Computational problems

- A computational problem specifies an input-output relationship
  - What does the input look like?
  - What should the output be for each input?
- Example:
  - Input: an integer number n
  - Output: Is the number prime?
- Example:
  - Input: A list of names of people
  - Output: The same list sorted alphabetically

Algorithms

- A tool for solving a well-specified computational problem

- Algorithms must be:
  - Correct: For each input produce an appropriate output
  - Efficient: run as quickly as possible, and use as little memory as possible – more about this later

Algorithms Cont.

- A well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
- Written in a pseudo code which can be implemented in the language of programmer's choice.

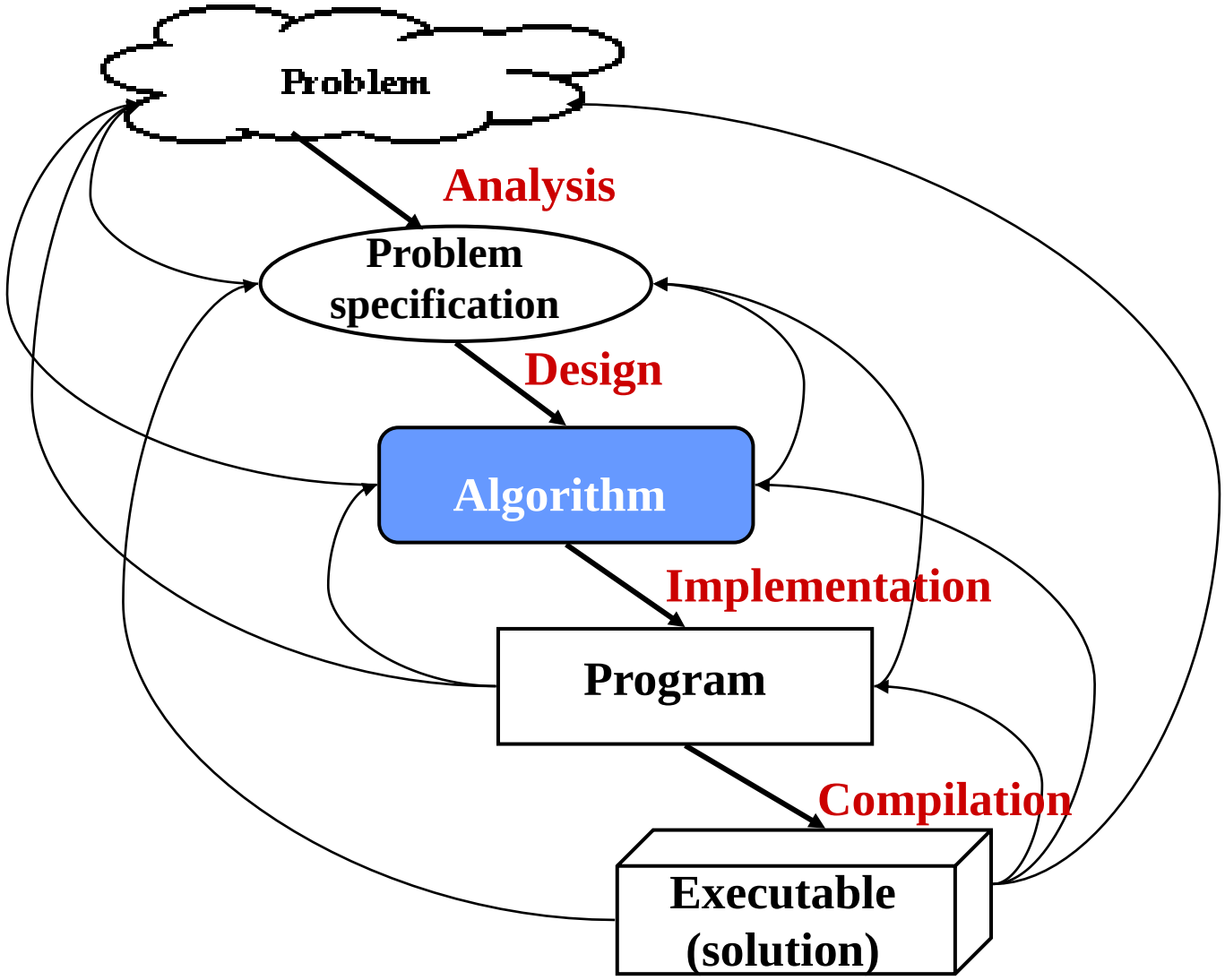
Correct and incorrect algorithms

- Algorithm is correct if, for every input instance, it ends with the correct output. We say that a correct algorithm solves the given computational problem.
- An incorrect algorithm might not end at all on some input instances, or it might end with an answer other than the desired one.
- We shall be concerned only with correct algorithms.

Properties of algorithms

- Algorithms generally share a set of properties:
  - Input: what the algorithm takes in as input
  - Output: what the algorithm produces as output
  - Definiteness: the steps are defined precisely
  - Correctness: should produce the correct output
  - Finiteness: the steps required should be finite
  - Effectiveness: each step must be able to be performed in a finite amount of time
  - Generality: the algorithm *should* be applicable to all problems of a similar form

The Problem-solving Process



Analysis of Algorithms

- An *algorithm* is a finite set of precise instructions for performing a computation or for solving a problem.
- What is the goal of analysis of algorithms?
  - To compare algorithms mainly in terms of running time but also in terms of other factors (e.g., memory requirements, programmer's effort etc.)
- What do we mean by running time analysis?
  - **Determine how running time increases as the size of the problem increases**.

Example: Searching

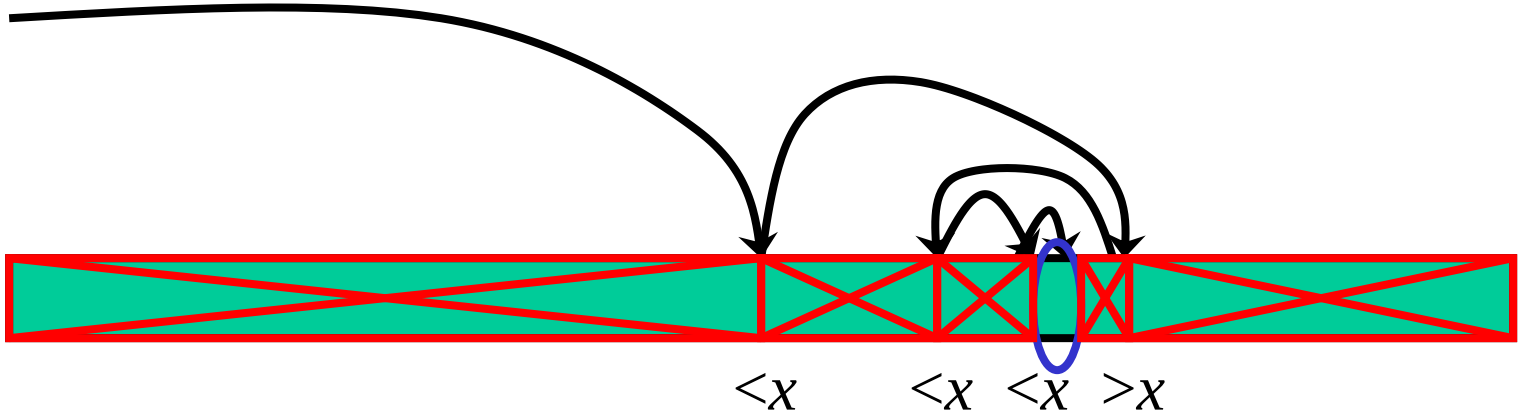
- Problem of *searching an ordered list*.
  - Given a list L of n elements that are sorted into a definite order (*e.g.*, numeric, alphabetical),
  - And given a particular element x,
  - Determine whether x appears in the list, and if so, return its index (position) in the list.

Search alg. #1: Linear Search

procedure *linear search*
(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while $(i \leq n \wedge x \neq a_i)$
    $i := i + 1$
if $i \leq n$ **then** *location* $:= i$
else *location* $:= 0$
return *location* {index or 0 if not found}

Search alg. #2: Binary Search

- Basic idea: On each step, look at the *middle* element of the remaining list to eliminate half of it, and quickly zero in on the desired element.



Search alg. #2: Binary Search

procedure *binary search*
(x:integer, a_1, a_2, …, a_n: distinct integers)
$i := 1$ {left endpoint of search interval}
$j := n$ {right endpoint of search interval}
while $i<j$ **begin** {while interval has >1 item}
$m := \lfloor (i+j)/2 \rfloor$ {midpoint}
if $x>a_m$ **then** $i := m+1$ **else** $j := m$
end
if $x = a_i$ **then** *location* $:= i$ **else** *location* $:= 0$
return *location*

Is Binary Search more efficient?

- **Number of iterations**:
  - For a list of *n* elements, Binary Search can execute at most $\log_2 n$ times!!
  - Linear Search, on the other hand, can execute up to *n* times !!

Average Number of Iterations

Length	Linear Search	Binary Search
10	5.5	2.9
100	50.5	5.8
1,000	500.5	9.0
10,000	5000.5	12.0

Is Binary Search more efficient?

- **Number of computations per iteration**:
  - Binary search does more computations than Linear Search per iteration.
- **Overall**:
  - If the number of components is small (say, less than 20), then Linear Search is faster.
  - If the number of components is large, then Binary Search is faster.

How do we analyze algorithms?

- We need to define a number of objective measures.

(1) Compare execution times?

Not good: times are specific to a particular computer !!

(2) Count the number of statements executed?

Not good: number of statements vary with the programming language as well as the style of the individual programmer.

Example (# of statements)

Algorithm 1

```
arr[0] = 0;
arr[1] = 0;
arr[2] = 0;
 ...
arr[N-1] = 0;
```

Algorithm 2

```
for(i=0; i<N; i++)
  arr[i] = 0;
```

How do we analyze algorithms?

(3) Express running time as a function of the input size n (i.e., $f(n)$).

– To compare two algorithms with running times $f(n)$ and $g(n)$, we need a **rough measure** of **how fast a function grows**.

– Such an analysis is independent of machine time, programming style, etc.

Computing running time

- Associate a "cost" with each statement and find the "total cost“ by finding the total number of times each statement is executed.
- Express running time in terms of the size of the problem.

Algorithm 1

	Cost
arr[0] = 0;	c1
arr[1] = 0;	c1
arr[2] = 0;	c1
...	
arr[N-1] = 0;	c1

c1+c1+...+c1 = c1 x N

Algorithm 2

	Cost
for(i=0; i<N; i++)	c2
arr[i] = 0;	c1

(N+1) x c2 + N x c1 =
(c2 + c1) x N + c2

Computing running time (cont.)

	Cost
sum = 0;	c1
for(i=0; i<N; i++)	c2
for(j=0; j<N; j++)	c2
sum += arr[i][j];	c3

$$c1 + c2 \times (N+1) + c2 \times N \times (N+1) + c3 \times N \times N$$

Comparing Functions Using Rate of Growth

- Consider the example of buying *elephants* and *goldfish:*

  Cost: cost_of_elephants + cost_of_goldfish

  Cost ~ cost_of_elephants (**approximation**)

- The low order terms in a function are relatively insignificant for **large** n

$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

i.e., $n^4 + 100n^2 + 10n + 50$ and n^4 have the same rate of growth

Rate of Growth ≡Asymptotic Analysis

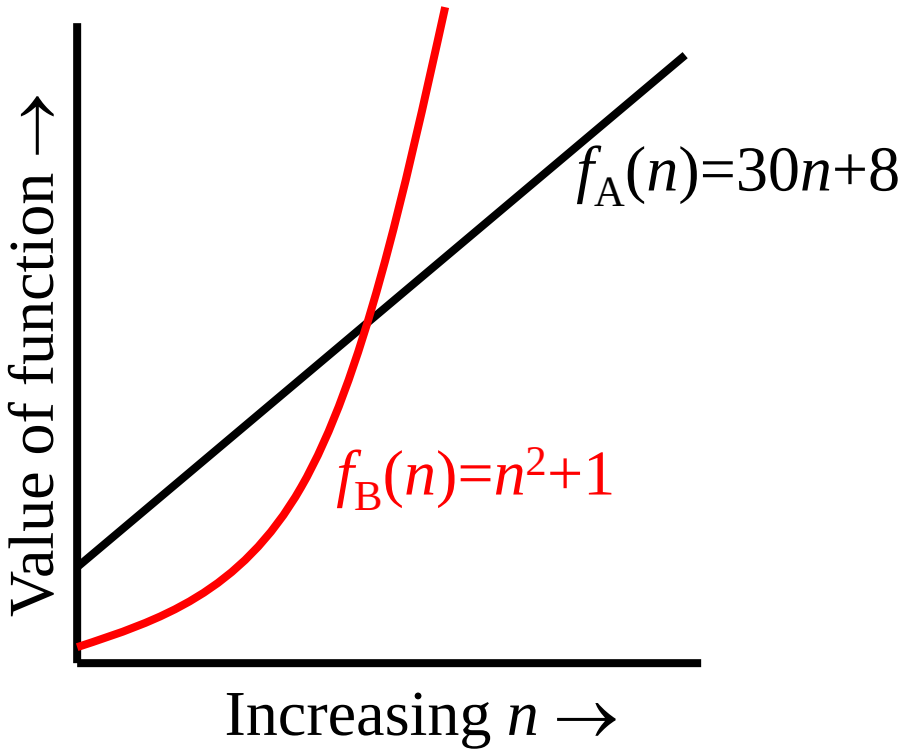
- Using *rate of growth* as a measure to compare different functions implies comparing them **asymptotically**.
- If $f(x)$ is *faster growing* than $g(x)$, then $f(x)$ always eventually becomes larger than $g(x)$ **in the limit** (for large enough values of x).

Example

- Suppose you are designing a web site to process user data (*e.g.*, financial records).
- Suppose program A takes $\boldsymbol{f_A(n)=30n+8}$ microseconds to process any n records, while program B takes $\boldsymbol{f_B(n)=n^2+1}$ microseconds to process the n records.
- Which program would you choose, knowing you'll want to support millions of users?

Visualizing Orders of Growth

- On a graph, as you go to the right, a faster growing function eventually becomes larger...



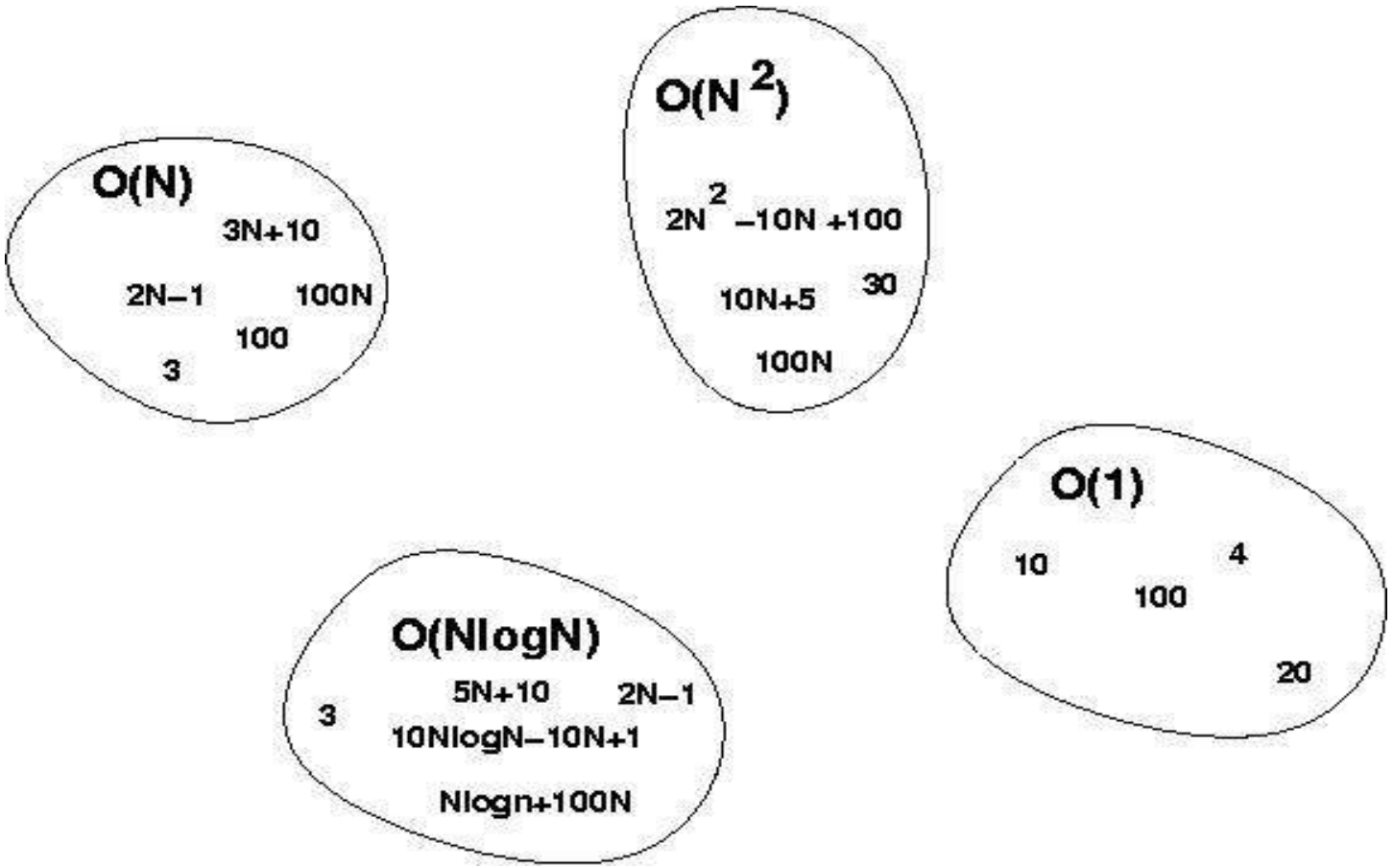
Big-O Notation

- We say $f_A(n)=30n+8$ is *order n*, or O(n). It is, **at most**, roughly *proportional* to n.
- $f_B(n)=n^2+1$ is *order* n^2, or O(n^2). It is, **at most**, roughly proportional to n^2.
- In general, an O(n^2) algorithm will be slower than O*(n)* algorithm.
- **Warning**: an O(n^2) function will grow faster than an O*(n)* function.

More Examples …

- We say that $n^4 + 100n^2 + 10n + 50$ is of the order of n^4 or $O(n^4)$
- We say that $10n^3 + 2n^2$ is $O(n^3)$
- We say that $n^3 - n^2$ is $O(n^3)$
- We say that 10 is $O(1)$,
- We say that 1273 is $O(1)$

Big-O Visualization



Computing running time

Algorithm 1

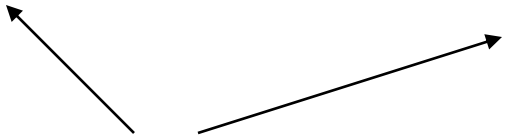
	Cost
arr[0] = 0;	c1
arr[1] = 0;	c1
arr[2] = 0;	c1
...	
arr[N-1] = 0;	c1

c1+c1+...+c1 = c1 x N

Algorithm 2

	Cost
for(i=0; i<N; i++)	c2
arr[i] = 0;	c1

(N+1) x c2 + N x c1 =
(c2 + c1) x N + c2



O(n)

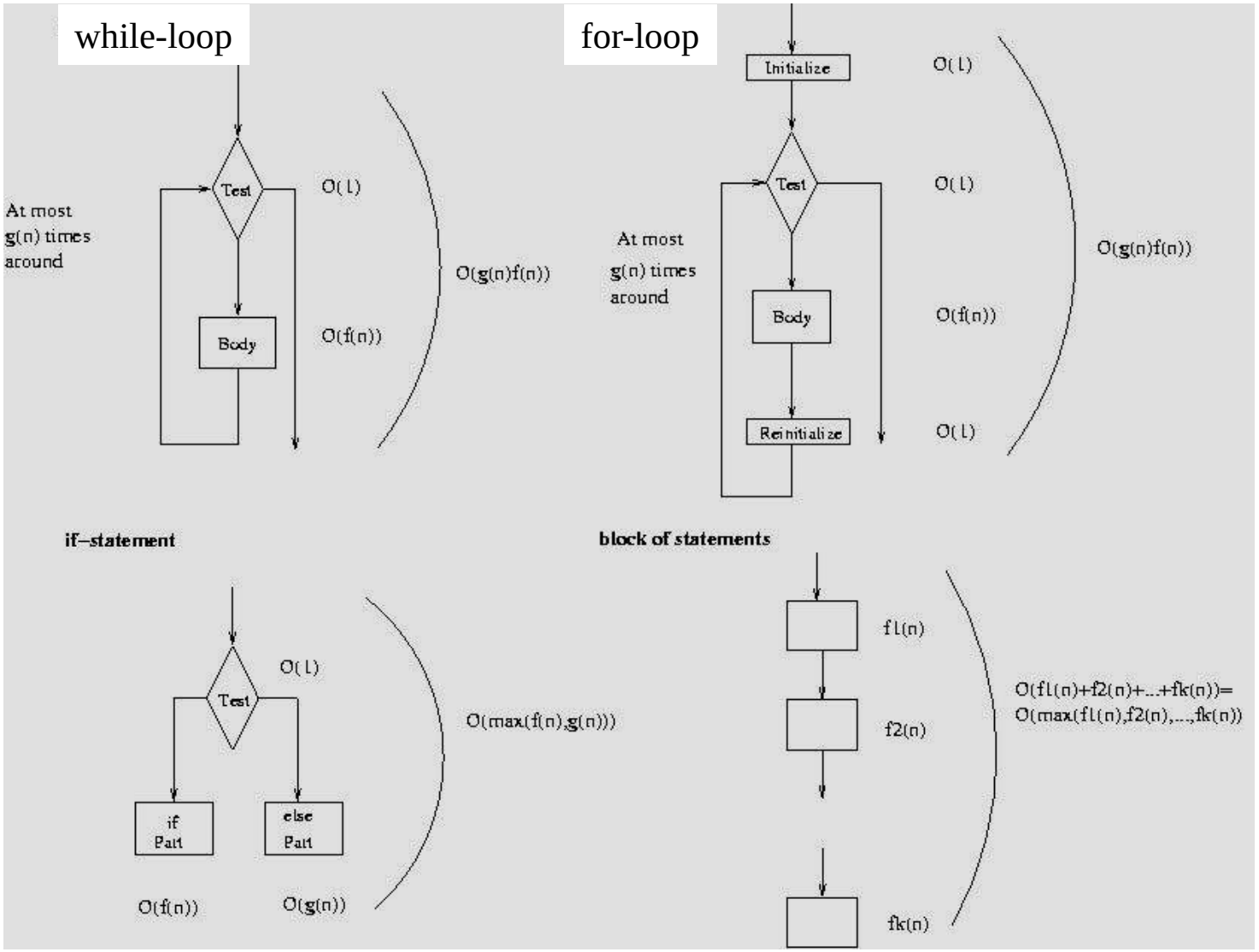
Computing running time (cont.)

