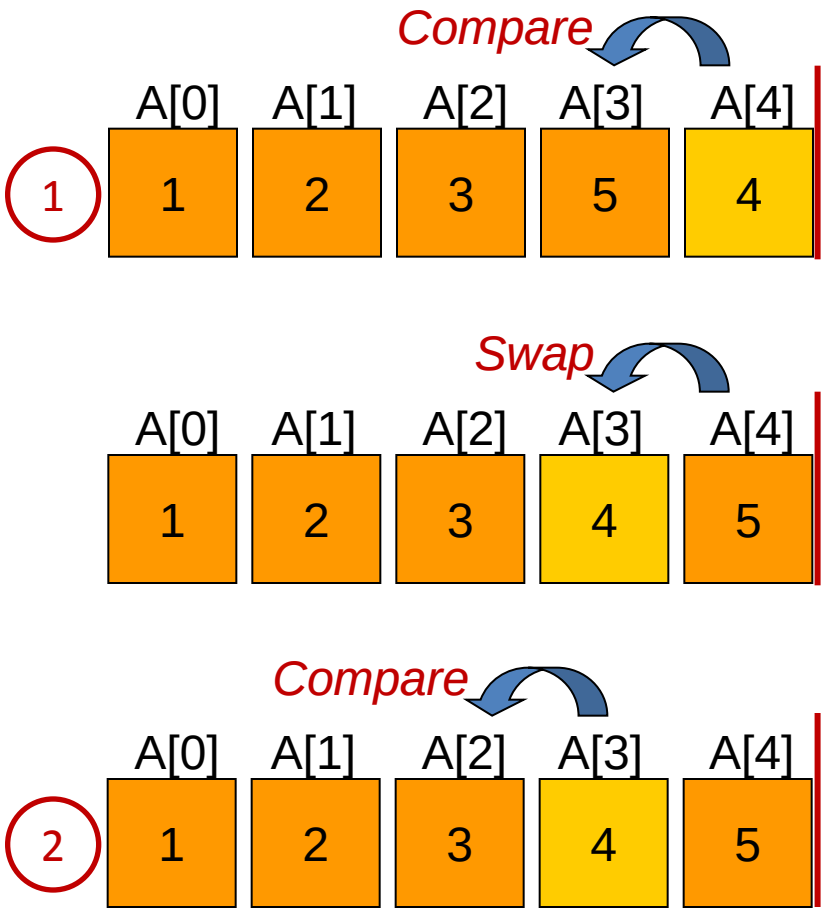


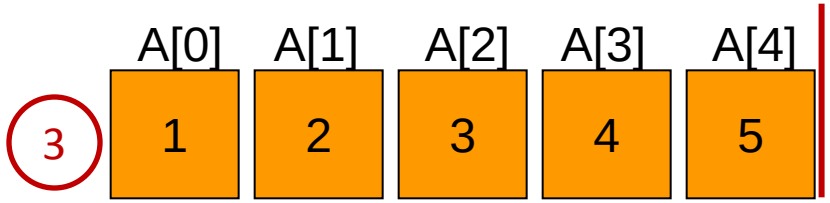


After pass 3: A[0..3] is sorted version of old A[0..3]
Pass 4 will consider A[4]

Insertion Sort (Pass 4)

Just before pass 4: A[0..3] is sorted version of old A[0..3]

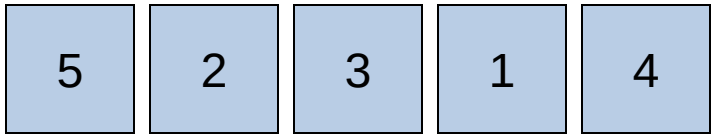




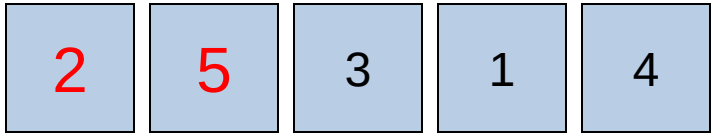
After pass 4: A[0..4] is sorted version of old A[0..4]

Summary of the Example

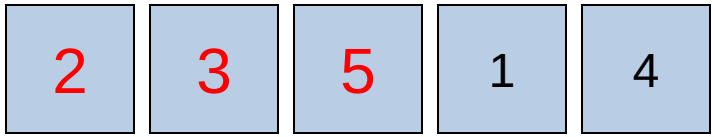
- Initially:



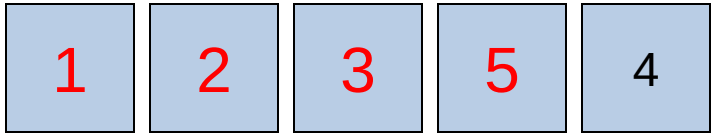
- After pass 1:



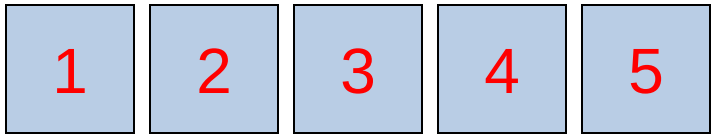
- After pass 2:



- After pass 3:



- After pass 4:



Insertion Sort

- Idea:
  - n-1 passes
  - For i = 1 through n-1,
    - Pass i ensures A[0..i] is a sorted version of original A[0..i]
- Code:

```
for (int i=1; i<n; i++)
{
   j=i;
   while (j>0 && A[j-1] > A[j])
   {
       swap(A[j-1],A[j]);
       j--;
   }
}
```

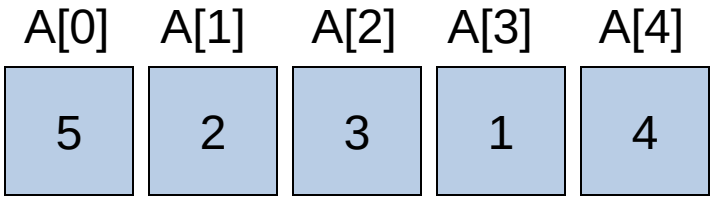
Theorem 1

- Can we design faster algorithms than bubble sort and insertion sort?
- If you perform only adjacent swaps,

  then NO!!!
- Theorem 1: Any sorting algorithm that sorts by only performing adjacent swaps requires $\Theta(n^2)$ time in the worst case

Proof of Theorem 1 (1/3)

- Definition: An *inversion* in an array A[0..n-1] of numbers is any ordered pair (i,j) such that i<j but A[i]>A[j]
- E.g., 6 inversions in:



check_inversion(0,1)=true
check_inversion(0,2)=true
check_inversion(0,3)=true
check_inversion(0,4)=true

check_inversion(1,2)=false
check_inversion(1,3)=true
check_inversion(1,4)=false

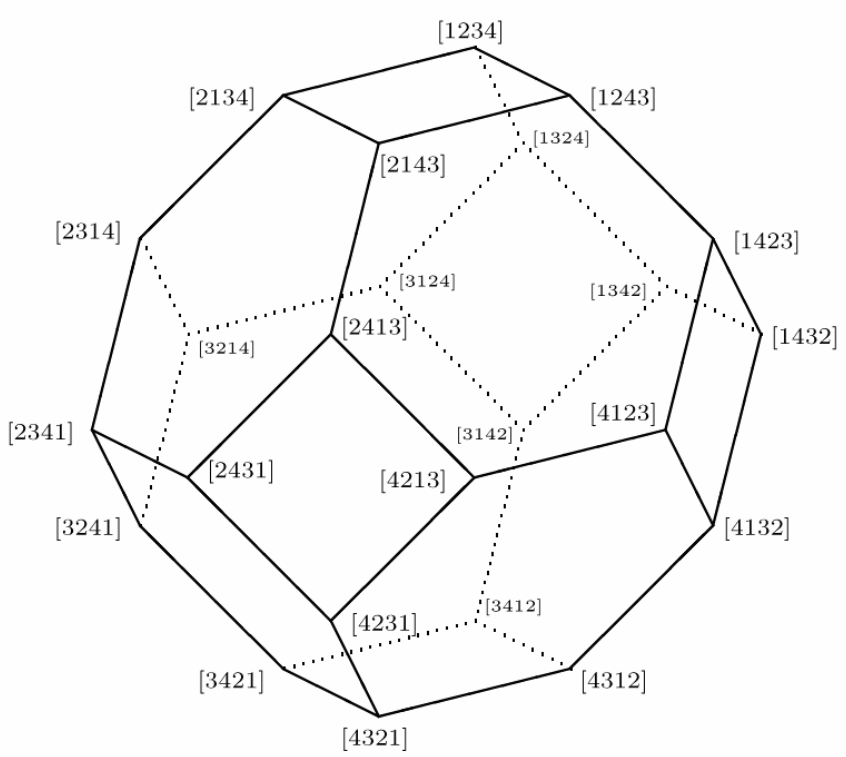
check_inversion(2,3)=true
check_inversion(2,4)=false

check_inversion(3,4)=false

check_inversion(i, j) = true, if i < j and A[i] > A[j]
check_inversion(i, j) = false, if i < j and A[i] < A[j]
return error, otherwise

Geometry of Inversions

- Example of permutation of four elements.
- Visualizes as a semi-octahedron where the number of inversions of a particular permutation is the length of a downward path from [1234] to that permutation.



