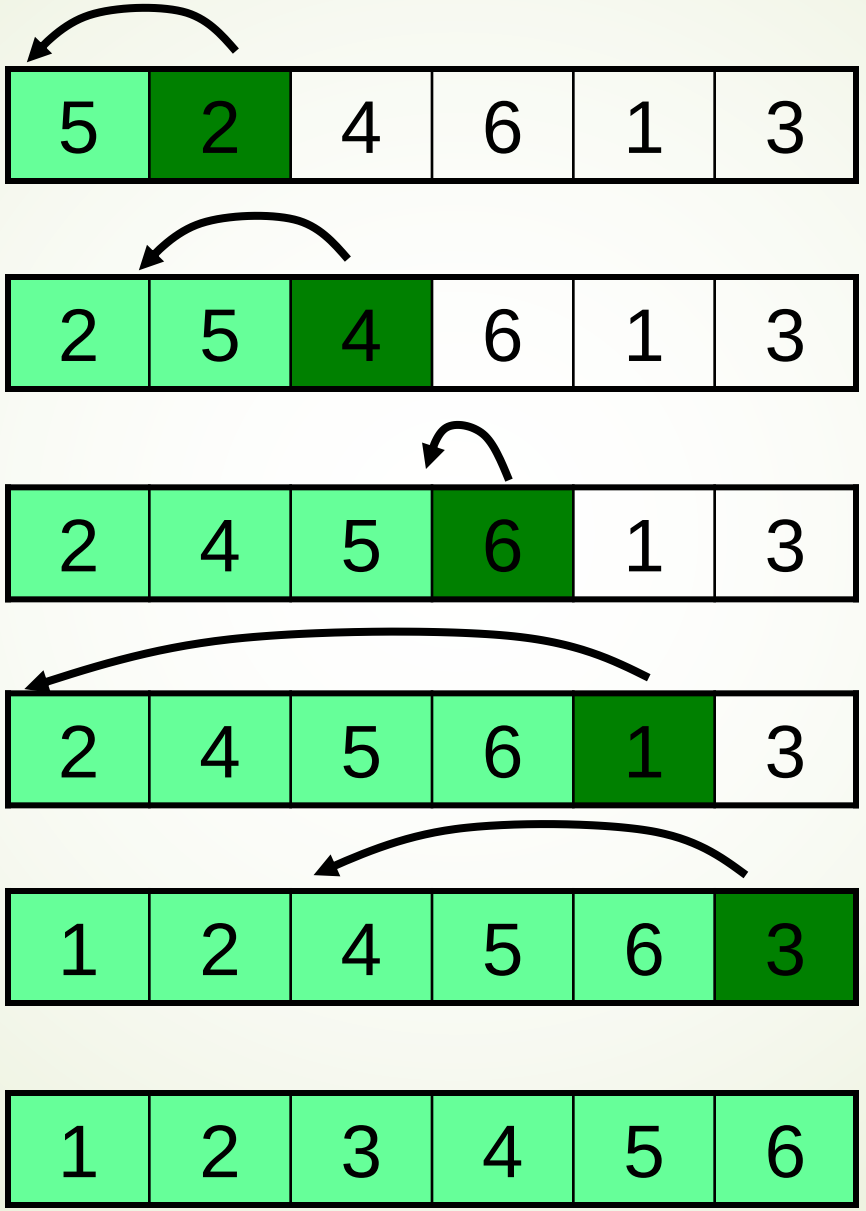
Running time of insertion sort

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

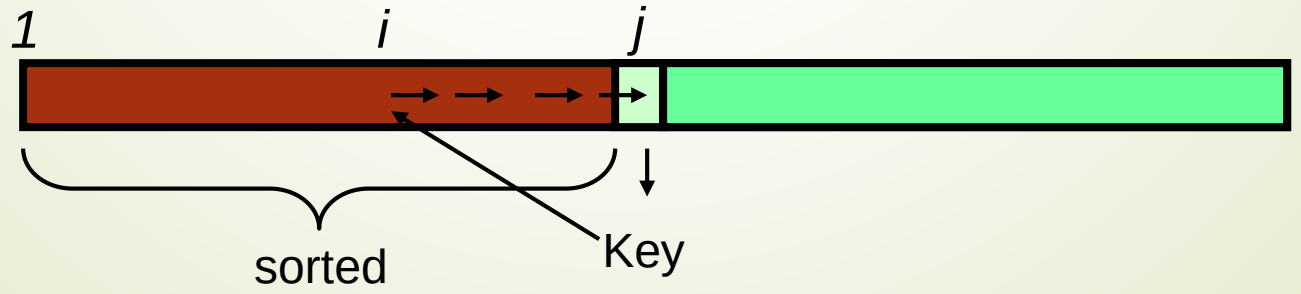
Example of insertion sort



Done!