	Cost
sum = 0;	c1
for(i=0; i<N; i++)	c2
for(j=0; j<N; j++)	c2
sum += arr[i][j];	c3

$$c1 + c2 \times (N+1) + c2 \times N \times (N+1) + c3 \times N \times N$$

$O(n^2)$

Running time of various statements



Examples

```
i = 0;
while (i<N) {
    X=X+Y;                    // O(1)
    result = mystery(X);      // O(N), just an example...
    i++;                      // O(1)
}
```

- The body of the while loop: O(N)
- Loop is executed: N times

$N \times O(N) = O(N^2)$

Examples (cont.'d)

```
if (i<j)
   for ( i=0; i<N; i++ )
        X = X+i;
else
   X=0;
```

O(N)

O(1)

Max (O(N), O(1)) = O (N)

Asymptotic Notation

- O notation: asymptotic "less than":
  - f(n)=O(g(n)) implies: f(n) "≤" g(n)
- Ω notation: asymptotic "greater than":
  - f(n)= Ω (g(n)) implies: f(n) "≥" g(n)
- Θ notation: asymptotic "equality":
  - f(n)= Θ (g(n)) implies: f(n) "=" g(n)

Definition: *O(g), at most order g*

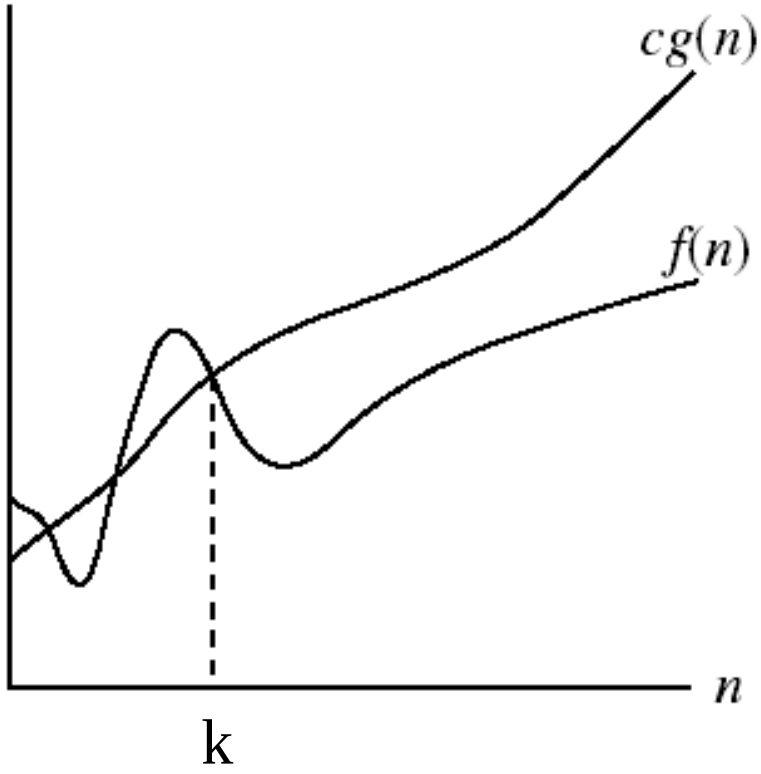
Let *f*,*g* are functions $\mathbf{R} \rightarrow \mathbf{R}$.

- We say that "*f* is *at most order g*", if:

$$\exists c,k: f(x) \leq cg(x), \forall x>k$$

  - "Beyond some point *k*, function *f* is at most a constant *c* times *g* (*i.e.*, proportional to *g*)."
- "*f* is *at most order g*", or "*f* is O(*g*)", or "*f*=O(*g*)" all just mean that $f \in O(g)$.
- Sometimes the phrase "at most" is omitted.

Big-O Visualization



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

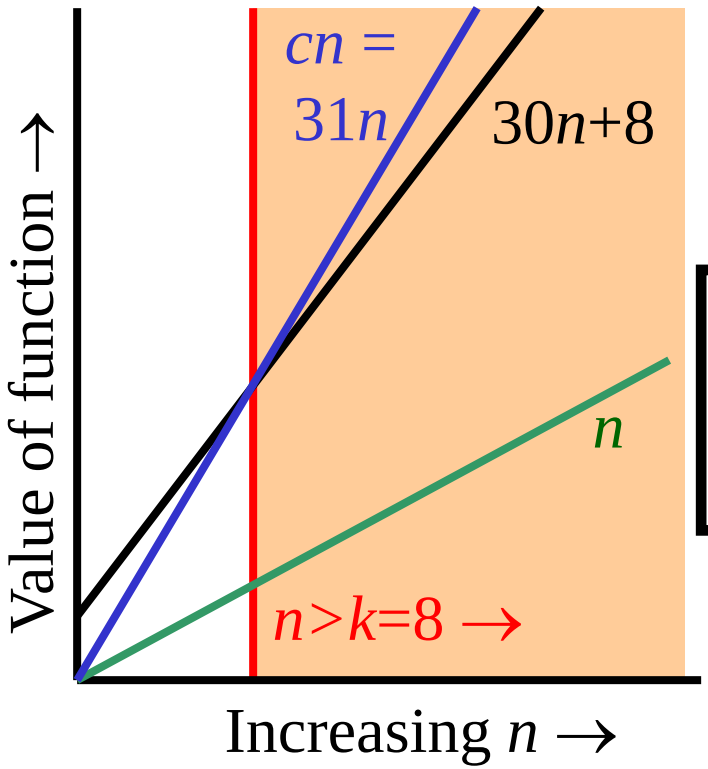
"Big-O" Proof Examples

- Show that $30n+8$ is $O(n)$.
  - Show $\exists c,k$: $30n+8 \leq cn$, $\forall n>k$.
    - Let $c=31$, $k=8$. Assume $n>k=8$. Then $cn = 31n = 30n + n > 30n+8$, so $30n+8 < cn$.
- Show that n^2+1 is $O(n^2)$.
  - Show $\exists c,k$: $n^2+1 \leq cn^2$, $\forall n>k$: .
    - Let $c=2$, $k=1$. Assume $n>1$. Then

    $cn^2 = 2n^2 = n^2+n^2 > n^2+1$, or $n^2+1< cn^2$.

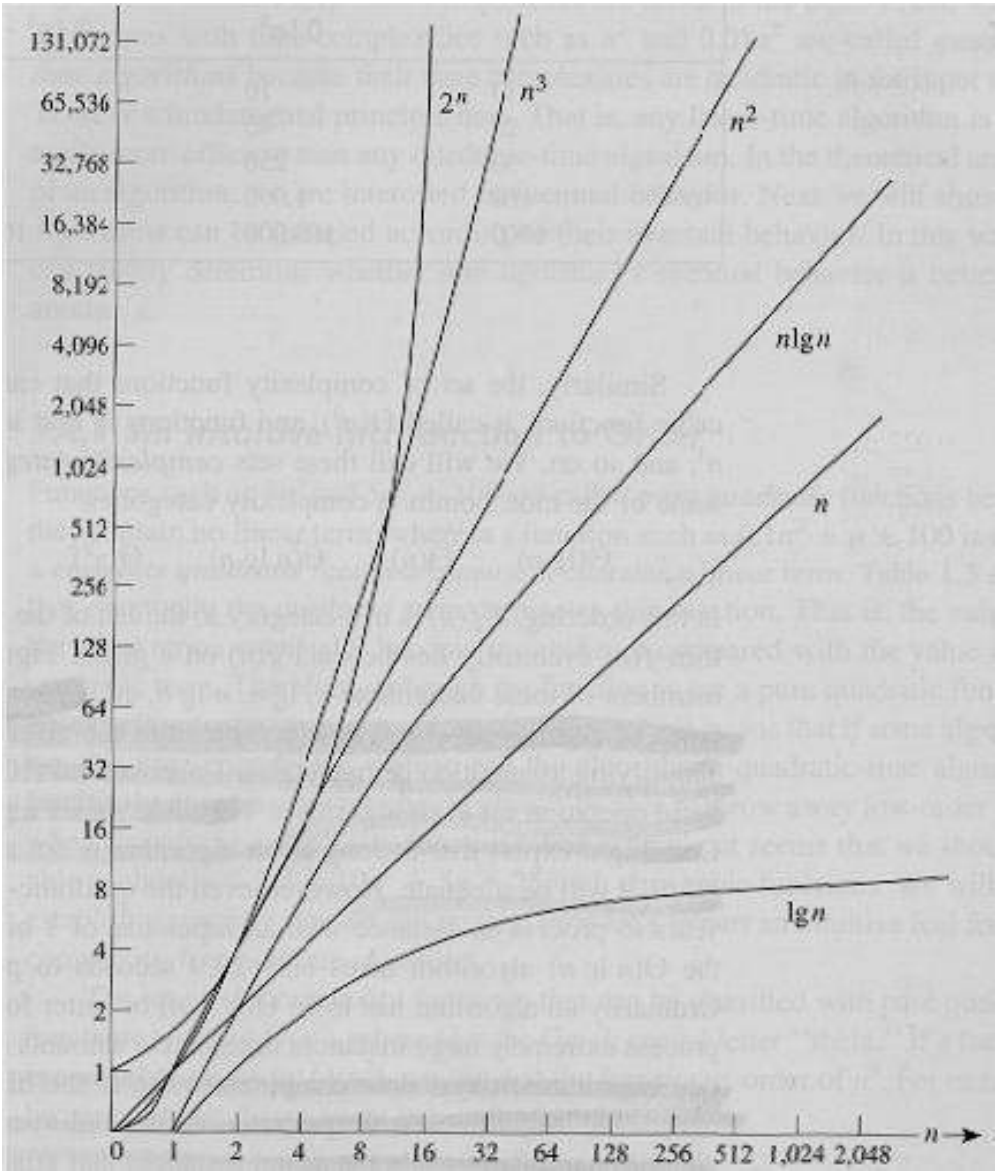
Big-O example, graphically

- Note 30*n*+8 isn't less than *n* *anywhere* (*n*>0).
- It isn't even less than 31*n* *everywhere*.
- But it *is* less than 31*n* everywhere to the right of *n*=8.



$30n+8 \in O(n)$

Common orders of magnitude



Recurrence Relations

- Can easily describe the runtime of recursive algorithms
- Can then be expressed in a closed form (not defined in terms of itself)

- Consider the linear search:

Linear Search

- Recursively
- Look at an element (constant work, c), then search the remaining elements…

- T(n) = T(n-1) + c
- “The cost of searching n elements is the cost of looking at 1 element, plus the cost of searching n-1 elements”

Linear Search (cont.)

- We'll "unwind" a few of these

$T(n) = T(n-1) + c$ (1)

But, $T(n-1) = T(n-2) + c$, from above

Substituting back in:

$T(n) = T(n-2) + c + c$

Gathering like terms

$T(n) = T(n-2) + 2c$ (2)

list of intermediates

Result at i^{th} unwinding	i
$T(n) = T(n-1) + 1c$	1
$T(n) = T(n-2) + 2c$	2
$T(n) = T(n-3) + 3c$	3
$T(n) = T(n-4) + 4c$	4

Linear Search (cont.)

- An expression for the kth unwinding:

  $T(n) = T(n-k) + kc$

- We have 2 variables, k and n, but we have a relation
- T(d) is constant (can be determined) for some constant d (we know the algorithm)
- Choose any convenient # to stop.

Linear Search (cont.)

- Let's decide to stop at T(0). When the list to search is empty, you're done…
- 0 is convenient, in this example…

  Let n-k = 0 => n=k
- Now, substitute n in everywhere for k:

  $T(n) = T(n-n) + nc$

  $T(n) = T(0) + nc = nc + c_0 = O(n)$

  (T(0) is some constant, c_0)

Binary Search

- Algorithm – "check middle, then search lower ½ or upper ½"
- $T(n) = T(n/2) + c$

  where c is some constant, the cost of checking the middle…
- Can we really find the middle in constant time? (Make sure.)

Binary Search (cont)

Let's do some quick substitutions:

$T(n) = T(n/2) + c$ (1)

but $T(n/2) = T(n/4) + c$, so

$T(n) = T(n/4) + c + c$

$T(n) = T(n/4) + 2c$ (2)

$T(n/4) = T(n/8) + c$

$T(n) = T(n/8) + c + 2c$

$T(n) = T(n/8) + 3c$ (3)

Binary Search (cont)

Result at i^{th} unwinding			i
T(n)	= T(n/2) + c	$=T(n/2^1) + 1c$	1
T(n)	= T(n/4) + 2c	$=T(n/2^2) + 2c$	2
T(n)	= T(n/8) + 3c	$=T(n/2^3) + 3c$	3
T(n)	= T(n/16) + 4c	$=T(n/2^4) + 4c$	4

Binary Search (cont)

- After k unwindings:

  $T(n) = T(n/2^k) + kc$

- Need a convenient place to stop unwinding – need to relate k & n
- Let's pick $T(0) = c_0$ So,

  $n/2^k = 0$ =>

  $n=0$

  Hmm. Easy, but not real useful…

Binary Search (cont)

- Okay, let's consider $T(1) = c_0$
- So, let:

$$n/2^k = 1 \quad =>$$

$$n = 2^k \quad =>$$

$$k = \log_2 n = \lg n$$

Binary Search (cont.)

- Substituting back in (getting rid of k):

$$T(n) = T(1) + c \lg(n)$$

$$= c \lg(n) + c_0$$

$$= O(\lg(n))$$

The Master Theorem

- if $T(n) = aT(n/b) + f(n)$ then

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \log n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{ AND} \\ & af(n/b) < cf(n) \text{ for large } n \end{cases} \quad \begin{array}{l} \varepsilon > 0 \\ c < 1 \end{array}$$

Master's method

- "Cookbook" for solving recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where, $a \geq 1$, $b > 1$, and $f(n) > 0$

Case 1: if $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then: $T(n) = \Theta(n^{\log_b a})$

Case 2: if $f(n) = \Theta(n^{\log_b a})$, then: $T(n) = \Theta(n^{\log_b a} \lg n)$

Case 3: if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, and if

$af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n, then:

$T(n) = \Theta(f(n))$

regularity condition

Examples

$$T(n) = 2T(n/2) + n$$

$a = 2, b = 2, \log_2 2 = 1$

Compare $n^{\log_2 2}$ with $f(n) = n$

$\Rightarrow f(n) = \Theta(n) \Rightarrow$ Case 2

$\Rightarrow T(n) = \Theta(n \lg n)$

Examples

$$T(n) = 2T(n/2) + n^2$$

$a = 2, b = 2, \log_2 2 = 1$

Compare n with $f(n) = n^2$

$\Rightarrow f(n) = \Omega(n^{1+\varepsilon})$ Case 3 $\Rightarrow$ verify regularity cond.

$a\, f(n/b) \leq c\, f(n)$

$\Leftrightarrow 2\, n^2/4 \leq c\, n^2 \Rightarrow c = \frac{1}{2}$ is a solution $(c<1)$

$\Rightarrow T(n) = \Theta(n^2)$

Examples (cont.)

$$T(n) = 2T(n/2) + \sqrt{n}$$

$a = 2, b = 2, \log_2 2 = 1$

Compare n with $f(n) = n^{1/2}$

$\Rightarrow f(n) = O(n^{1-\varepsilon})$ Case 1

$\Rightarrow T(n) = \Theta(n)$

Examples

$$T(n) = 3T(n/4) + n\lg n$$

$a = 3$, $b = 4$, $\log_4 3 = 0.793$

Compare $n^{0.793}$ with $f(n) = n\lg n$

$f(n) = \Omega(n^{\log_4 3+\varepsilon})$ Case 3

Check regularity condition:

$$3*(n/4)\lg(n/4) \le (3/4)n\lg n = c*f(n),$$

$c=3/4$

Examples

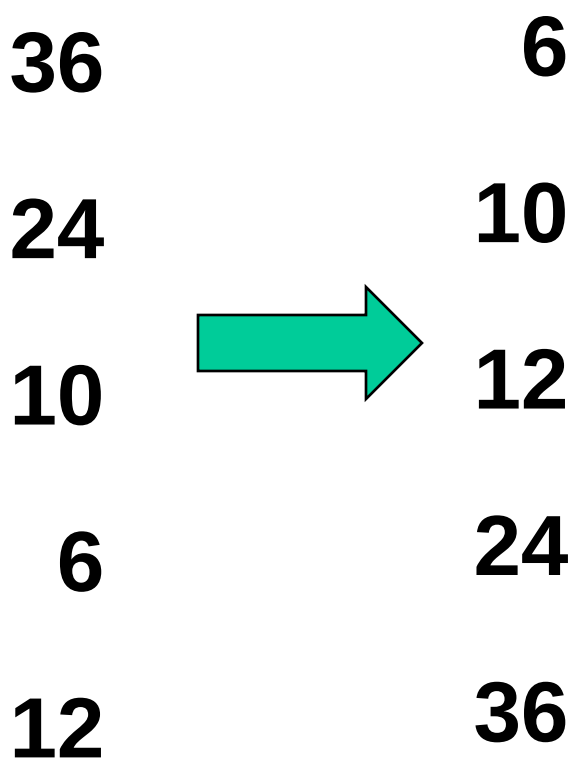
$$T(n) = 2T(n/2) + n\lg n$$

$a = 2, b = 2, \log_2 2 = 1$

- Compare n with $f(n) = n\lg n$
  - seems like case 3 should apply
- f(n) must be polynomially larger by a factor of n^{ε}
- In this case it is only larger by a factor of $\lg n$

Sorting means . . .

- Sorting rearranges the elements into either ascending or descending order within the array (we'll use ascending order).



The Sorting Problem

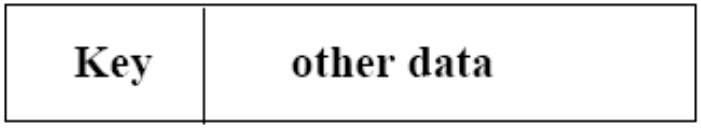
- **Input:**
  - A sequence of n numbers $a_1, a_2, \ldots, a_n$
- **Output:**
  - A permutation (reordering) $a_1', a_2', \ldots, a_n'$ of the input sequence such that $a_1' \leq a_2' \leq \cdots \leq a_n'$

Structure of data

- Usually, the numbers to be sorted are part of a collection of data called a record

- Each record contains a key, which is the value to be sorted key, which is the value to be sorted

example of a record



- Note that when the keys must be rearranged, the data associated with the keys must also be rearranged (time consuming !!)

- Pointers can be used instead (space consuming !!)

Why Study Sorting Algorithms?

- Two important applications of sorting:
  - An aid to search
  - Matching entries in lists
- Various algorithms are better suited to some of these situations

Some Definitions

- Internal Sort
  - The data to be sorted is all stored in the computer's main memory.
- External Sort
  - Some of the data to be sorted might be stored in some external, slower, device.
- In Place Sort
  - The amount of extra space required to sort the data is constant with the input size.

Stability

- A STABLE sort preserves relative order of records with equal keys

Sorted on first key:

Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

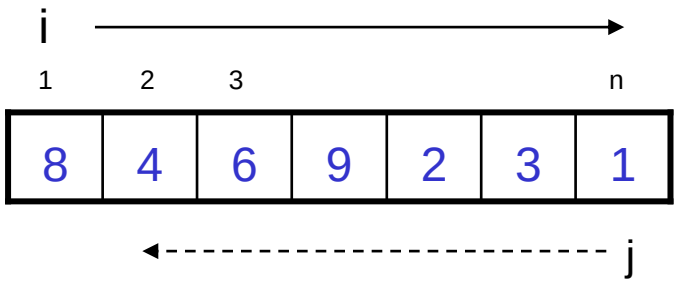
Sort file on second key:

Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Furia	3	A	766-093-9873	22 Brown
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Gazsi	4	B	665-303-0266	113 Walker
Aaron	4	A	664-480-0023	097 Little

Records with key value 3 are not in order on first key!!