Proof of Theorem 1 (2/3)

- Let X = no. of inversions in the original array
- An adjacent swap can only remove one inversion. Therefore,
  - Any algorithm that sorts by only performing adjacent swaps must perform at least X swaps
- Worst case: $X = \Theta(n^2)$
  - Suppose original array is sorted in reverse order
  - $X = (n-1) + (n-2) + \ldots + 1$

    $= n(n-1)/2$

    $= \Theta(n^2)$

Proof of Theorem 1 (3/3)

- The maximum number of inversions in a permutation (worst case)
- Consider a permutation n, n-1, n-2, …, 1 (the worst scenario)
  - For “n”, we have n-1 inversions
  - For “n-1”, we have n-2 inversions
  - For “n-2”, we have n-3 inversions
  - …
  - For “2”, we have 1 inversion
  - Thus, we have 1+2+3+…+(n-1) inversions, i.e., [1+(n-1)](n-1)/2 = n(n-1)/2

Note: Permutation

- In mathematics, the notion of permutation relates to the act of permuting (rearranging) objects or values.
- Informally, a permutation of a set of objects is an arrangement of those objects into a particular order.
- For example, there are six permutations of the set {1,2,3}, namely (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), and (3,2,1).
- The number of permutations of n distinct objects is “n factorial” usually written as “n!”, which means the product of all positive integers less than or equal to n.

Source: http://en.wikipedia.org/wiki/Permutation

Theorem 2

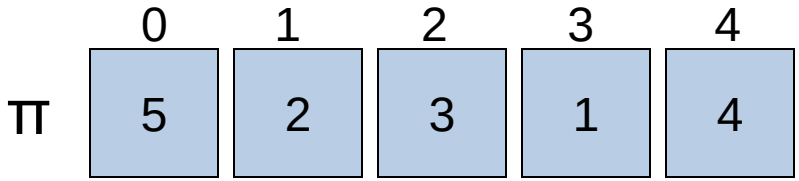
- Theorem 2: Any sorting algorithm that sorts by only performing adjacent swaps requires $\Theta(n^2)$ time in the average case
- Definition of "average case":
  - Assume input array is a permutation of {1, …, n}
  - For a fixed n, there are n! possible inputs
  - "Average case time complexity" means "average running time of these n! inputs"
- Note: Theorem 2 is *stronger* than Theorem 1, i.e., Theorem 2 implies Theorem 1
- Average case time complexity, $T_A(n)$:
  - The average case is no worse than the worst case, i.e, $T_A(n) \leq T_W(n)$

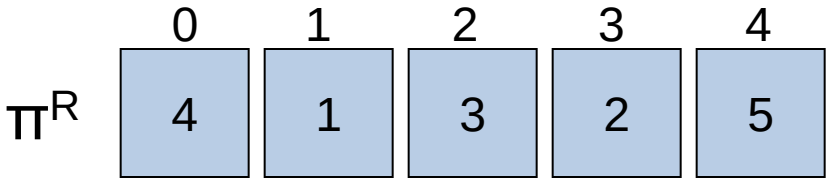
Proof of Theorem 2

- What is X on average?

  (let us bound E[X] by the average no. of inversions in the n! inputs?)
- Answer: $\Theta(n^2)$
- Why?
  - Consider an arbitrary permutation π and its reverse permutation π^R.
  - Total no. of inversions in π and π^R = $n(n-1)/2$

➢ An illustration:





1. check_inversion(0,1)=true
2. check_inversion(0,2)=true
3. check_inversion(0,3)=true
4. check_inversion(0,4)=true
5. check_inversion(1,2)=false
6. check_inversion(1,3)=true
7. check_inversion(1,4)=false
8. check_inversion(2,3)=true
9. check_inversion(2,4)=false
10. check_inversion(3,4)=false

1. check_inversion(3,4)=false
2. check_inversion(2,4)=false
3. check_inversion(1,4)=false
4. check_inversion(0,4)=false
5. check_inversion(2,3)=true
6. check_inversion(1,3)=false
7. check_inversion(0,3)=true
8. check_inversion(1,2)=false
9. check_inversion(0,2)=true
10. check_inversion(0,1)=true

Total no. of inversions of a (π, π^R) pair:
= (n-1)+(n-2)+…+1=(n-1+1)(n-1)/2
= n(n-1)/2

check_inversion(i, j) = true, if i < j and A[i] > A[j]
check_inversion(i, j) = false, if i < j and A[i] < A[j]
return error, otherwise

- (continuing):
  - There are n! permutations
  - Average no. of inversions in the n! permutations

    = (Total number of inversions in all n! permutations) / (n!)

    = Σ_{π} (no. of inversions in π) / (n!)