Insertion Sort

```
InsertionSort(A, n) {
  for j = 2 to n {
    key = A[j];
    i = j - 1;
    while (i > 0) and (A[i] > key) {
      A[i+1] = A[i];
      i = i - 1;
    }
    A[i+1] = key
  }
}
```

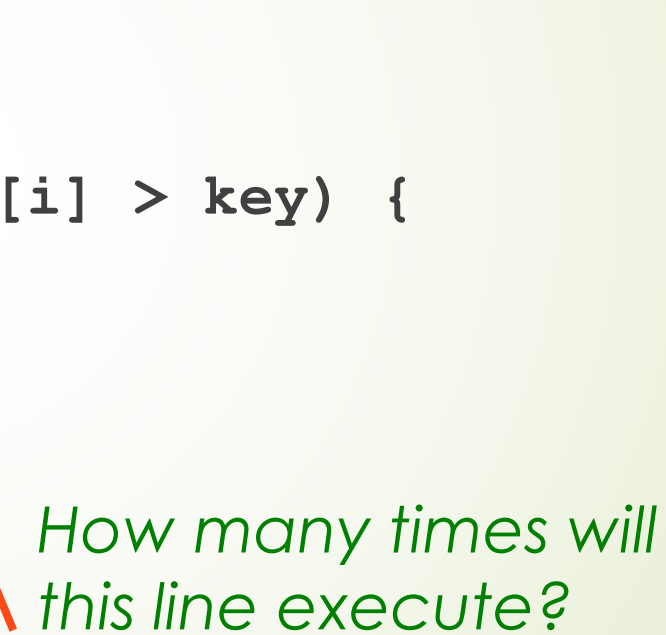


Kinds of analyses

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee
- Best case – not very useful
- Average case
  - Provides the expected running time
  - Very useful, but treat with care: what is "average"?
    - Random (equally likely) inputs
    - Real-life inputs

Analysis of insertion Sort

```
InsertionSort(A, n) {
   for j = 2 to n {
    key = A[j]
    i = j - 1;
    while (i > 0) and (A[i] > key) {
       A[i+1] = A[i]
       i = i - 1
    }
    A[i+1] = key
   }
}
```



How many times will this line execute?

Analysis of insertion Sort

```
InsertionSort(A, n) {
   for j = 2 to n {
    key = A[j]
    i = j - 1;
    while (i > 0) and (A[i] > key) {
       A[i+1] = A[i]
       i = i - 1
    }
    A[i+1] = key
   }
}
```

How many times will this line execute?

Analysis of insertion Sort

Statement	cost	time
`InsertionSort(A, n) {`		
`  for j = 2 to n {`	c_1	n
`   key = A[j]`	c_2	(n-1)
`   i = j - 1;`	c_3	(n-1)
`   while (i > 0) and (A[i] > key) {`	c_4	S
`      A[i+1] = A[i]`	c_5	(S-(n-1))
`      i = i - 1`	c_6	(S-(n-1))
`   }`	0	
`   A[i+1] = key`	c_7	(n-1)
`  }`	0	
`}`		

$S = t_2 + t_3 + \ldots + t_n$ where t_j is number of while expression evaluations for the j^{th} for loop iteration

Analysis of insertion Sort

Statement	cost	time
`InsertionSort(A, n) {`		
`  for j = 2 to n {`	c_1	n
`   key = A[j]`	c_2	(n-1)
`   i = j - 1;`	c_3	(n-1)
`   while (i > 0) and (A[i] > key) {`	c_4	S
`      A[i+1] = A[i]`	c_5	(S-(n-1))
`      i = i - 1`	c_6	(S-(n-1))
`   }`	0	
`   A[i+1] = key`	c_7	(n-1)
`  }`	0	
`}`		

What are the basic operations (most executed lines)?