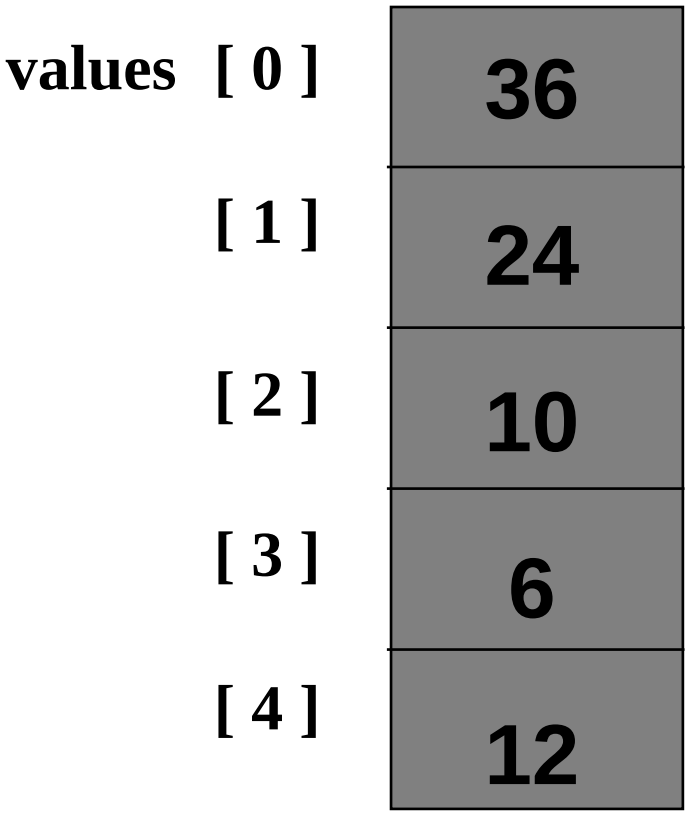
Bubble Sort

- Idea:
  - Repeatedly pass through the array
  - Swaps adjacent elements that are out of order



- Easier to implement, but slower

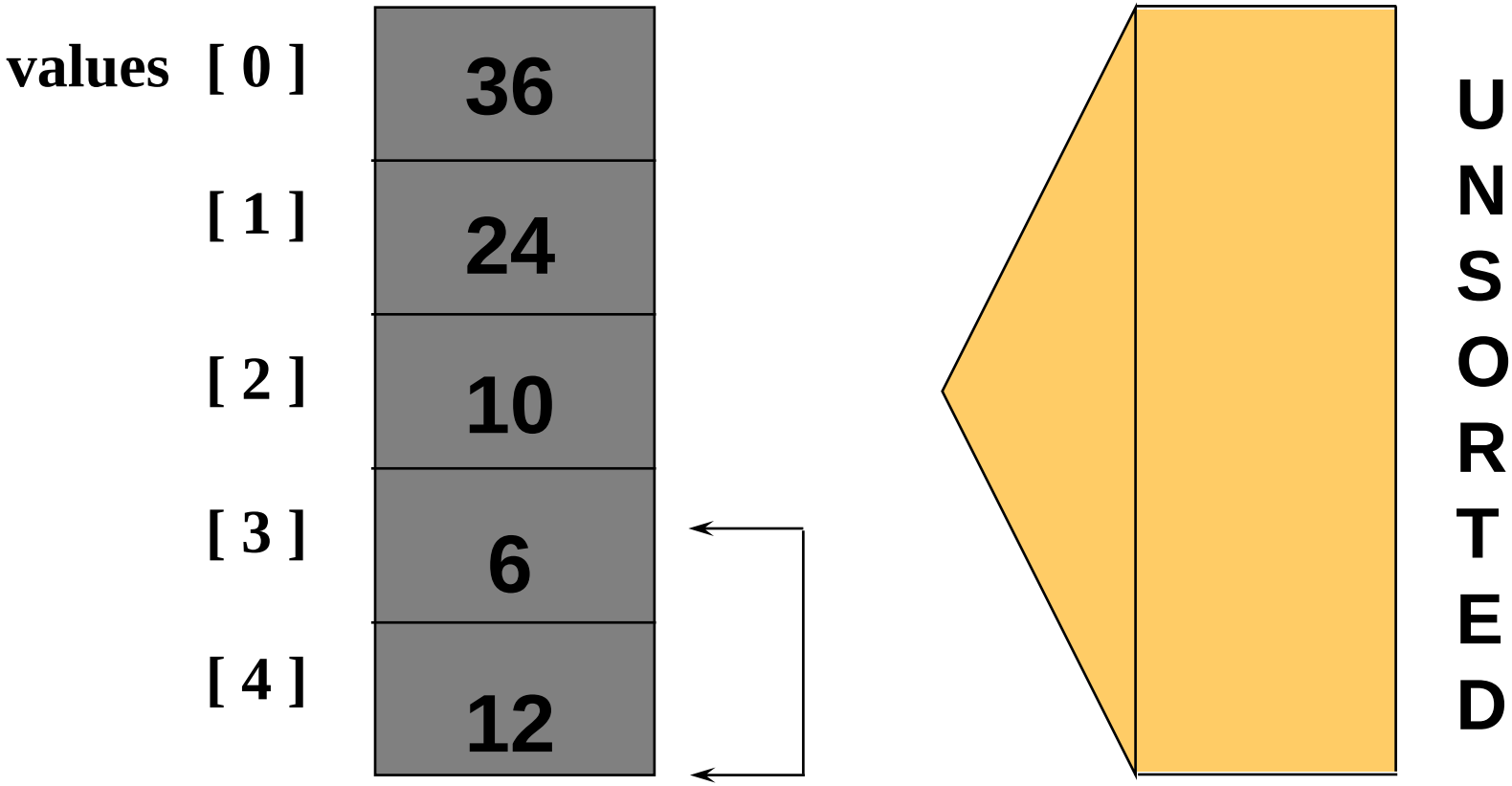
Bubble Sort



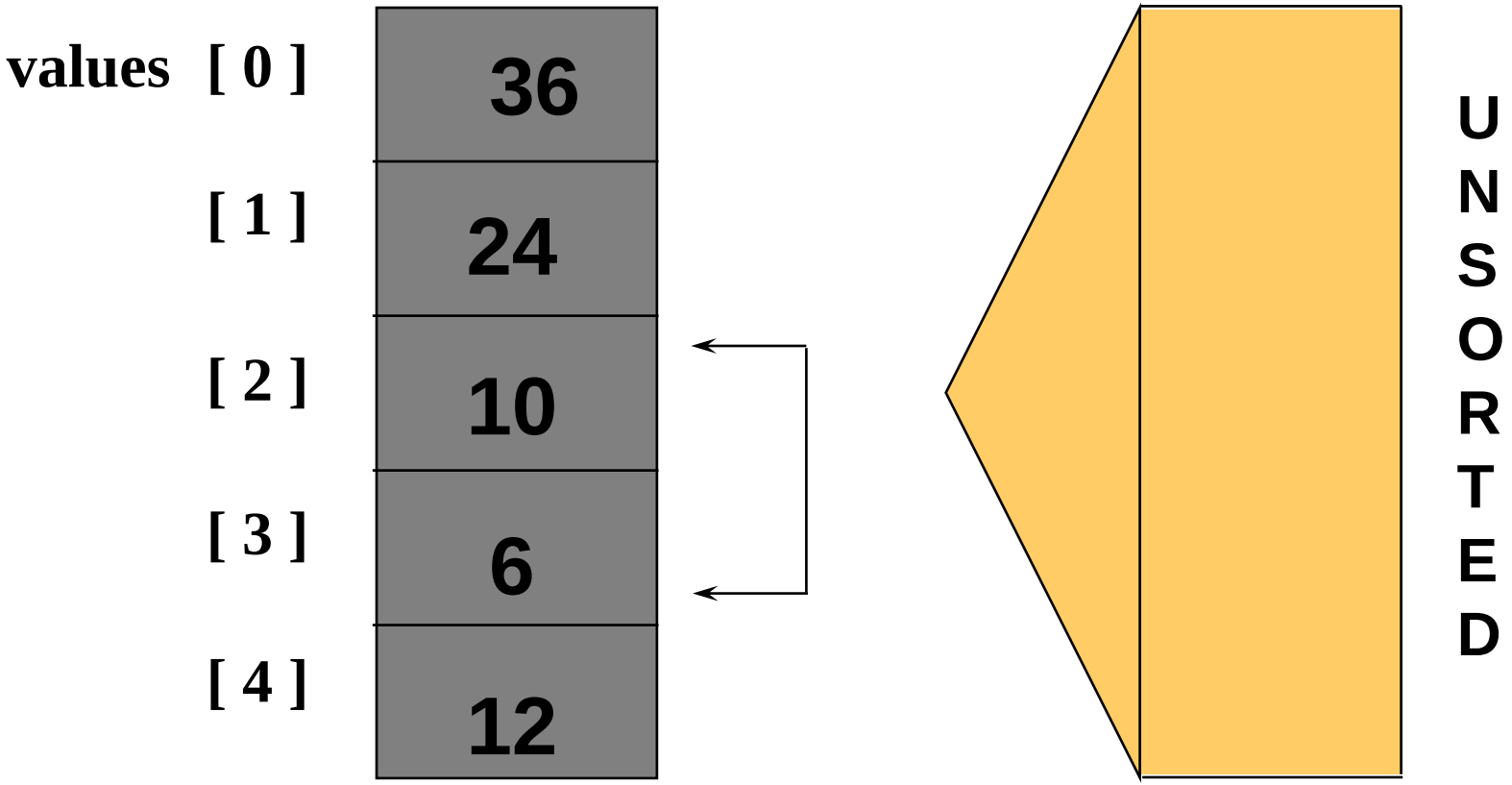
Compares neighboring pairs of array elements, starting with the last array element, and swaps neighbors whenever they are not in correct order.

On each pass, this causes the smallest element to "bubble up" to its correct place in the array.