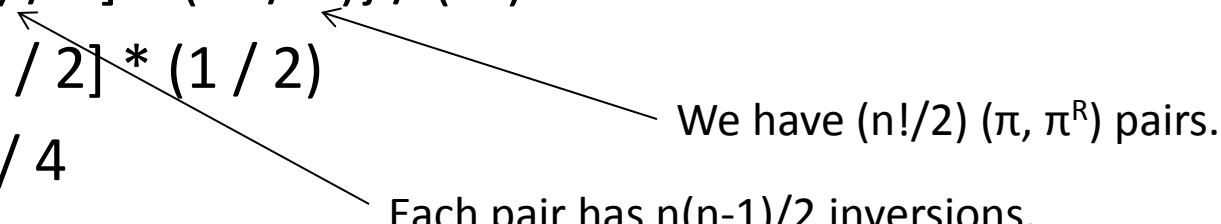
    = {[n(n - 1) / 2] * (n! / 2)} / (n!)

    = [n(n - 1) / 2] * (1 / 2)

    = n(n - 1) / 4

    = $n^2 / 4 - n / 4$

    thus $(n^2/4 - n/4) \leq E[X] \leq (n(n-1)/2)$



- So, average case time complexity is $\Theta(n^2)$

Software for Learning

- Visualization of a rich variety of data structures
- **“An algorithm must be seen to be believed”**
  - **Donald Knuth, Art of Computer Programming**
- https://visualgo.net/en/sorting
- https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

Sorting Since 1945

John Mauchly and J. Presper Eckert
- ENIAC (1946)
- Invented Insertion Sort (1946)
- UNIVAC - Information Age: Then and Now, Computer History Museum, 1960
https://www.youtube.com/watch?v=h4wQJfdhOlU

(1) A n+1 - complex: $X^{(p)}_{(x^0, x', \ldots, x^p)}$ consists of the main number: x^0 and the satellites: $x', \ldots, x^p$. Throughout what follows $p = 1, 2, \ldots$ will be fixed. A complex $X^{(p)}$ precedes a complex $Y^{(p)}$ $X^{(p)} \leq Y^{(p)}$, if their main numbers are in this order: $x^0 \leq y^0$.

An n- sequence of complexes: $\{X^{(p)}_0, \ldots, X^{(p)}_{n-1}\}$.

If $0', \ldots, (n-1)'$ is a permutation of 0, ..., (n-1), then the sequence $\{X^{(p)}_{0'}, \ldots, X^{(p)}_{(n-1)'}\}$ is a permutation of the sequence $\{X^{(p)}_0, \ldots, X^{(p)}_{n-1}\}$

A sequence $\{X^{(p)}_0, \ldots, X^{(p)}_{n-1}\}$ is monotone if its elements appear in their order of precedence: $X^{(p)}_0 \leq X^{(p)}_1 \leq \ldots \leq X^{(p)}_{n-1}$, i.e. $x^0_0 \leq x^0_1 \leq \ldots \leq x^0_{n-1}$.

Every sequence $\{X^{(p)}_0, \ldots, X^{(p)}_{n-1}\}$ possesses a monotone permutation: $\{X^{(p)}_{0'}, \ldots, X^{(p)}_{(n-1)'}\}$ (at least one). Obtaining this monotone permutation is the operation of sorting the original sequence.

Given two (separately) monotone sequences $\{X^{(p)}_0, \ldots, X^{(p)}_{m-1}\}$ and $\{Y^{(p)}_0, \ldots, Y^{(p)}_{m-1}\}$ sorting the composite sequence $\{X^{(p)}_0, \ldots, X^{(p)}_{m-1}, Y^{(p)}_0, \ldots, Y^{(p)}_{m-1}\}$ is the operation of meshing.

(2) We wish to formulate code instructions for sorting and for meshing, and to see how much control-capacity they tie up and how much time they require. It is convenient to consider meshing first and sorting afterwards.

First Stored Program https://www.amphilsoc.org/exhibits/treasures/vonneuma.htm

It is striking that von Neumann and Mauchly explain meshing in as much detail as its application to sorting, and highlight the use of the 'comparison operator' to discriminate between two data items. The automation of conditional control was a new and untested technology in 1945, and although the meshing procedure might appear trivial to us, we should not underestimate its novelty. As Knuth (1973, 384) suggested, implementing these non-numerical procedures did provide reassurance that, as John von Neumann (1945e) put it, "the present principles for the logical controls are sound". - Routines of Substitution, John von Neumann's Work on Software Development, 1945–1948, Mark Priestley (2018)

Trains of Thought: Data Structures at Play

1) Wikipedia on Permutation Inversions
https://en.wikipedia.org/wiki/Inversion_(discrete_mathematics)
2) https://www.cut-the-knot.org/books/Reviews/The15Puzzle.shtml
3) A modern treatment of the 15 Puzzle, Aaron F. Archer, American Mathematics Monthly, Vol. 106, No. 9, 1999
http://www.cs.cmu.edu/afs/cs/academic/class/15859-f01/www/notes/15-puzzle.pdf

How to compute the solution with fewest moves for this game, if it exists?

7	2	4
5		6
8	3	1