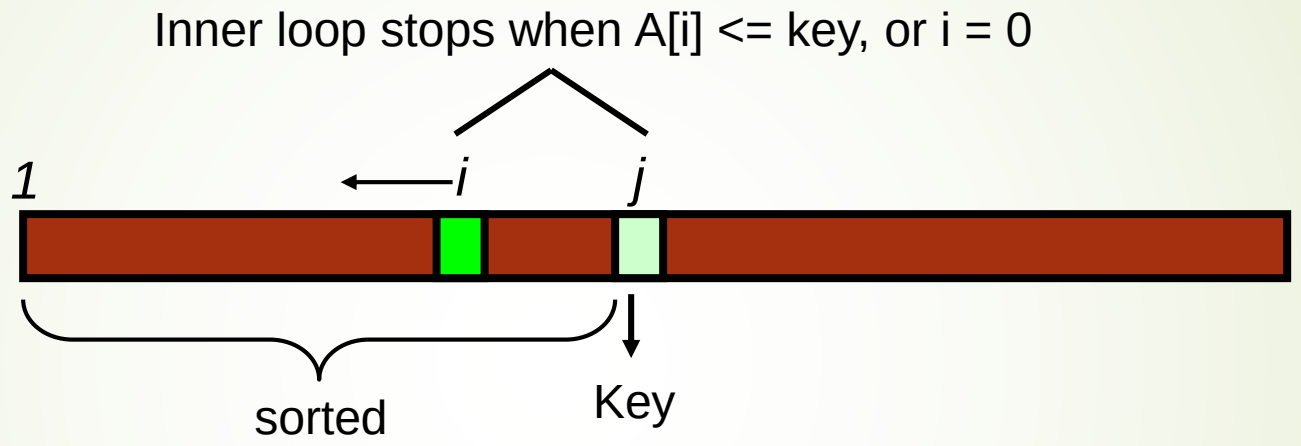
Analysis of insertion Sort

Statement	cost	time
`InsertionSort(A, n) {`		
`  for j = 2 to n {`	c_1	n
`  key = A[j]`	c_2	(n-1)
`  i = j - 1;`	c_3	(n-1)
`  while (i > 0) and (A[i] > key) {`	c_4	S
`  A[i+1] = A[i]`	c_5	(S-(n-1))
`  i = i - 1`	c_6	(S-(n-1))
`  }`	0	
`  A[i+1] = key`	c_7	(n-1)
`  }`	0	
`}`		

Analysis of insertion Sort

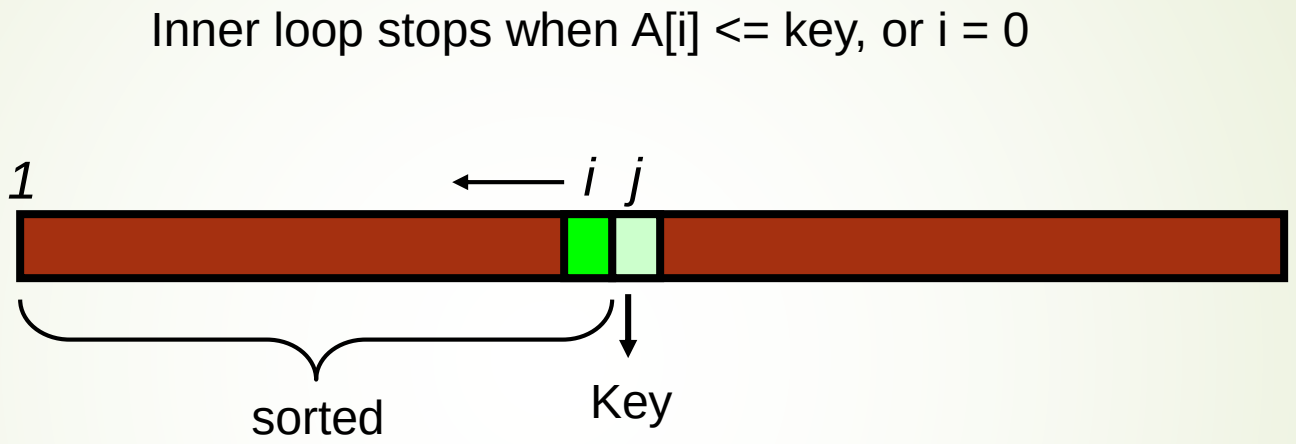
Statement	cost	time
`InsertionSort(A, n) {`		
`  for j = 2 to n {`	c_1	n
`   key = A[j]`	c_2	(n-1)
`   i = j - 1;`	c_3	(n-1)
`   while (i > 0) and (A[i] > key) {`	c_4	S
`      A[i+1] = A[i]`	c_5	(S-(n-1))
`      i = i - 1`	c_6	(S-(n-1))
`   }`	0	
`   A[i+1] = key`	c_7	(n-1)
`  }`	0	
`}`		

What can S be?