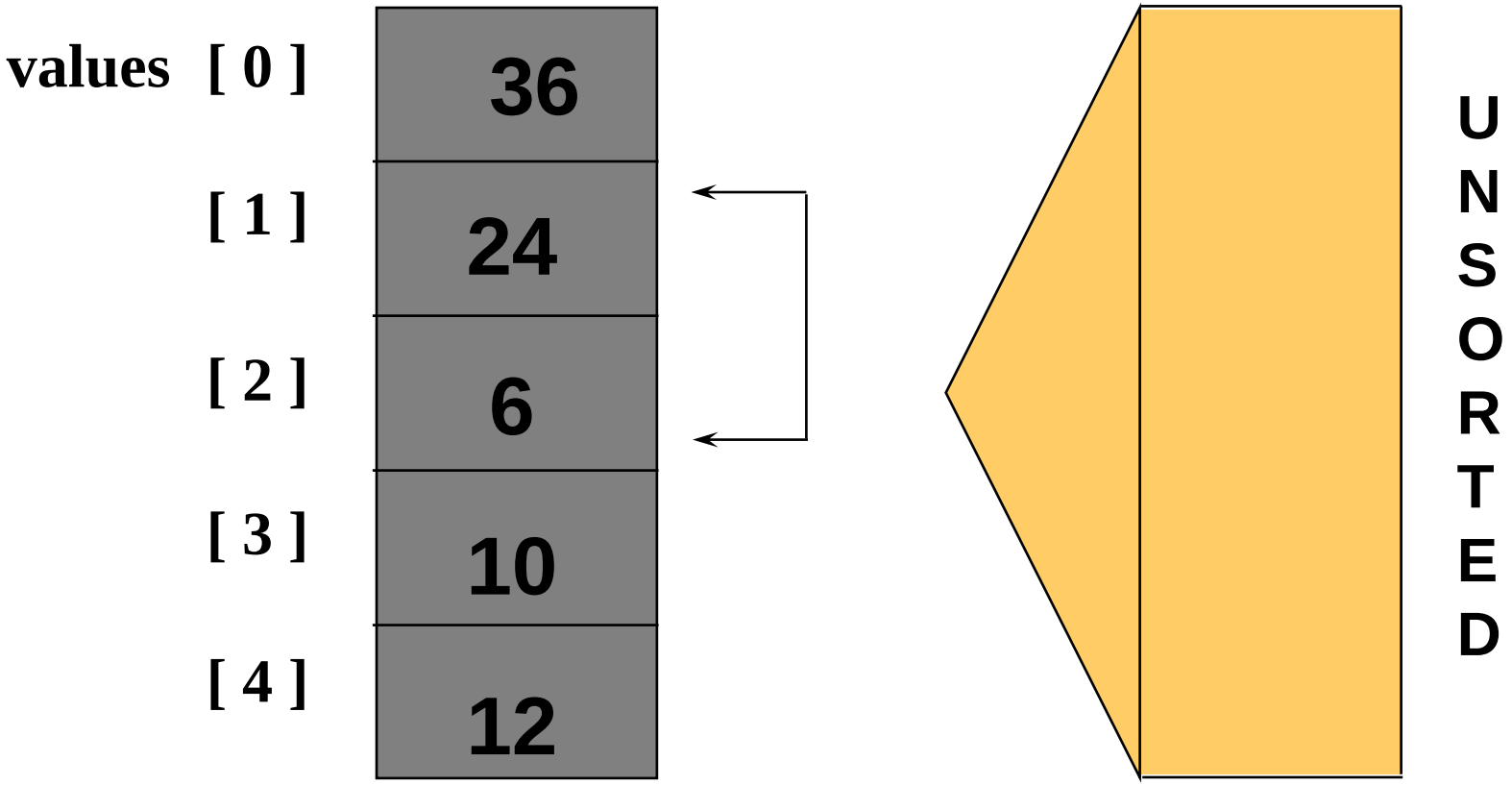
Bubble Sort: Pass One



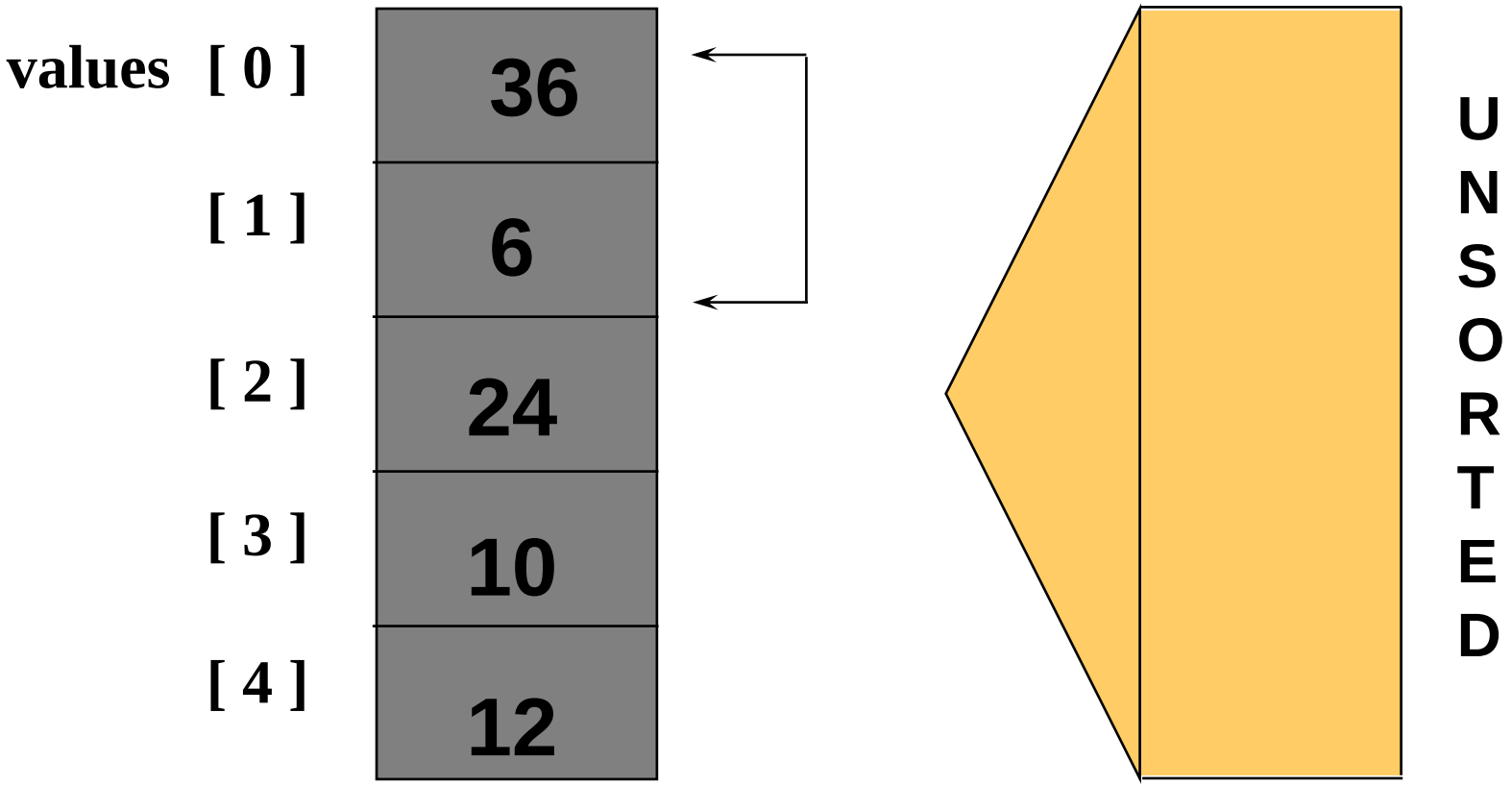
Bubble Sort: Pass One



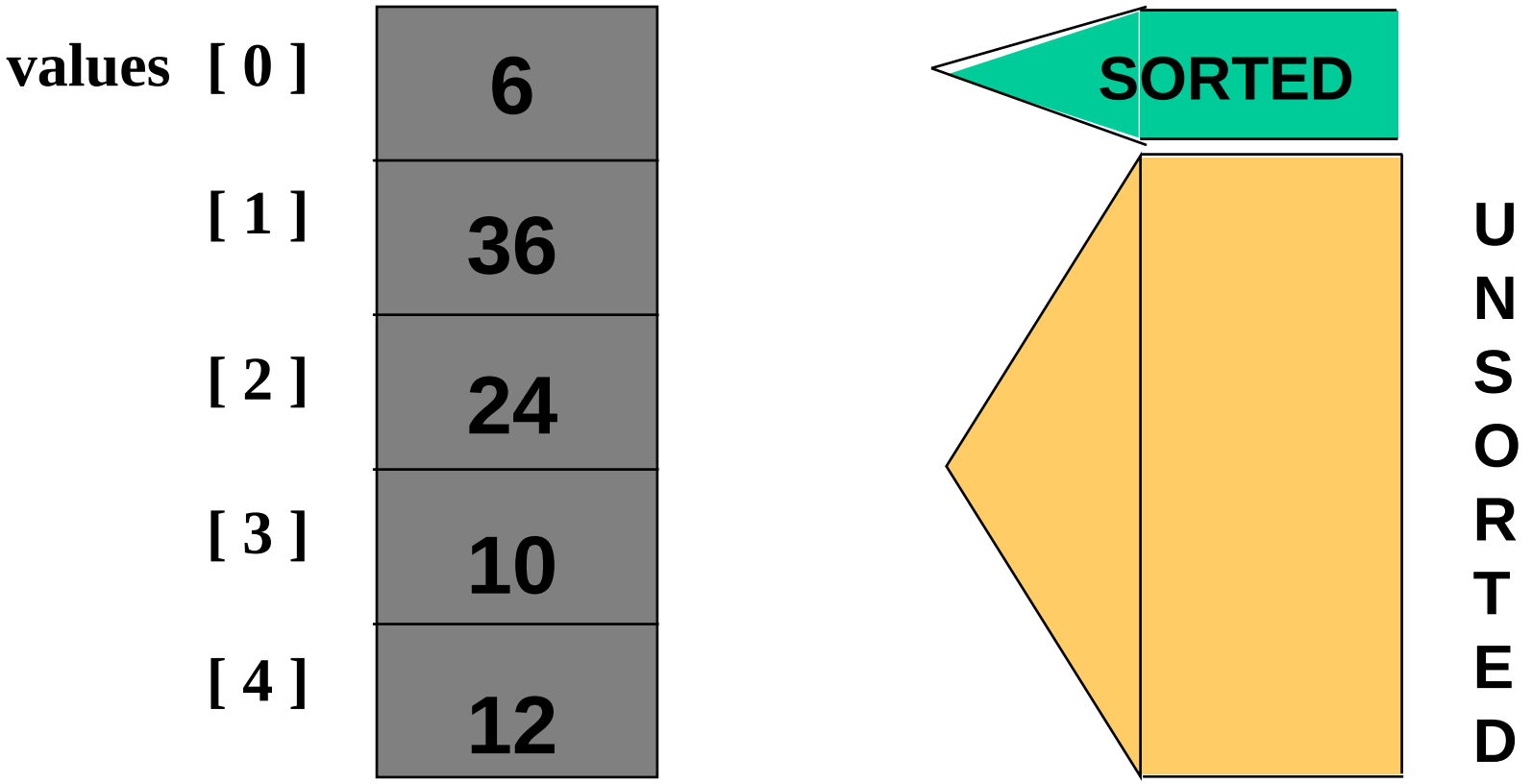
Bubble Sort: Pass One



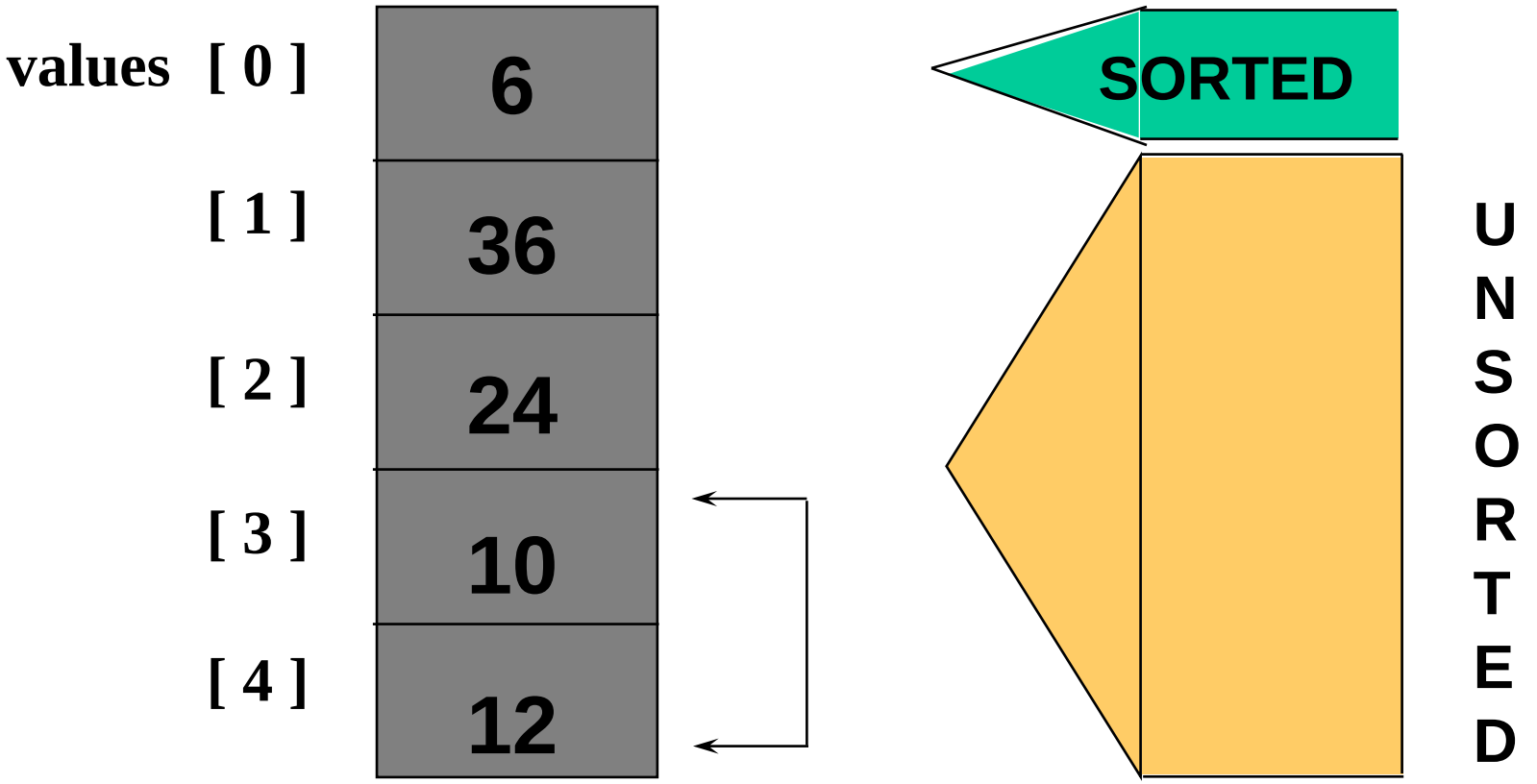
Bubble Sort: Pass One



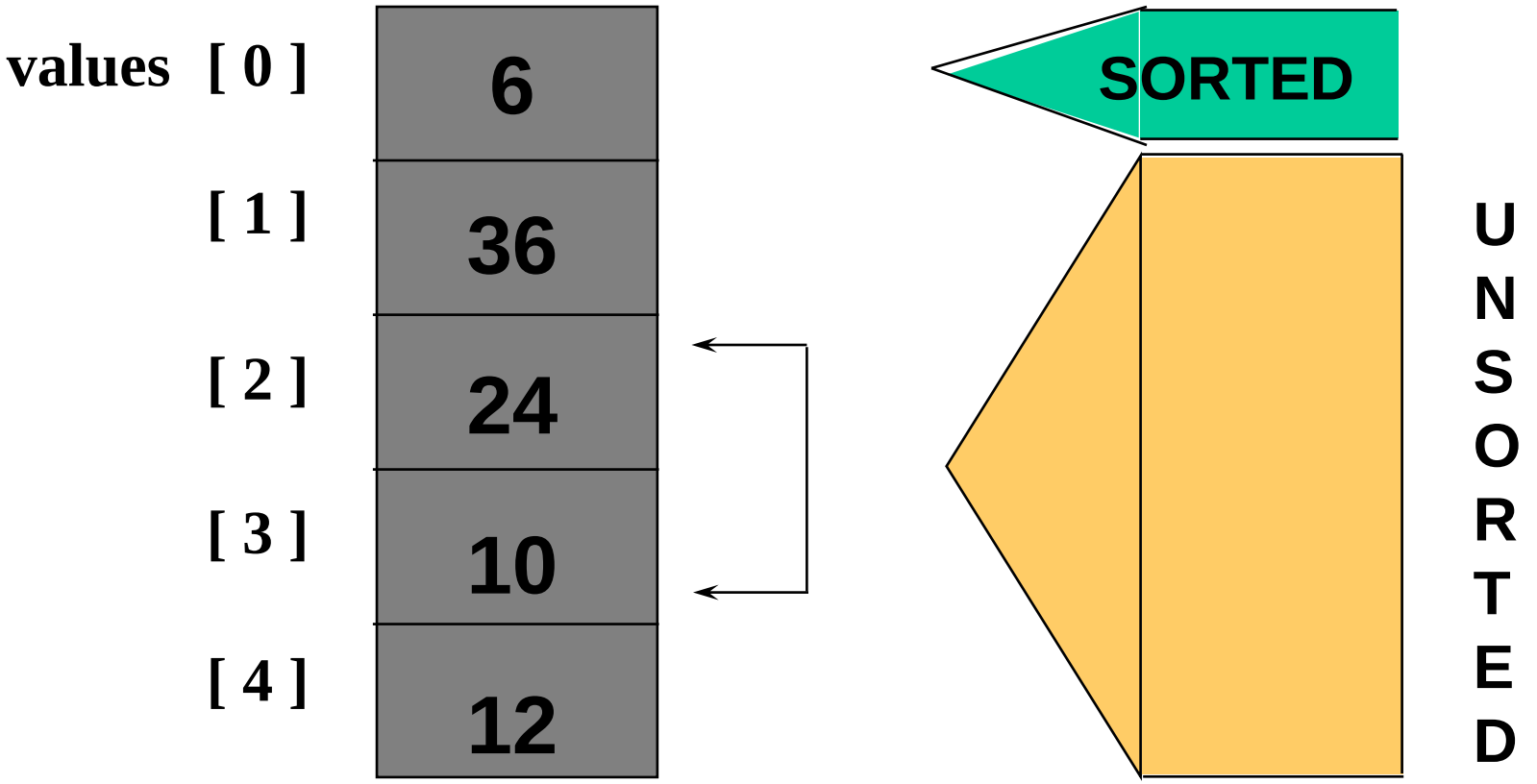
Bubble Sort: End Pass One



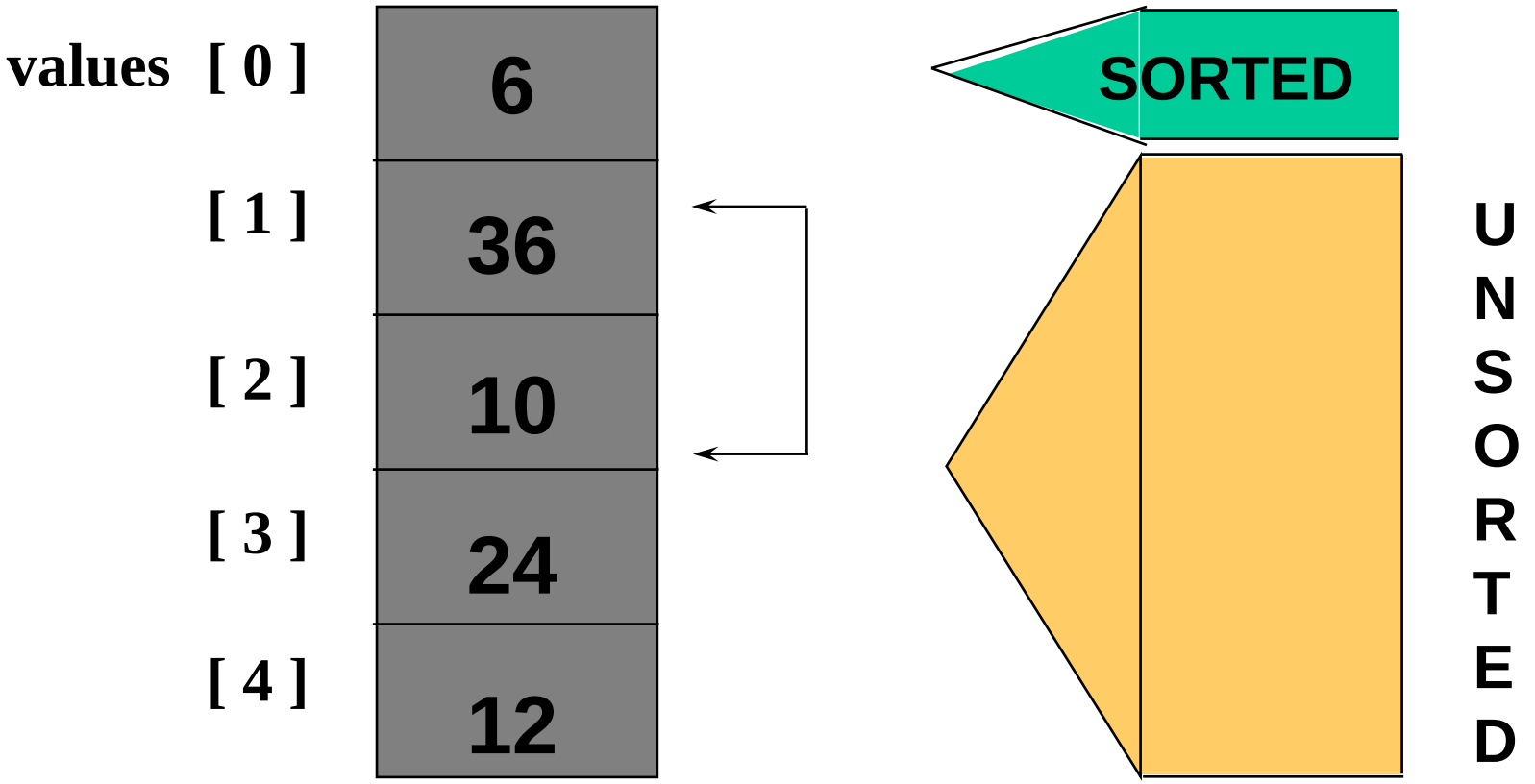
Bubble Sort: Pass Two



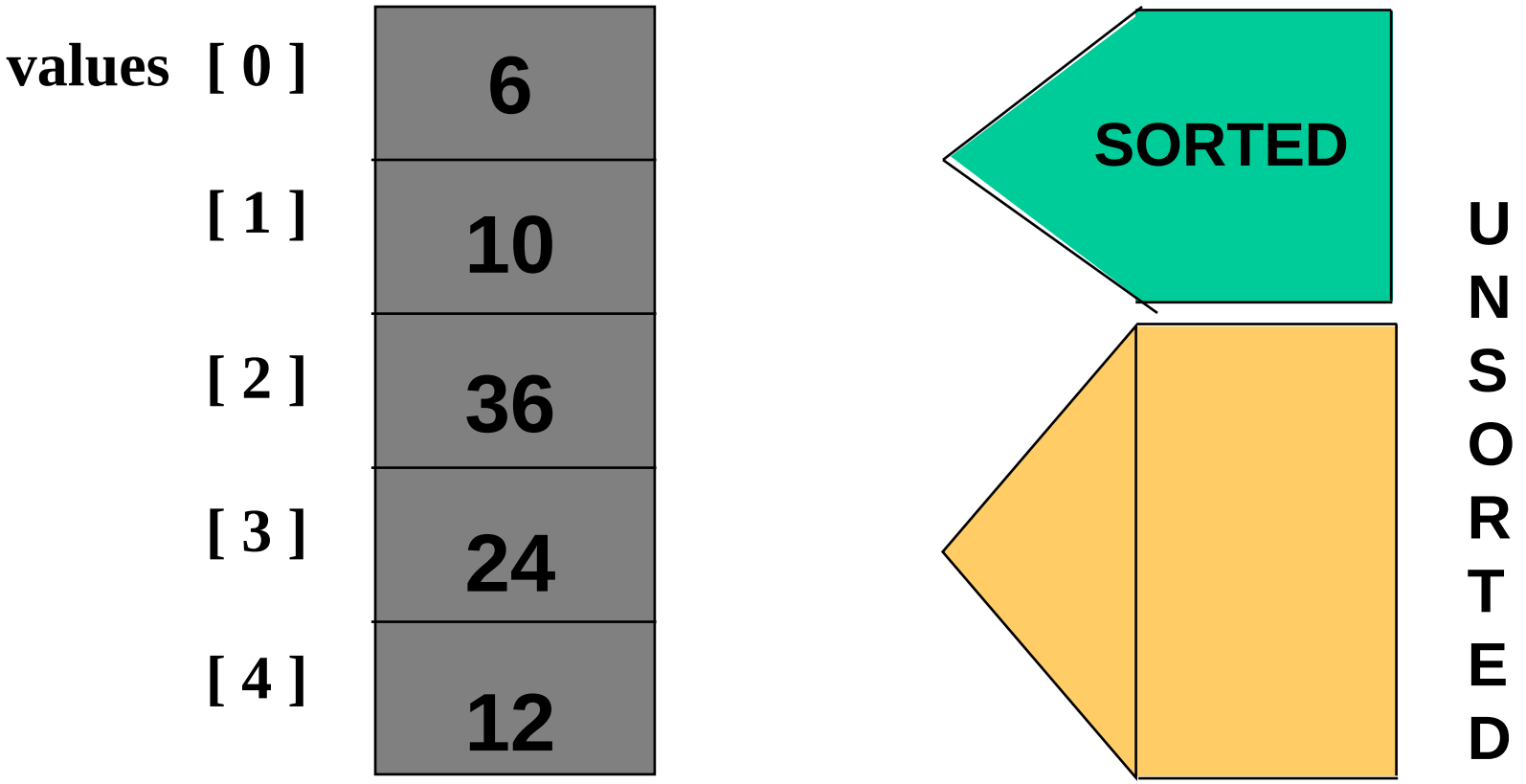
Bubble Sort: Pass Two



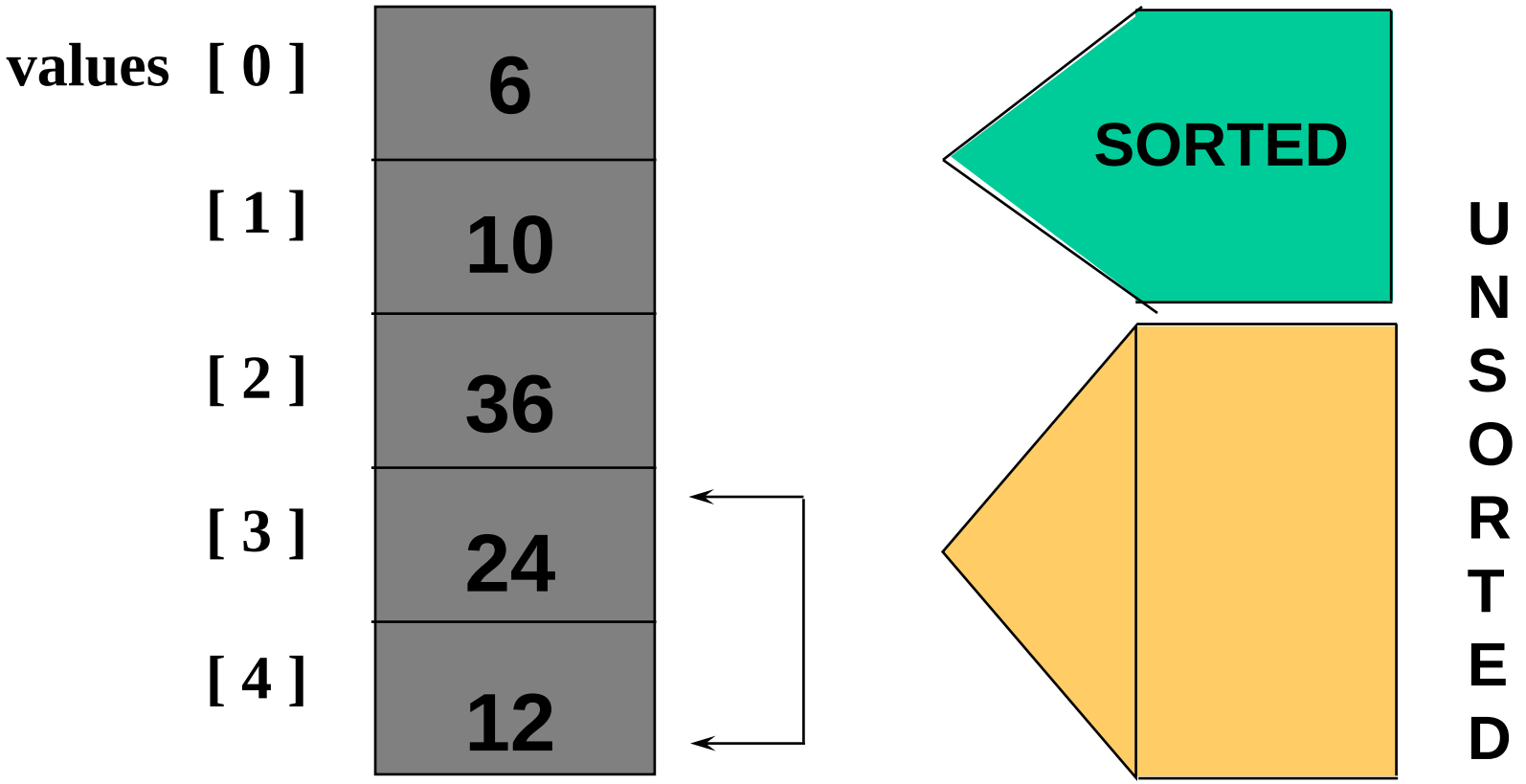
Bubble Sort: Pass Two



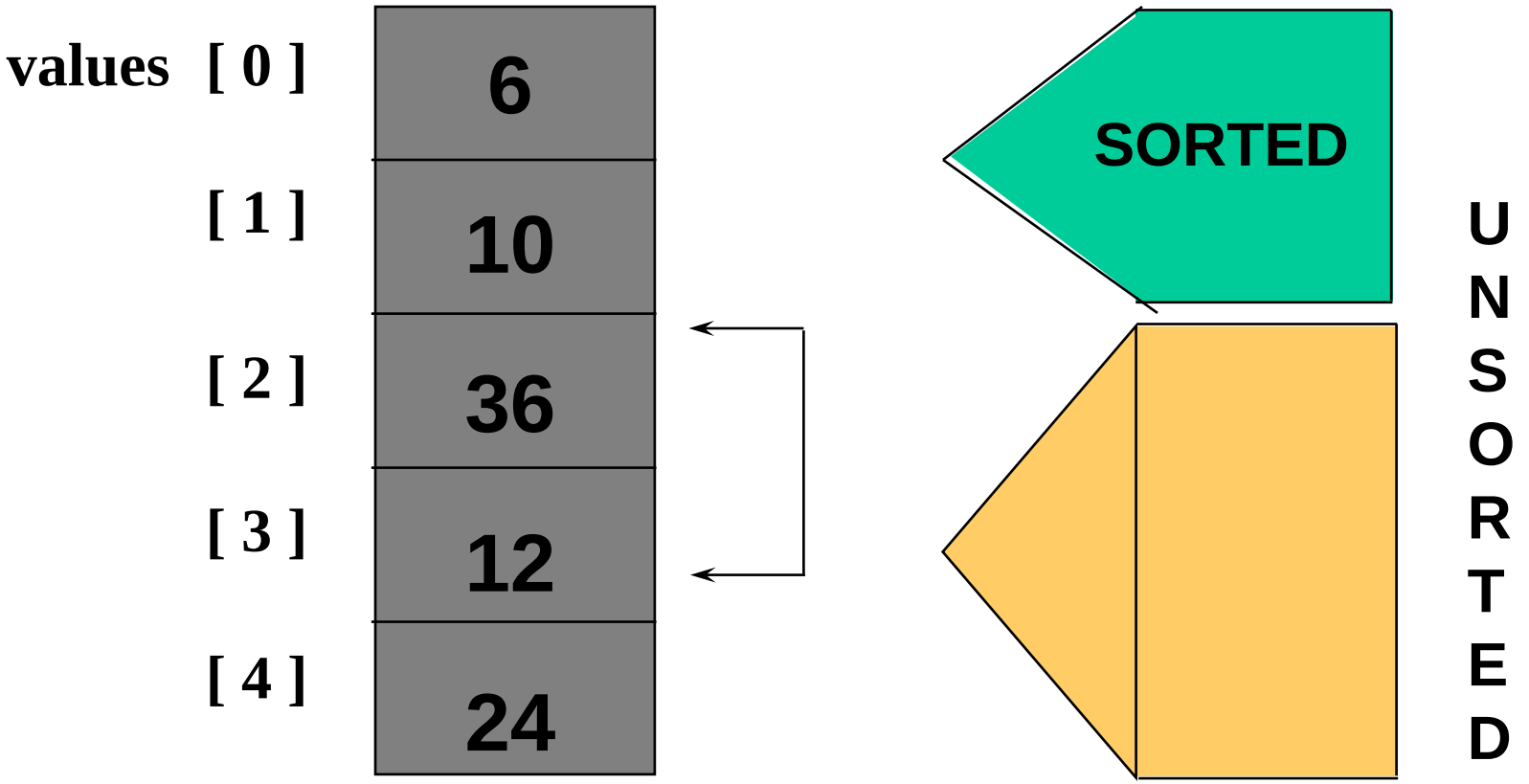
Bubble Sort: End Pass Two



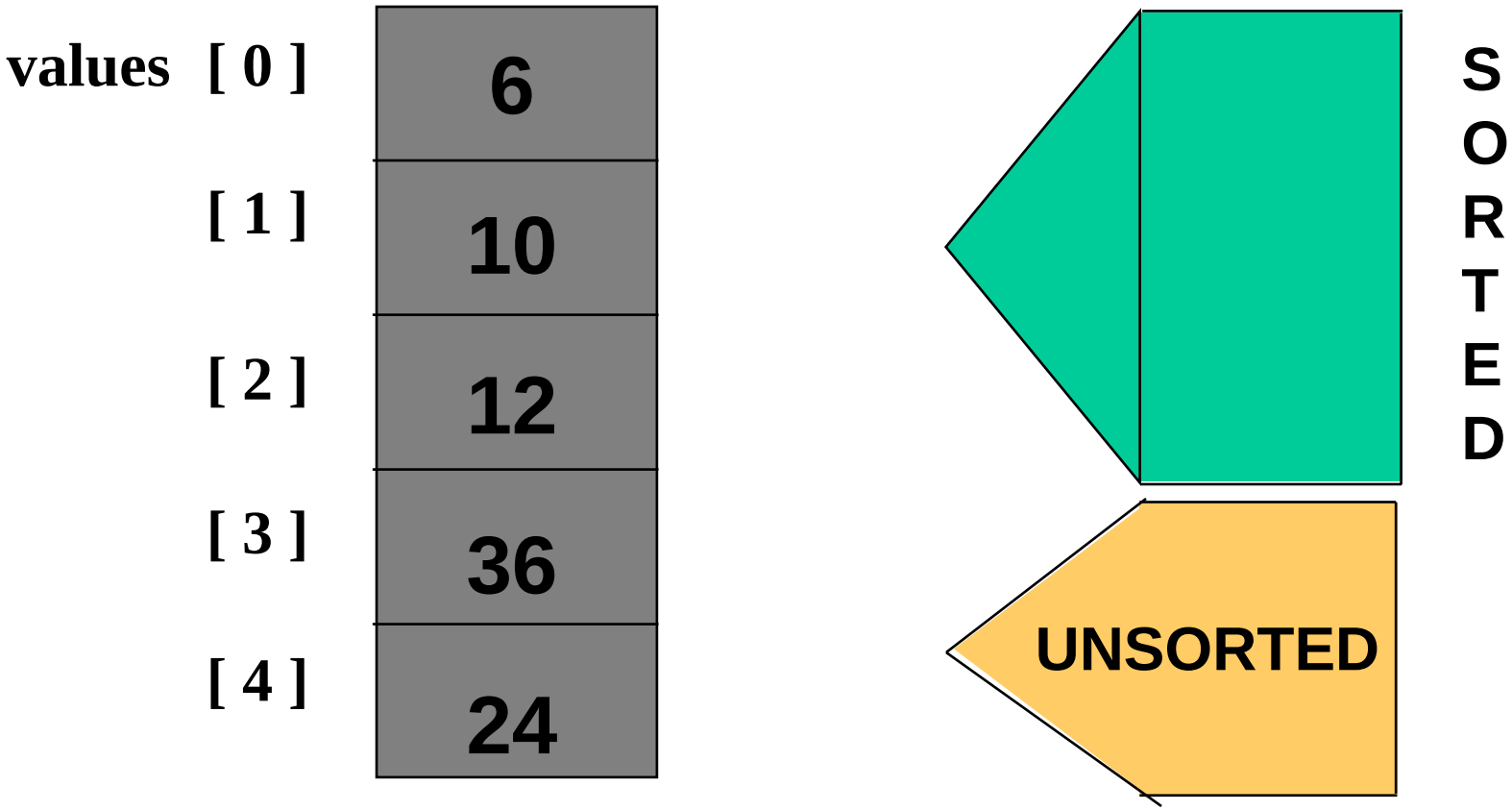
Bubble Sort: Pass Three



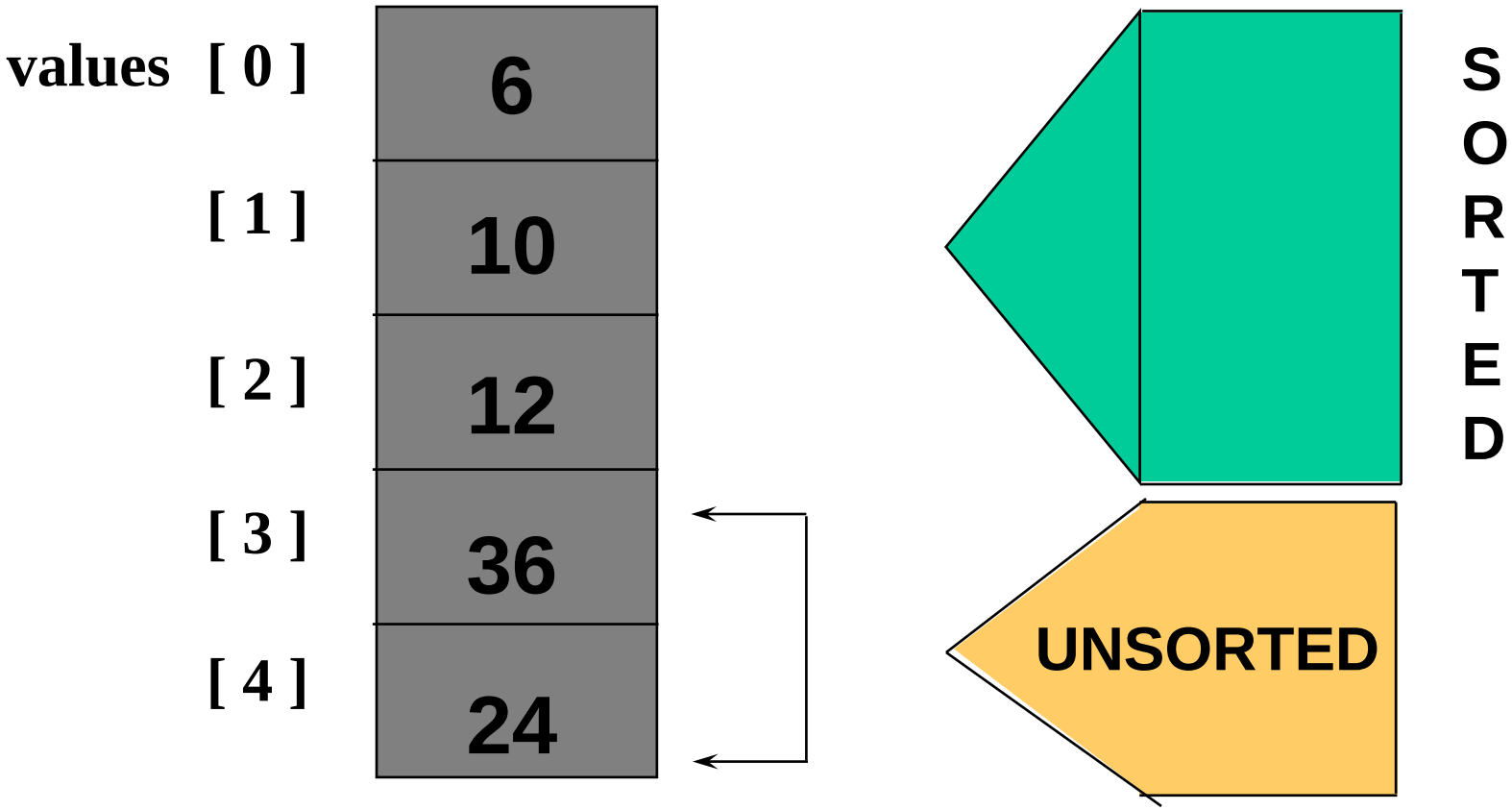
Bubble Sort: Pass Three



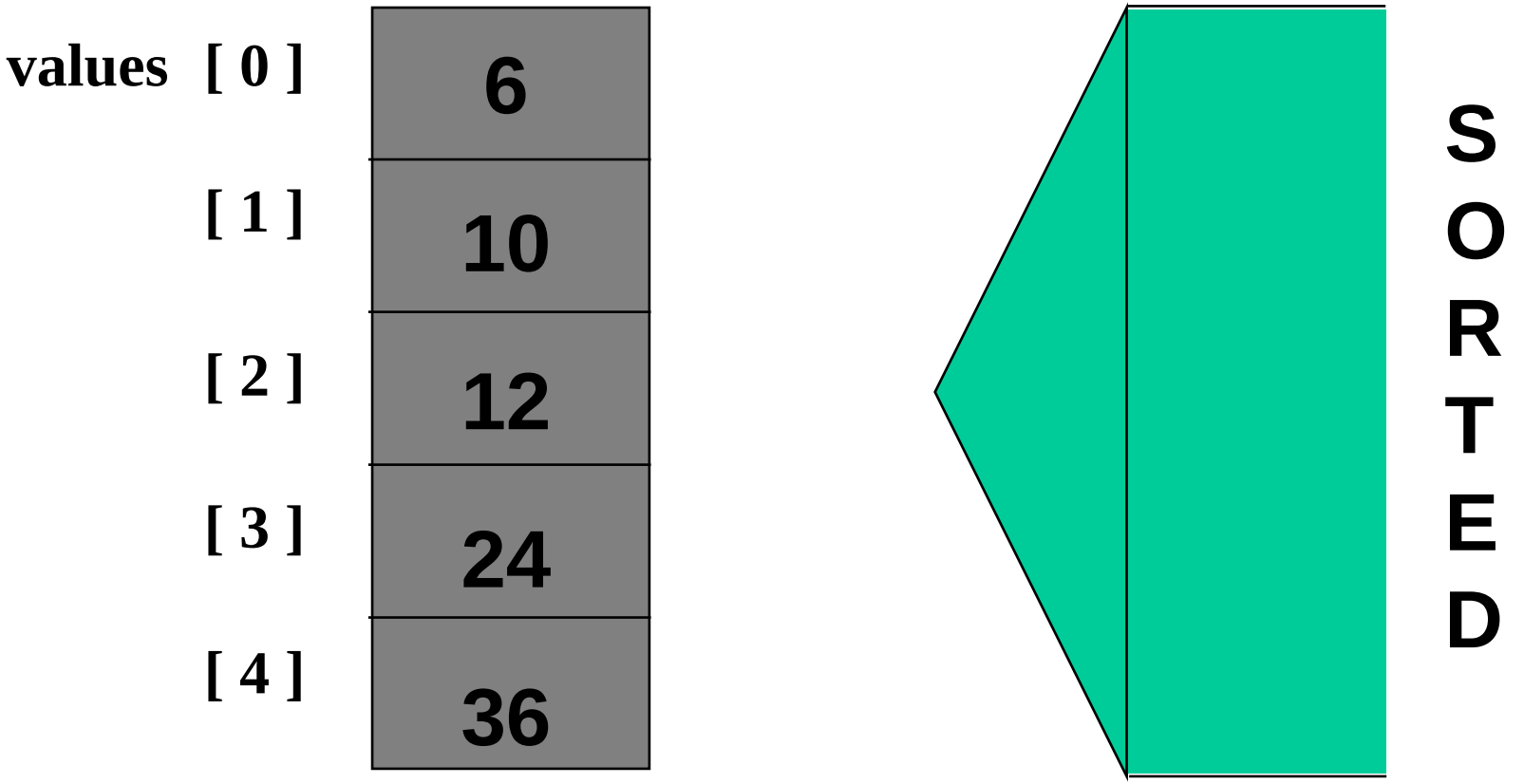
Bubble Sort: End Pass Three



Bubble Sort: Pass Four



Bubble Sort: End Pass Four



Bubble Sort

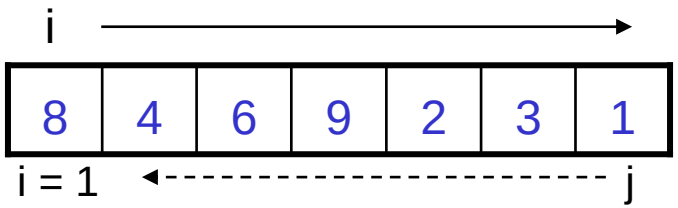
Alg.: BUBBLESORT(A)

for i ← 1 **to** length[A]

do for j ← length[A] **downto** i + 1

do if A[j] < A[j -1]

then exchange A[j] ↔ A[j-1]



Bubble-Sort Running Time

Alg.: BUBBLESORT(A)

for $i \leftarrow 1$ **to** length[A] c_1

do for $j \leftarrow$ length[A] **downto** $i + 1$ c_2

Comparisons: $\approx n^2/2$ **do if** $A[j] < A[j-1]$ c_3

