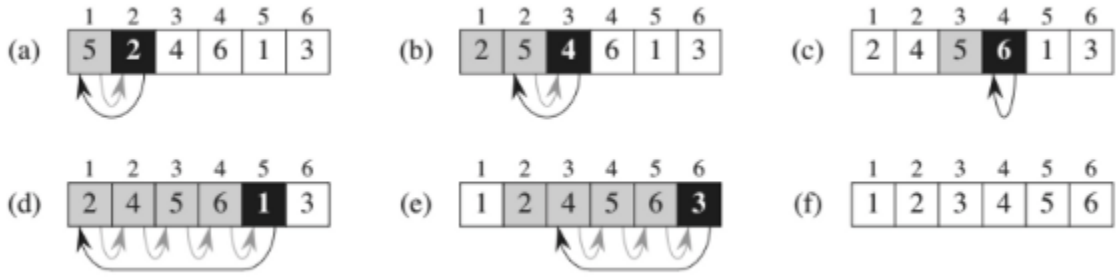
Insertion sort

Insertion sort works the way we sort a hand of playing cards: We start with an empty left hand and all remaining cards on the table. Then we remove one card at a time from the table [unsorted array] and insert it into the correct position in the left hand [sorted array]. To find the correct place for the card, we compare it with each card already in hand, from right to left.

The core idea is: If the first few array elements are sorted, we can easily insert the new element into its correct position in the sorted part. We need to compare the new element with each element in the sorted part from right to left.

Insertion sort visualization



Algorithm: INSERTION-SORT(A)

		cost	time	
1	for j ← 2 to A.size do	c1		n
2	key ← A[j]	c2		n-1
	// Insert A[j] into the sorted	0		n-1
	sequence A[1 . . j − 1]	c3		n-1
3	i ← j − 1	c4		n-1
			Min	Max
4	while i > 0 and A[i] > key do	c5	n-1	n*(n-1)/2
5	A[i + 1] ← A[i]	c6	0	n*(n-1)/2
6	i−−	c7	0	n*(n-1)/2
7	A[i + 1] ← key	c8		n-1

		J = 2	J = 3	……..	J = n	Total
Compare	Min	1	1		1	n-1
	Max	1	2		n-1	n*(n-1)/2
Swap	Min	0	0		0	0
	Max	1	2		n-1	n*(n-1)/2

Time and Space Complexity Analysis:

```
                  j=5
Case 1:  1 2 3 6 8
                <-
```

Minimum Compare = 1 Minimum Swap = 0

```
                  j=5
Case 2: 1 2 3 6 4
                <-
          1 2 3 4 6
              <-

                  j=5
Case 3: 1 2 3 6 0        1 2 3 0 6        1 2 0 3 6        1 0 2 3 6        0 1 2 3 6
                <-              <-              <-              <-              <-
```

Maximum Compare = j-1 Maximum Swap = j-1

So, Best case occurred when the array is already sorted and worst case occurred when the array is fully unsorted.

Best Case: T(n) = c1*n+ c2*(n-1)+ c3*(n-1)+ c4*(n-1)+ c5*1+ c6*0+ c7*0+ c8*(n-1)

We can express this running time as **an + b** for constants a and b that depend on the statement costs ci; it is thus a linear function of n.

Time complexity: O(n)

Worst Case: T(n) = c1*n+ c2*(n-1)+ c3*(n-1)+ c4*(n-1)+ c5*n*(n-1)/2+ c6*n*(n-1)/2+ c7*n*(n-1)/2+ c8*(n-1)

We can express this worst-case running time as **an^2 + bn + c** for constants a, b, and c that again depend on the statement costs ci; it is thus a quadratic function of n.

Time complexity: O(n^2)

Space Complexity: Above the algorithm, we are only creating 3 variables (j,key,i). This is the independent on n. It is constant.

Time complexity: O(1)

CLASS LECTURE