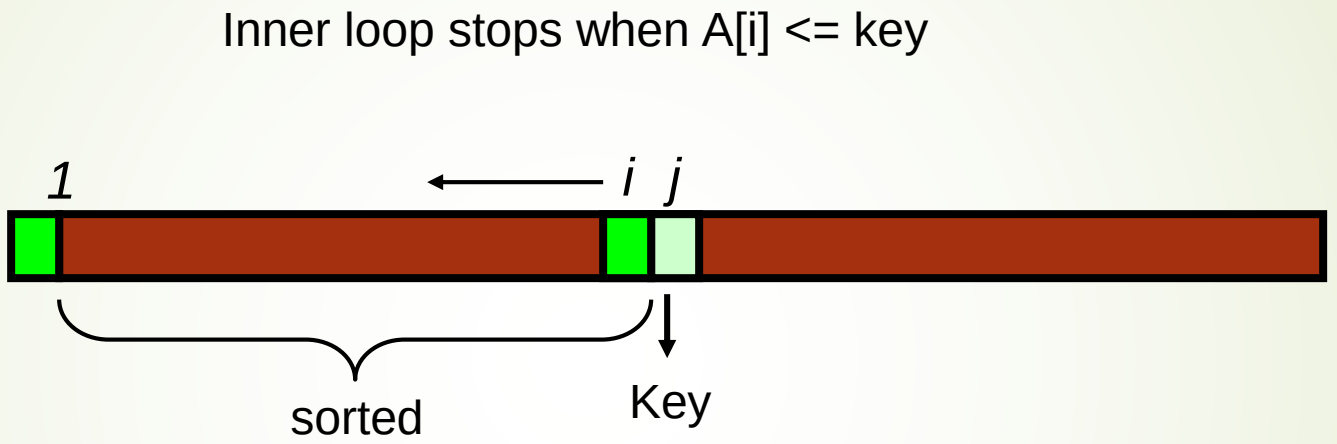
- $S = \sum_{j=2..n} t_j$
- Best case:
- Worst case:
- Average case:

Best case



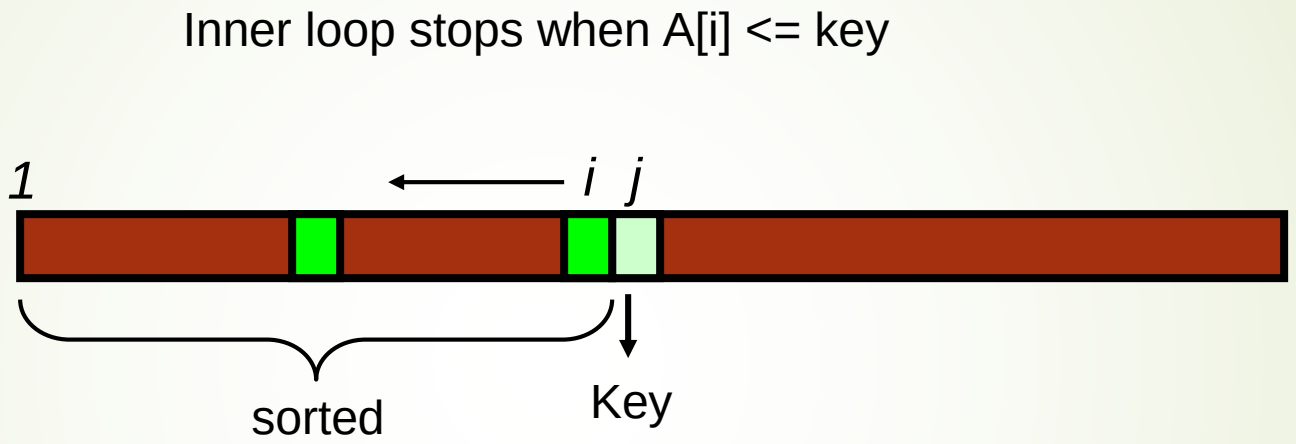
- Array already sorted
- $S = \sum_{j=2..n} t_j$
- $t_j = 1$ for all j
- S = n-1 $T(n) = \Theta(n)$

Worst case



- Array originally in reverse order sorted
- $S = \sum_{j=2..n} t_j$
- $t_j = j$
- $S = \sum_{j=2..n} j = 2 + 3 + \ldots + n = (n-1)\ (n+2) / 2 = \Theta\ (n^2)$

Average case



- Array in random order
- $S = \sum_{j=2..n} t_j$
- $t_j = j / 2$ on average
- $S = \sum_{j=2..n} j/2 = ½ \sum_{j=2..n} j = (n-1)(n+2) / 4 = \Theta(n^2)$