Exchanges: $\approx n^2/2$ **then** exchange $A[j] \leftrightarrow A[j-1]$ c_4

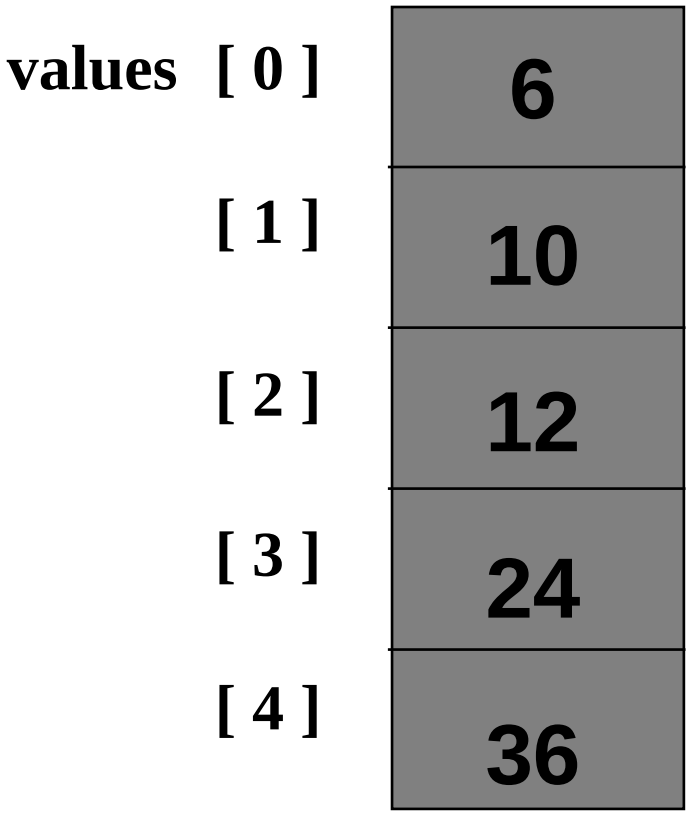
$$T(n) = c_1(n+1) + c_2 \sum_{i=1}^{n}(n-i+1) + c_3 \sum_{i=1}^{n}(n-i) + c_4 \sum_{i=1}^{n}(n-i)$$

$$= O(n) + (c_2 + c_2 + c_4) \sum_{i=1}^{n}(n-i)$$

where $$\sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n} n - \sum_{i=1}^{n} i = n^2 - \frac{n(n+1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Thus, $T(n) = O(n^2)$

Improved Bubble Sort



Add a “flag” to exit iterations if nothing changes in a single iteration

Example [Improved Bubble Sort]

Can we improve the RT of bubble sort in some special cases?!

i^{th} pass

2	5	8	9	11	17

If in a pass through the array there is no swaps => array is sorted!

Bubble Sort Algorithm with Slightly Improved Running Time – do the $(i+1)^{st}$ pass if there was at least one swap in the i^{th} pass

```
boolean newSwaps = true;
while (newSwaps) {
        newSwaps = false;
        for (int i=0; i<a.length-1; i++) {
                if (a[i] > a[i+1]) {
                        temp = a[i];
                        a[i] = a[i+1];
                        a[i+1] = temp;
                        newSwaps = true;
                };
        };
}
```

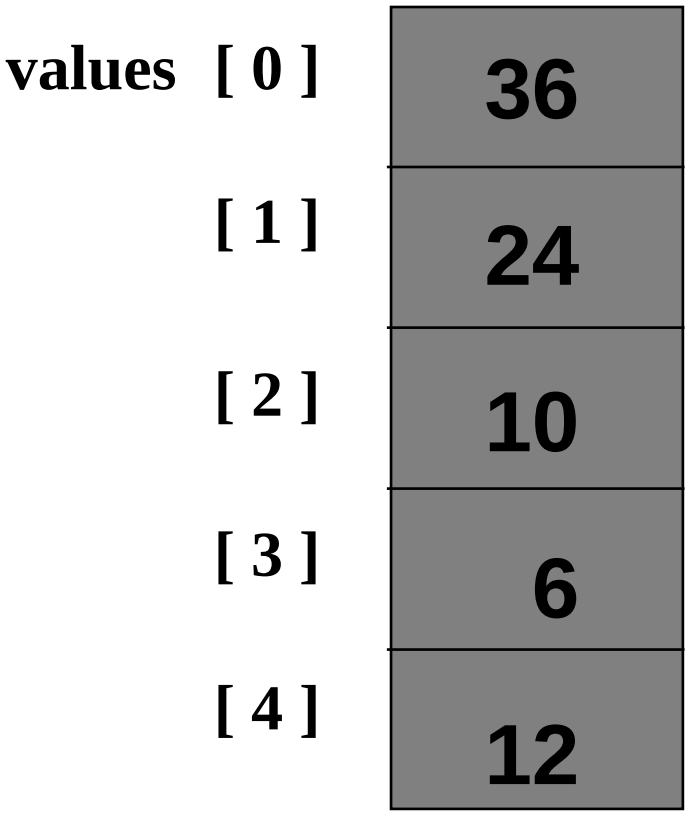
Worst case: elements in decreasing order ⇒ Bubble Sort RT = $O(n^2)$.

Best case: sequence almost sorted ⇒ Bubble Sort RT = $O(n)$

Selection Sort

- Idea:
  - Find the smallest element in the array
  - Exchange it with the element in the first position
  - Find the second smallest element and exchange it with the element in the second position
  - Continue until the array is sorted
- Disadvantage:
  - Running time depends only slightly on the amount of order in the file

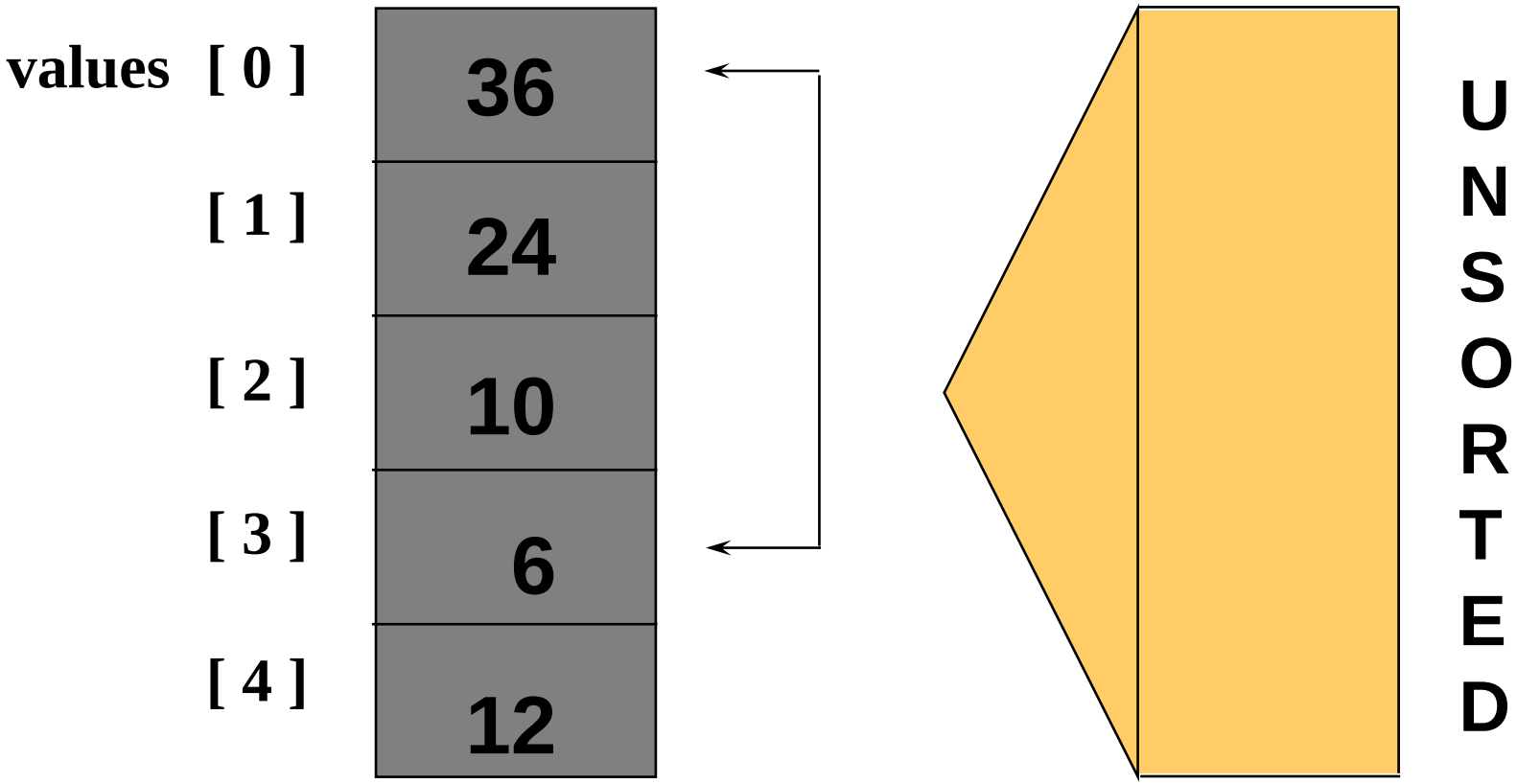
Selection Sort



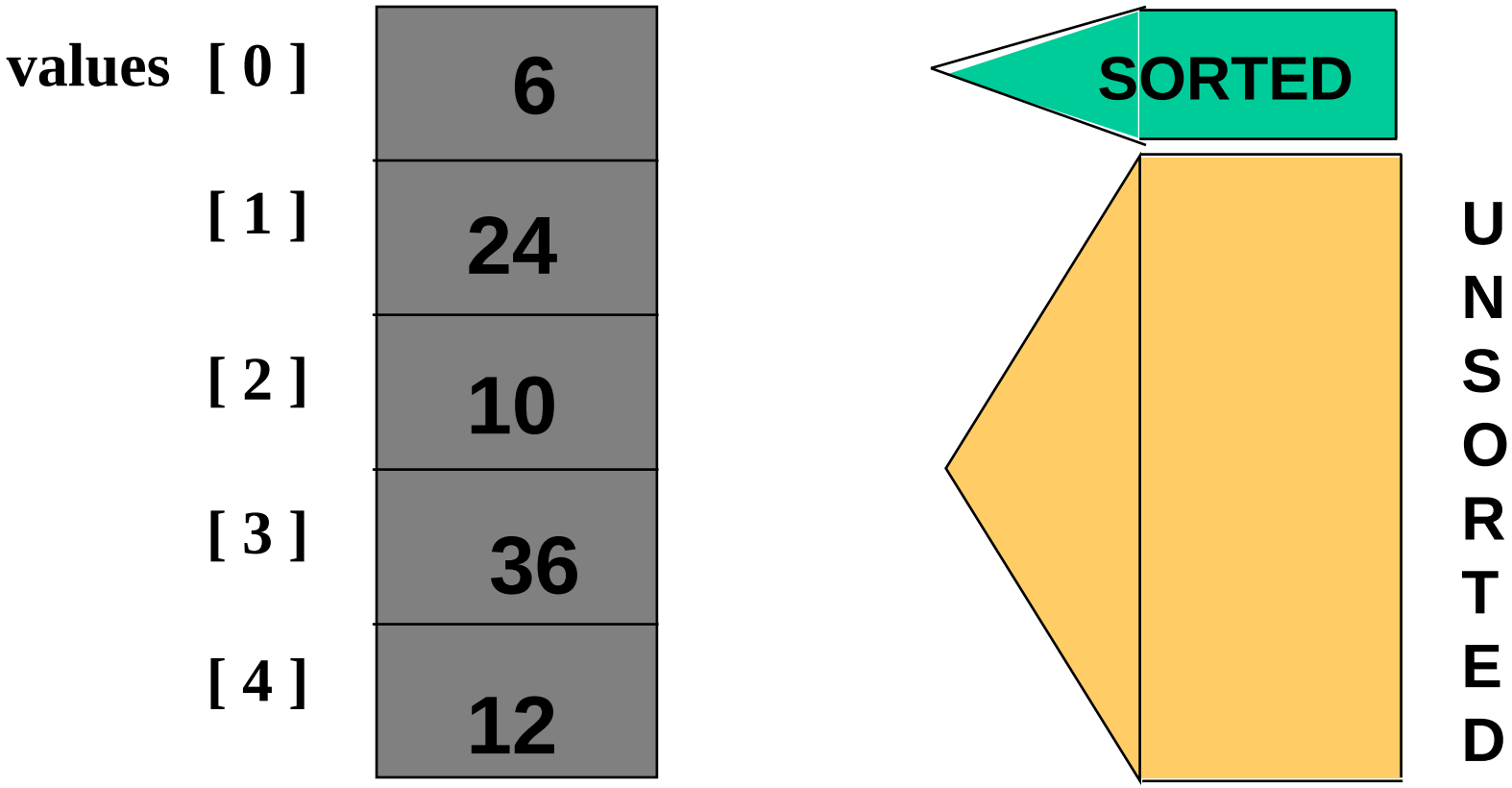
Divides the array into two parts: already sorted, and not yet sorted.

On each pass, finds the smallest of the unsorted elements, and swaps it into its correct place, thereby increasing the number of sorted elements by one.

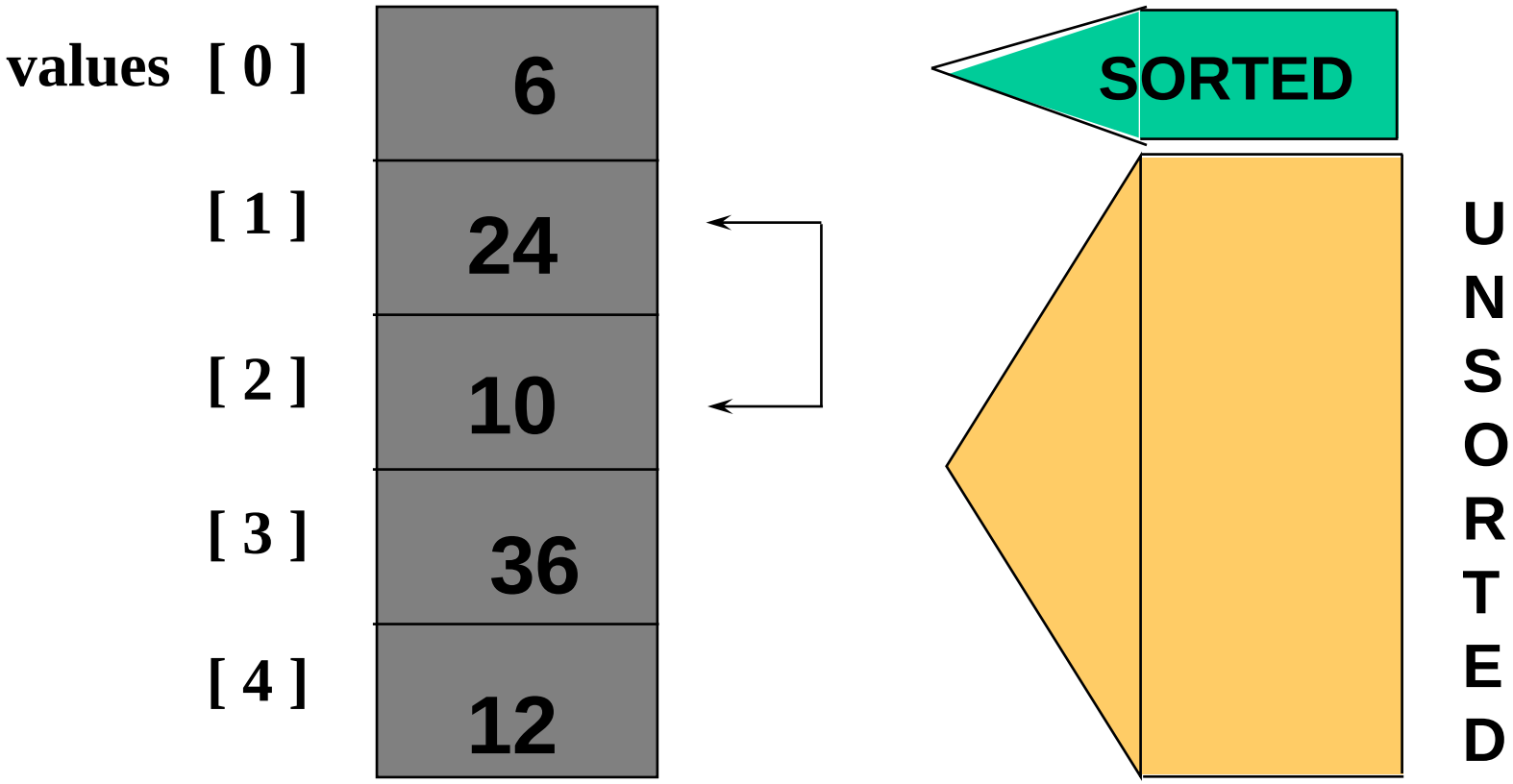
Selection Sort: Pass One



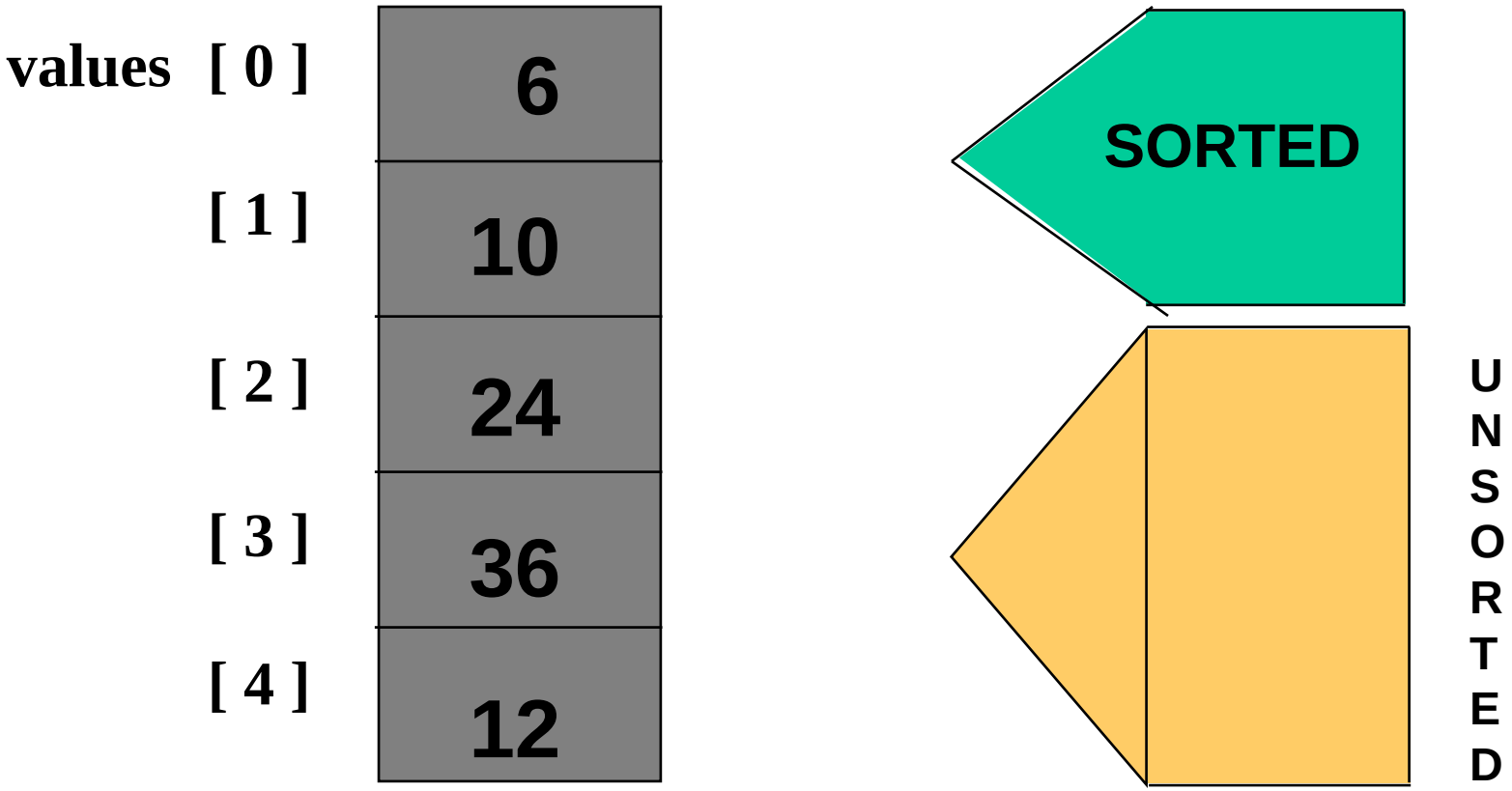
Selection Sort: End Pass One



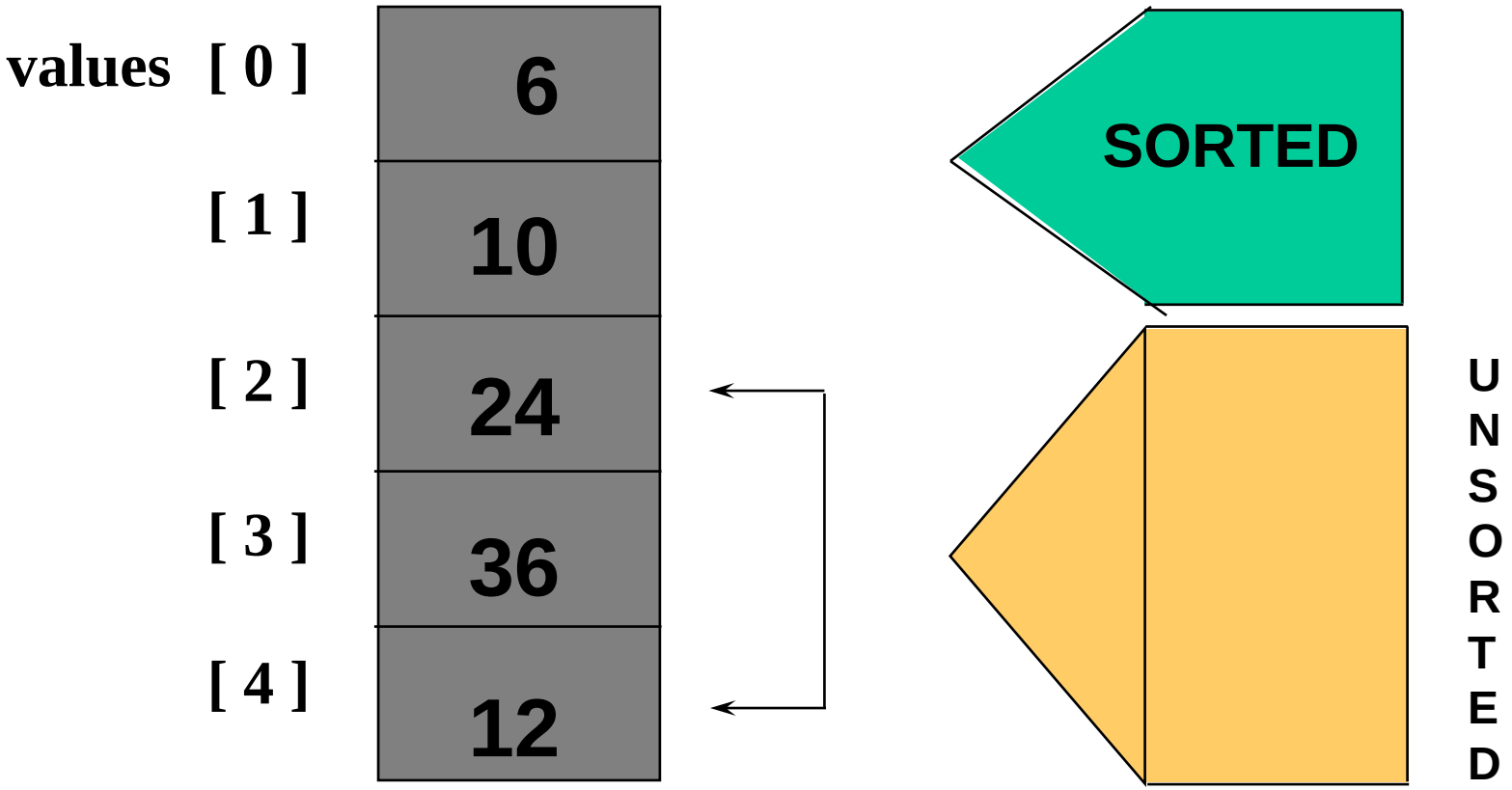
Selection Sort: Pass Two



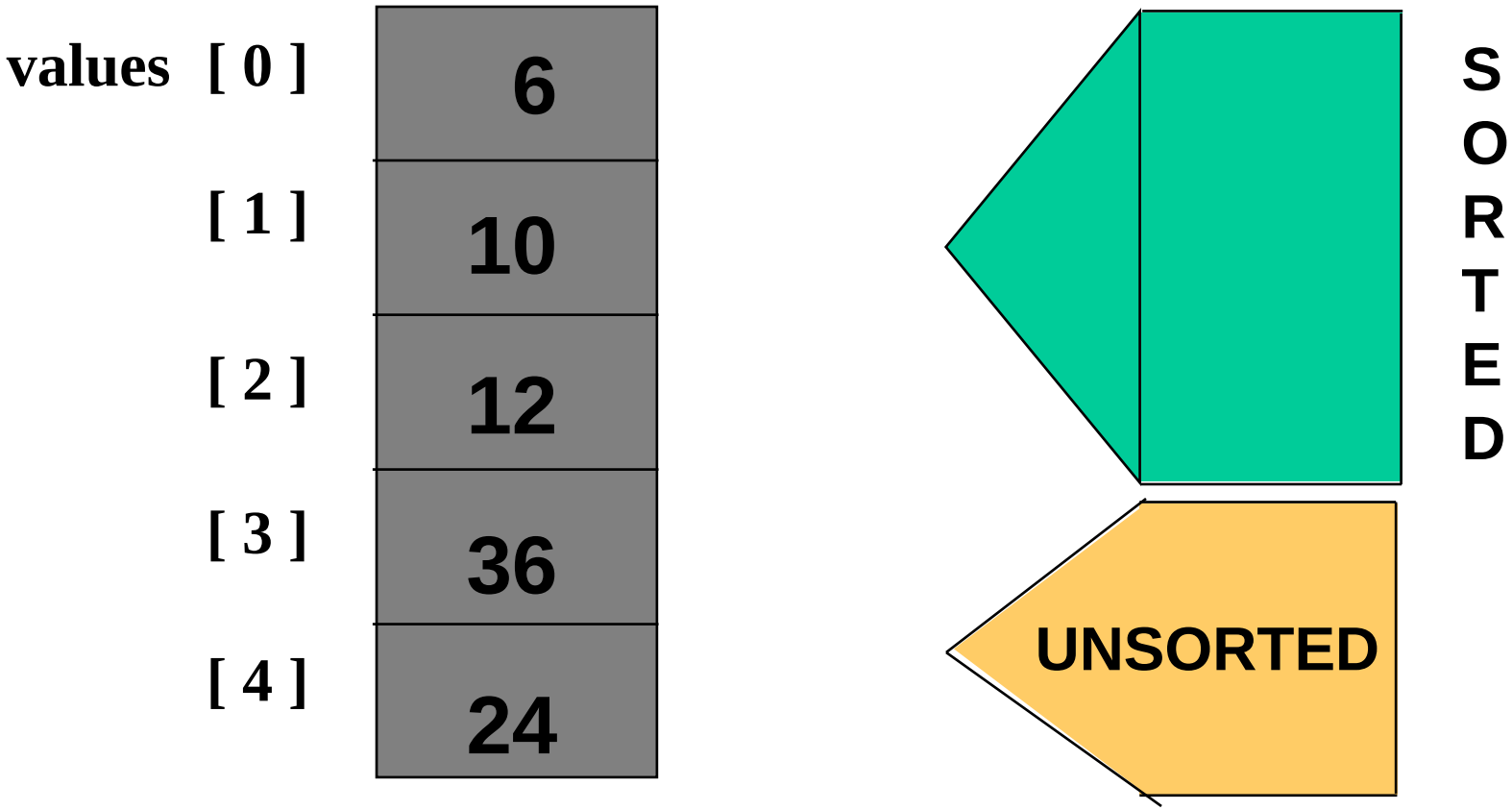
Selection Sort: End Pass Two



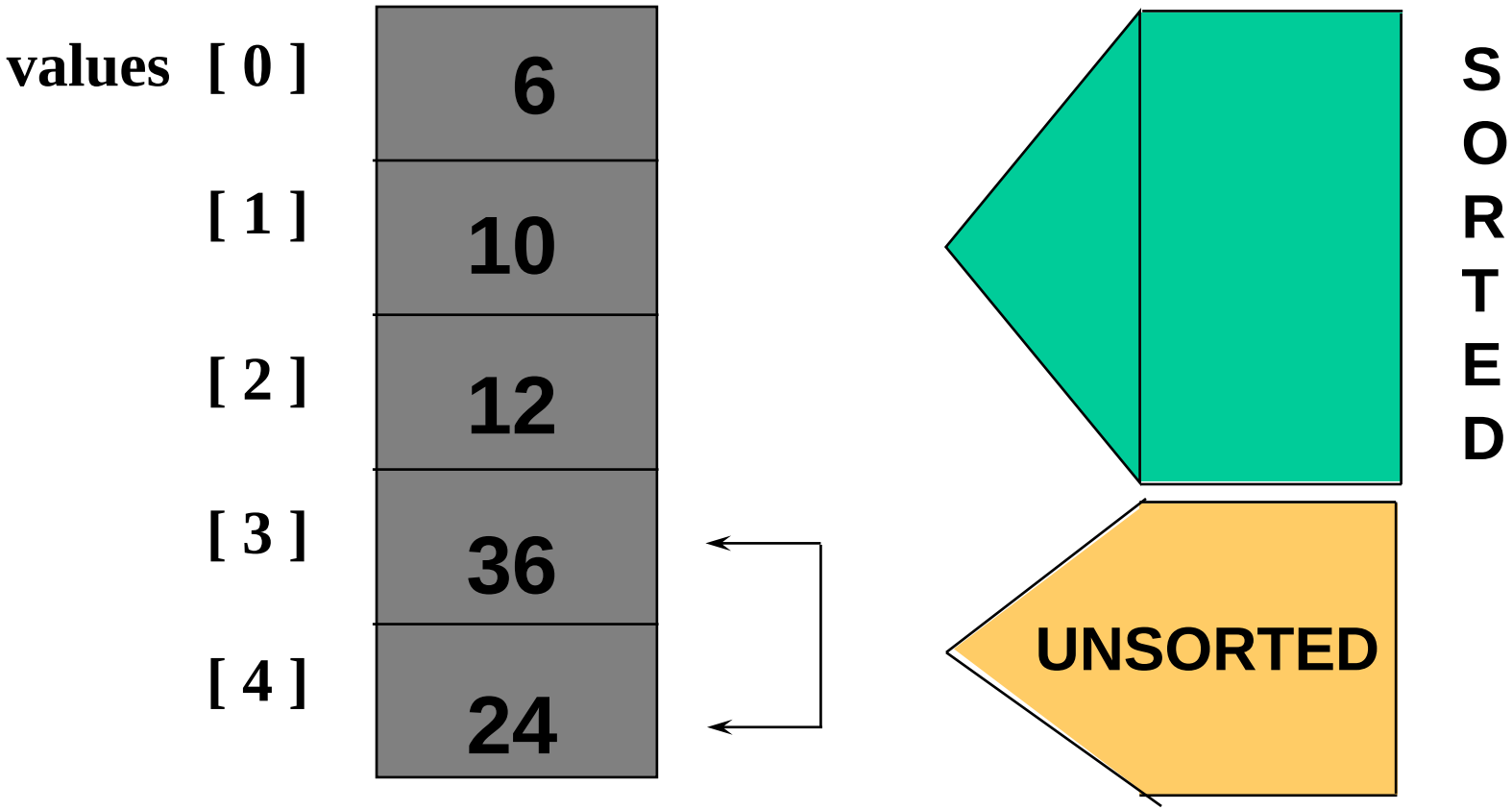
Selection Sort: Pass Three



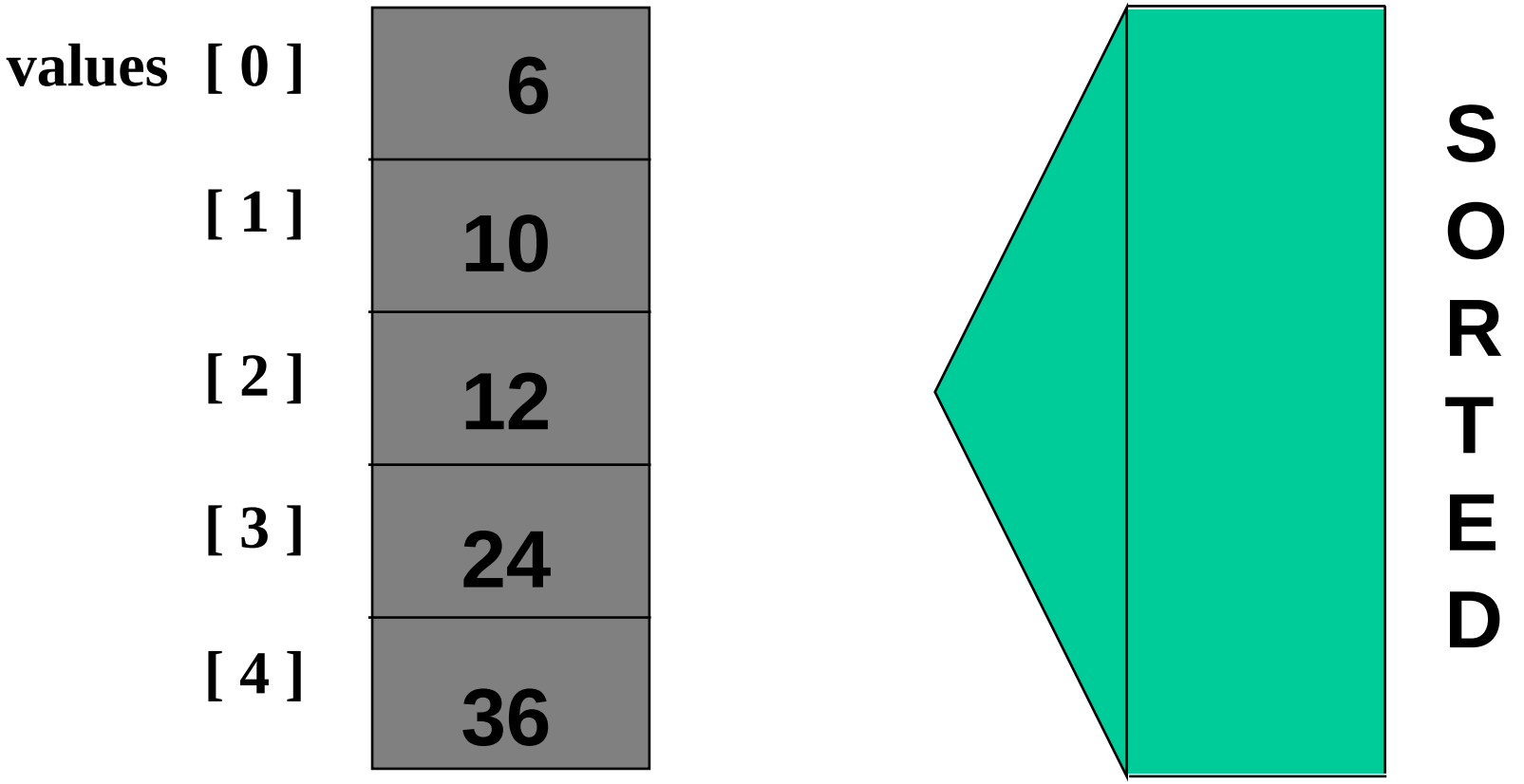
Selection Sort: End Pass Three



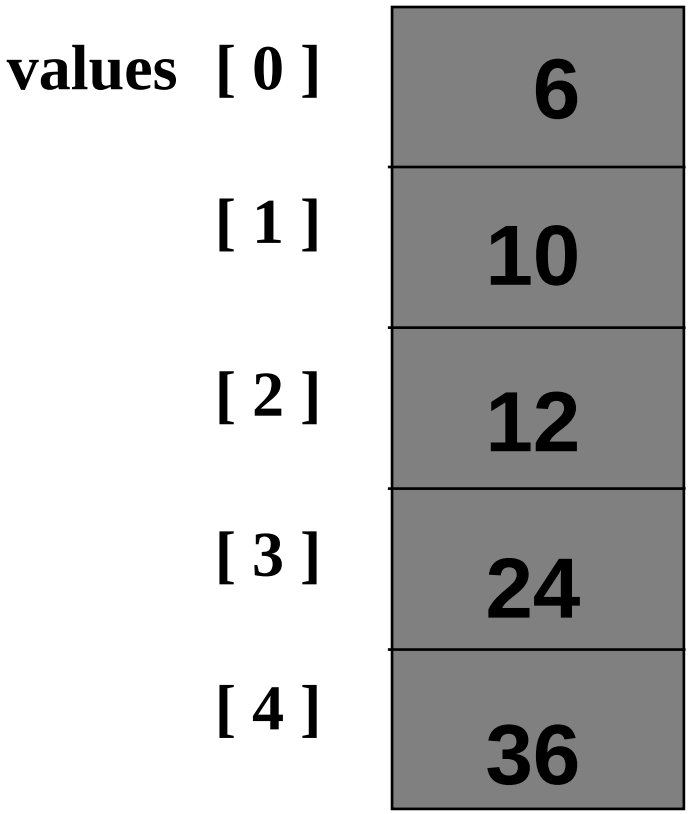
Selection Sort: Pass Four



Selection Sort: End Pass Four



Selection Sort: How many comparisons?



4 comparisons for **values[0]**

3 comparisons for **values[1]**

2 comparisons for **values[2]**

1 comparison for **values[3]**

= 4 + 3 + 2 + 1

Selection Sort: How many comparisons?

- **The number of comparisons when the array contains N elements is**

Sum = (N-1) + (N-2) + . . . + 2 + 1

Arithmetic series:

$$Sum = \sum_{i=1}^{N-1} i = \frac{(N-1)N}{2}$$

$O(N^2)$

Selection Sort

Alg.: SELECTION-SORT*(A)*

```
n ← length[A]
for j ← 1 to n - 1
    do smallest ← j
        for i ← j + 1 to n
            do if A[i] < A[smallest]
                then smallest ← i
        exchange A[j] ↔ A[smallest]
```

Analysis of Selection Sort

Alg.: SELECTION-SORT(*A*)

	cost	times
n ← length[A]	c_1	1
for j ← 1 **to** n - 1	c_2	n
do smallest ← j	c_3	n-1
for i ← j + 1 **to** n	c_4	$\sum_{j=1}^{n-1}(n-j+1)$
do if A[i] < A[smallest]	c_5	$\sum_{j=1}^{n-1}(n-j)$
then smallest ← i	c_6	$\sum_{j=1}^{n-1}(n-j)$
exchange A[j] ↔ A[smallest]	c_7	n-1

≈$n^2/2$ comparisons

≈n exchanges

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4\sum_{j=1}^{n-1}(n-j+1) + c_5\sum_{j=1}^{n-1}(n-j) + c_6\sum_{j=2}^{n-1}(n-j) + c_7(n-1) = \text{O}(n^2)$$

Running Time of Selection Sort and Bubble Sort

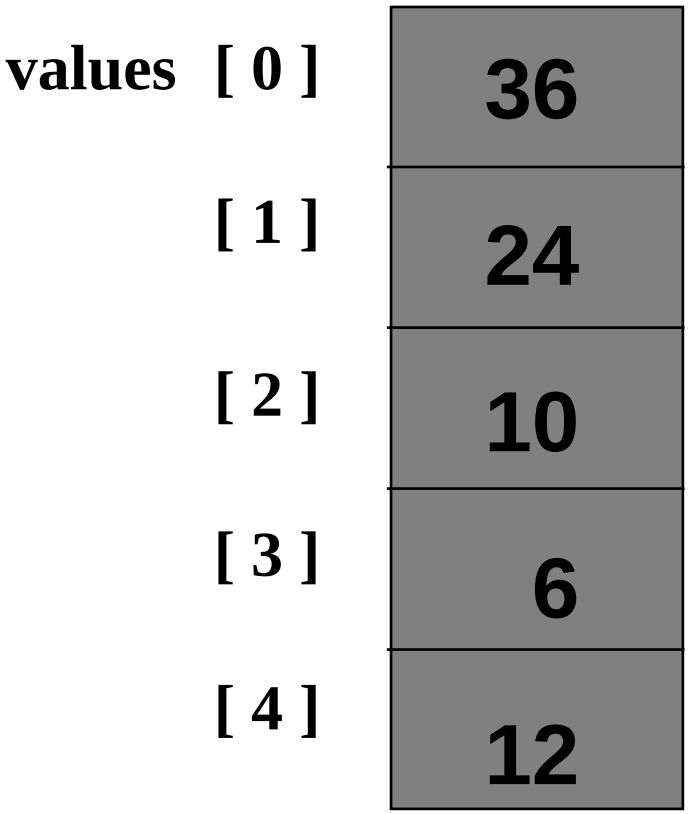
Same amount of work compared to Bubble Sort: $O(N^2)$

Lesser number of **swaps** compared to Bubble Sort!

Insertion Sort

- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

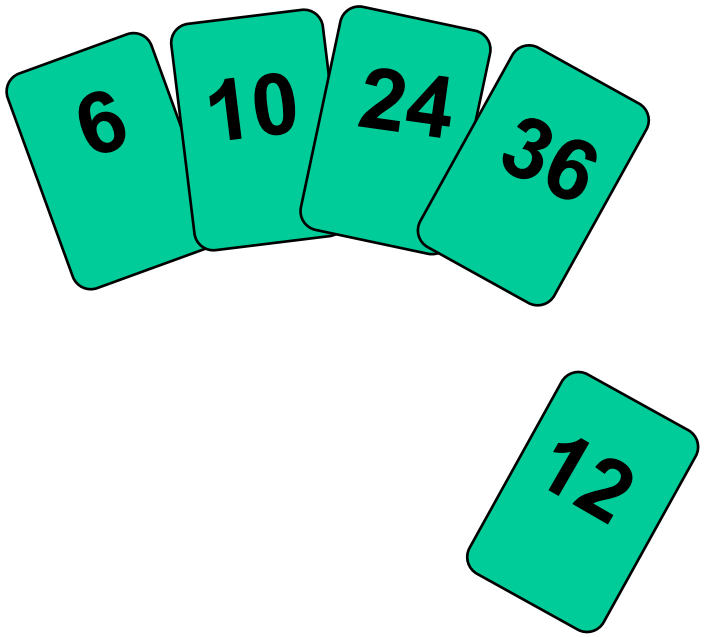
Insertion Sort



Insert, one by one, each unsorted array element into its proper place.

On each pass, this causes the number of already sorted elements to increase by one.

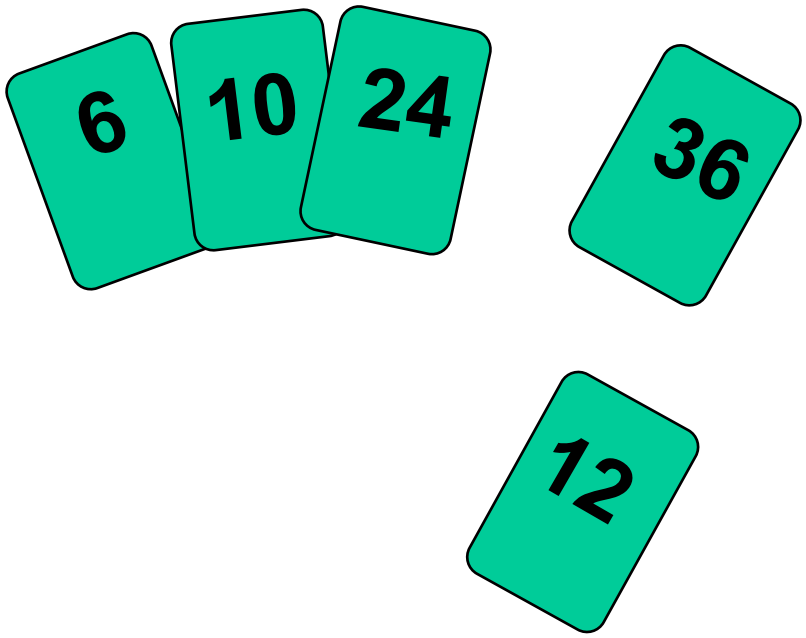
Insertion Sort



Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

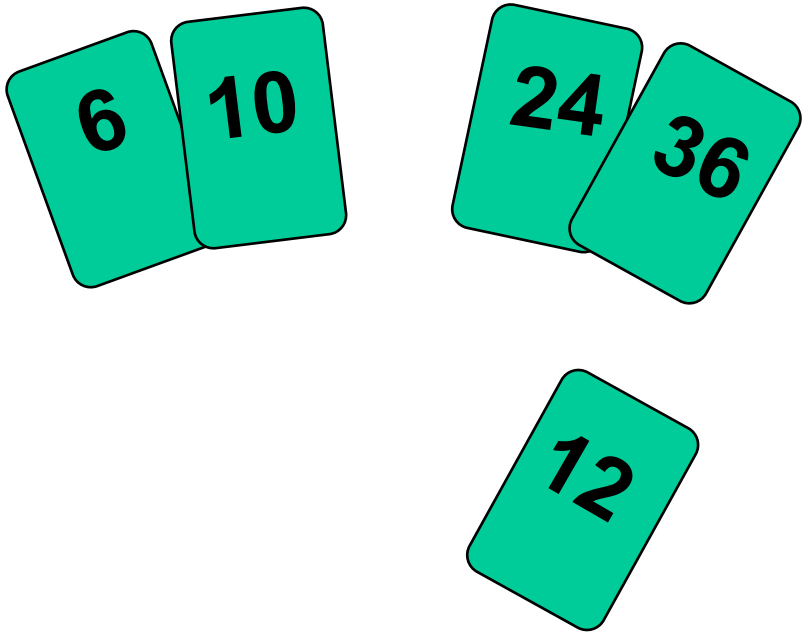
Insertion Sort



Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

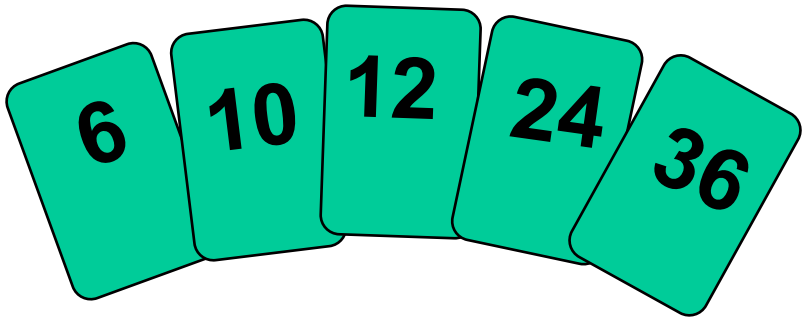
Insertion Sort



Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

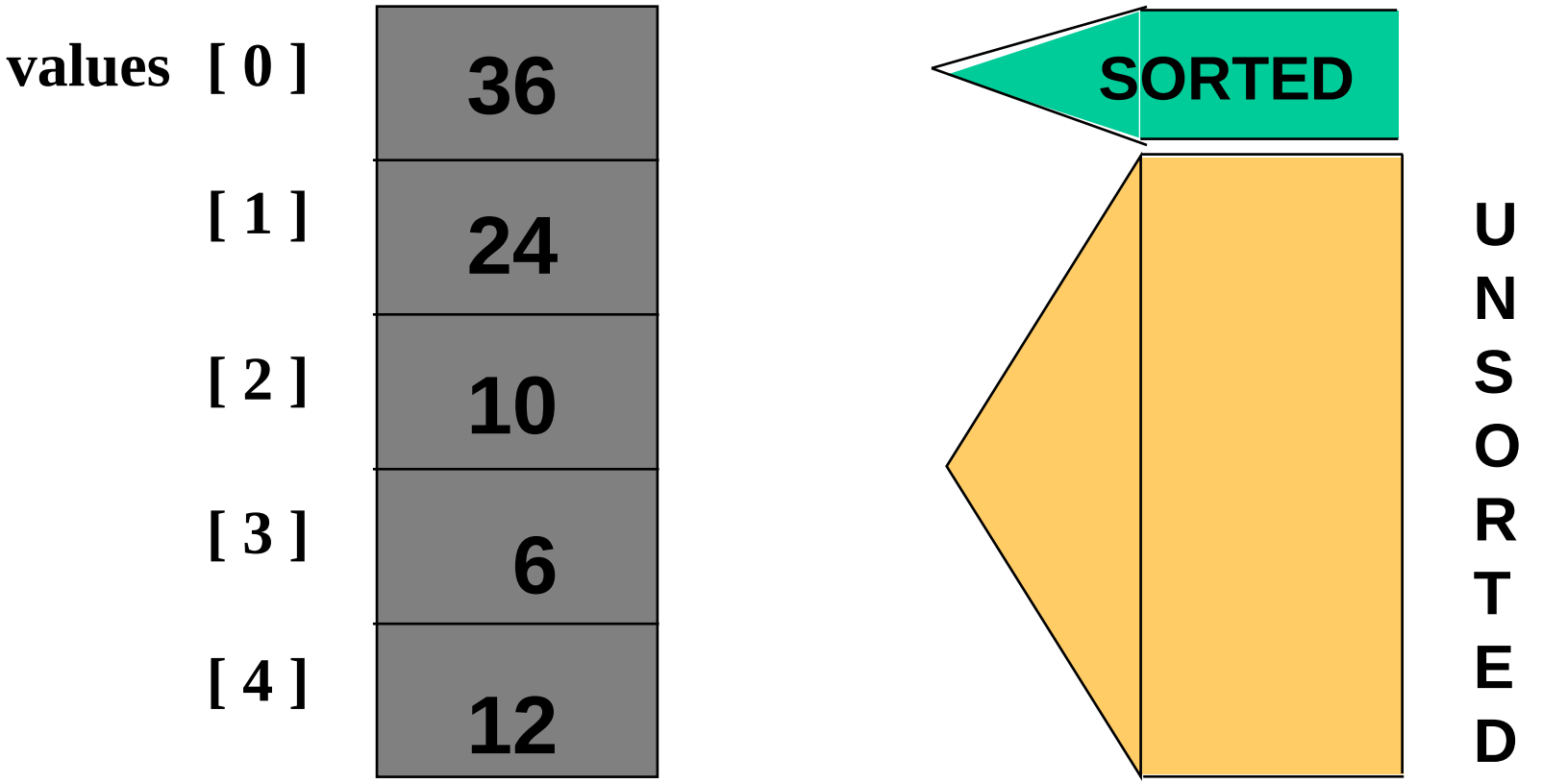
Insertion Sort



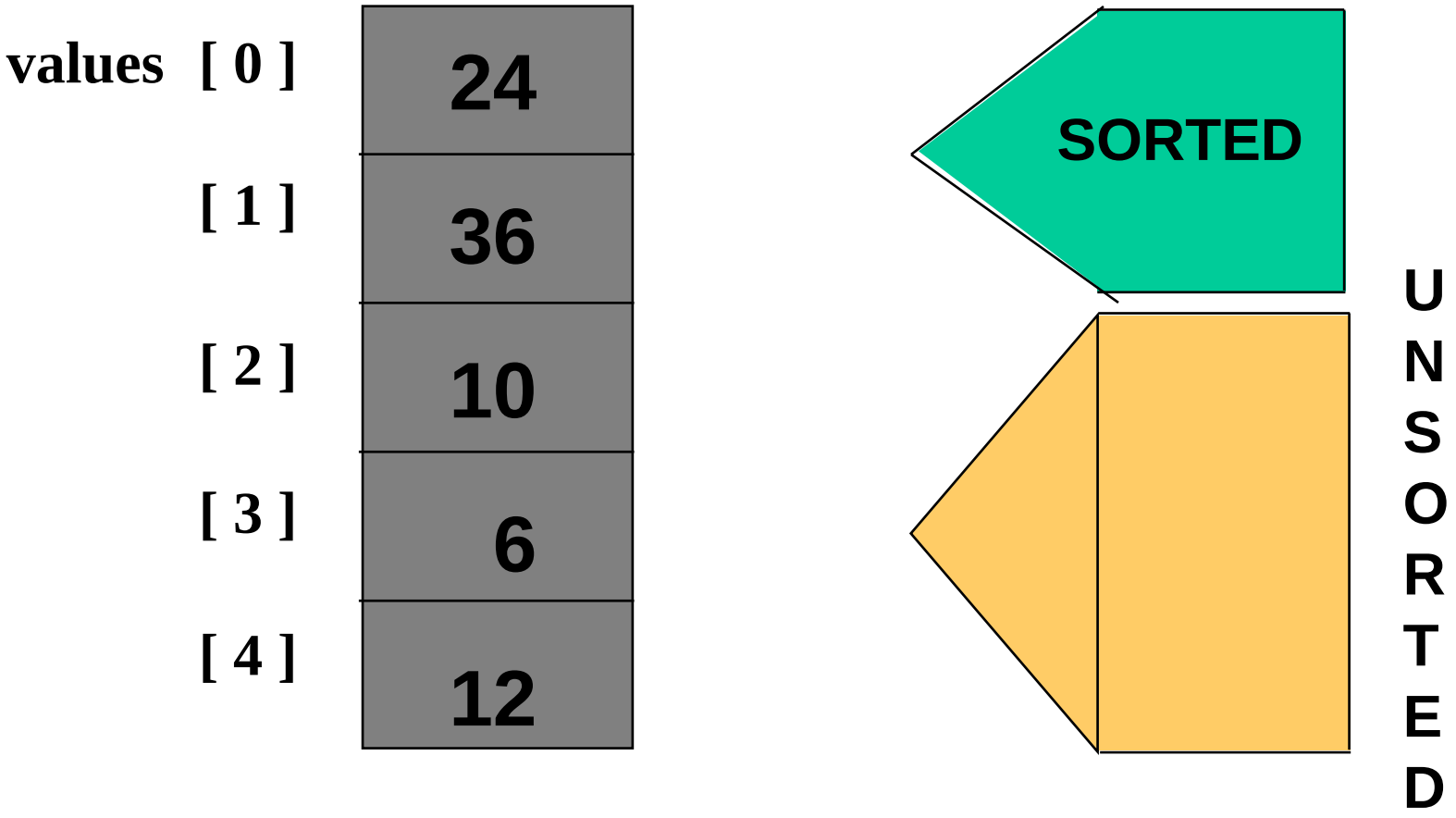
Works like someone who "inserts" one more card at a time into a hand of cards that are already sorted.

To insert 12, we need to make room for it by moving first 36 and then 24.

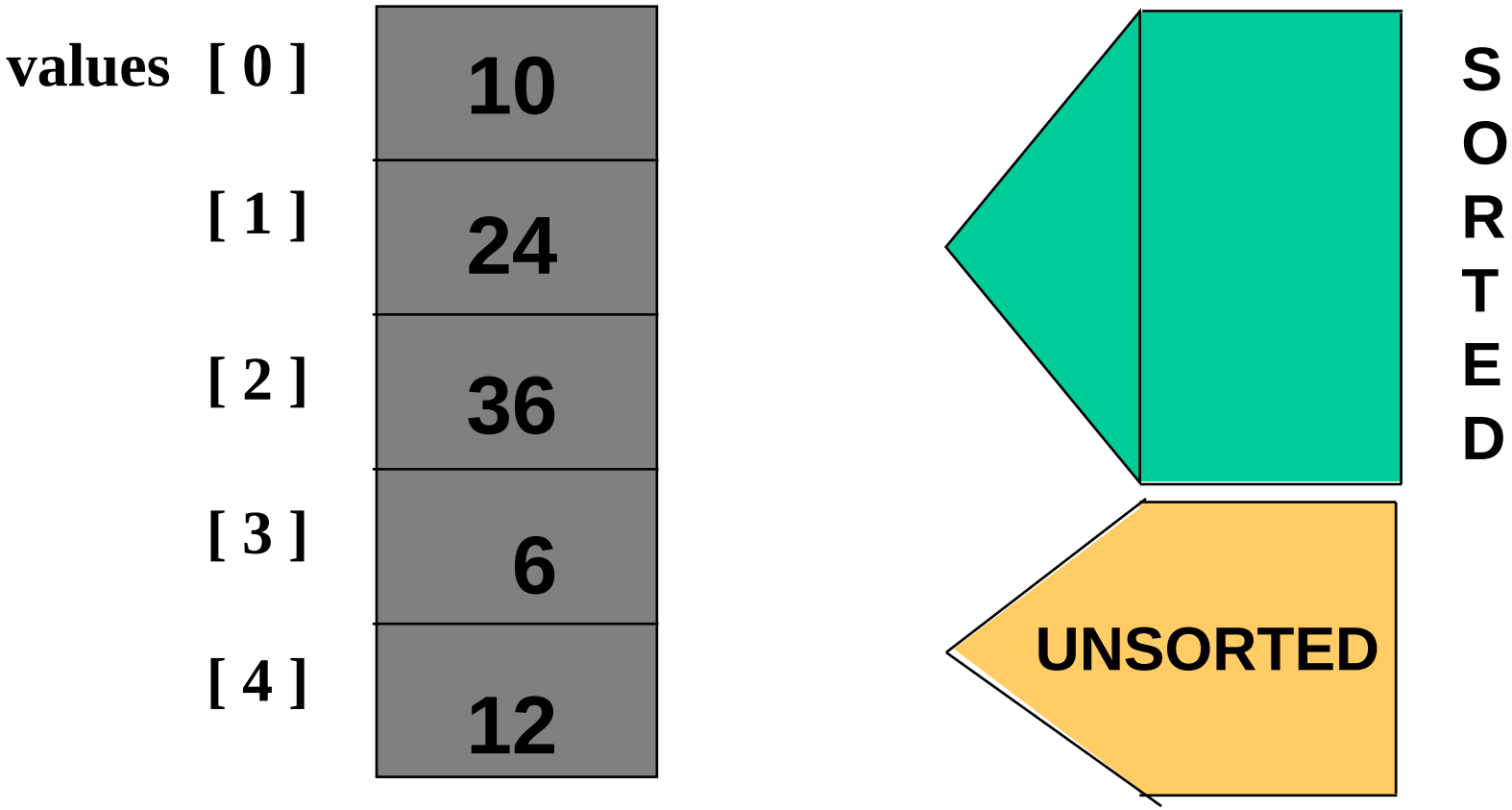
Insertion Sort: Pass One



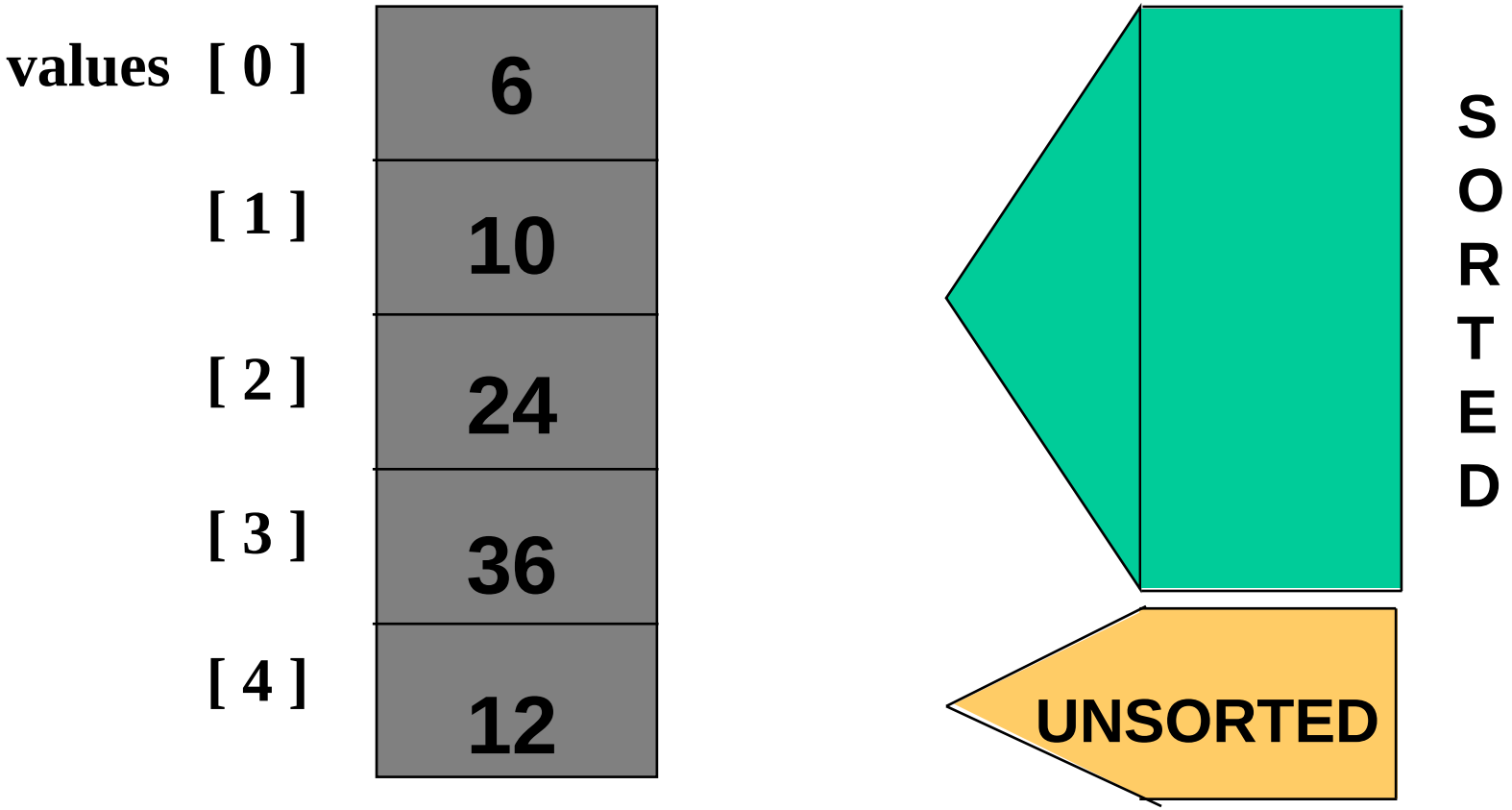
Insertion Sort: Pass Two



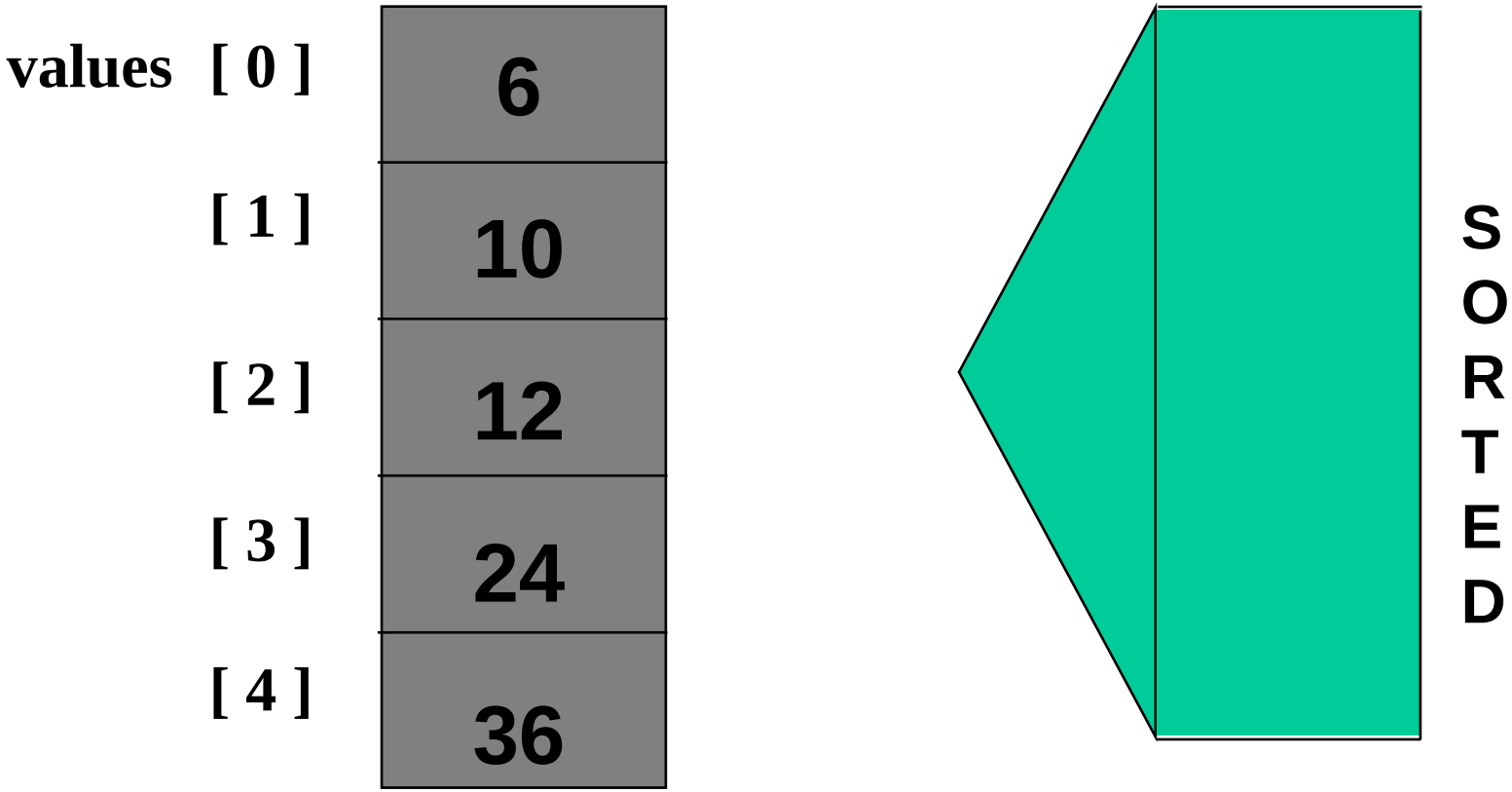
Insertion Sort: Pass Three



Insertion Sort: Pass Four



Insertion Sort: Pass Five

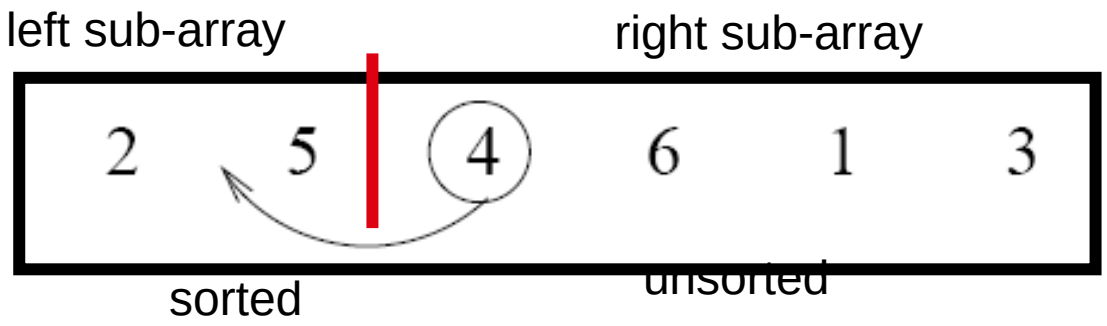


Insertion Sort

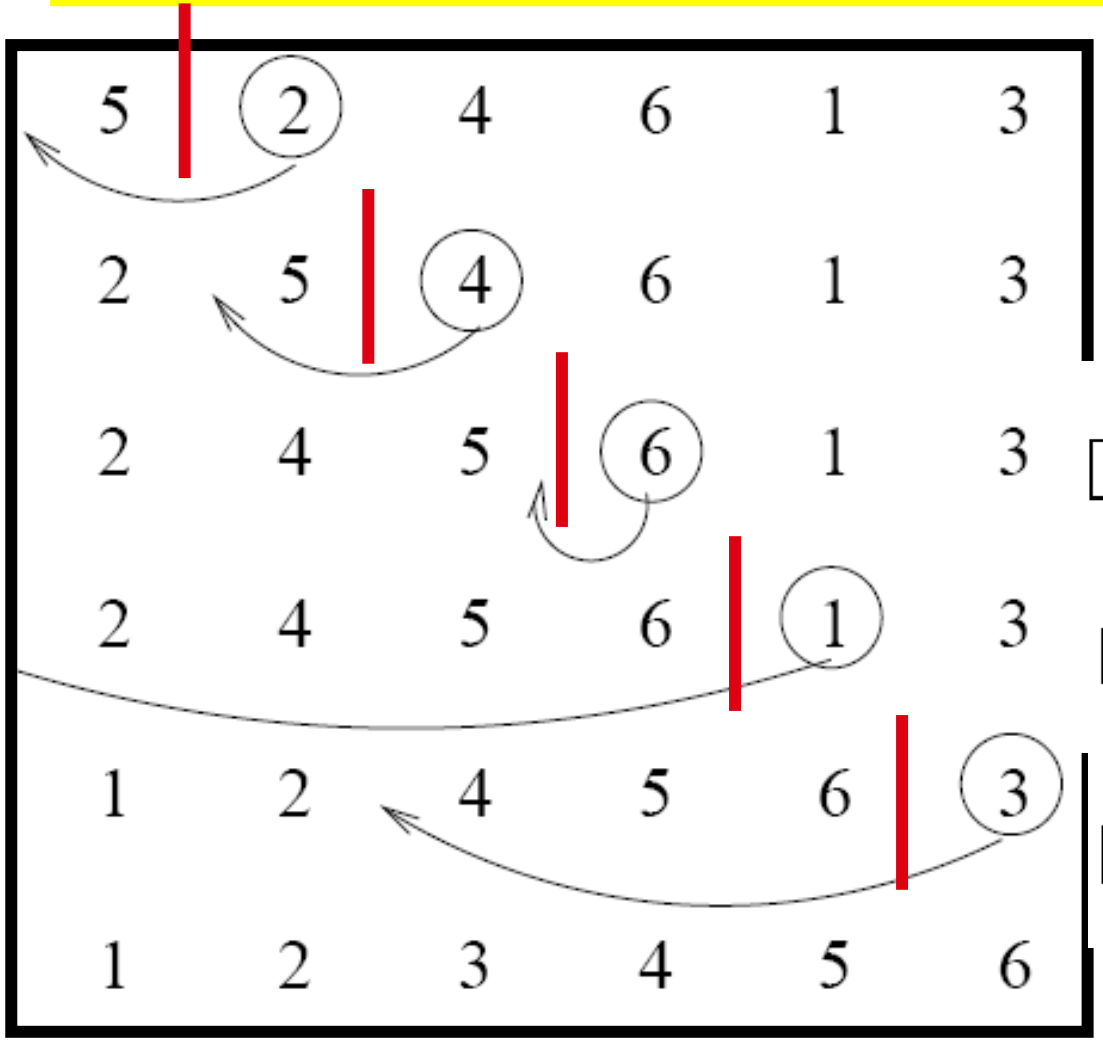
input array

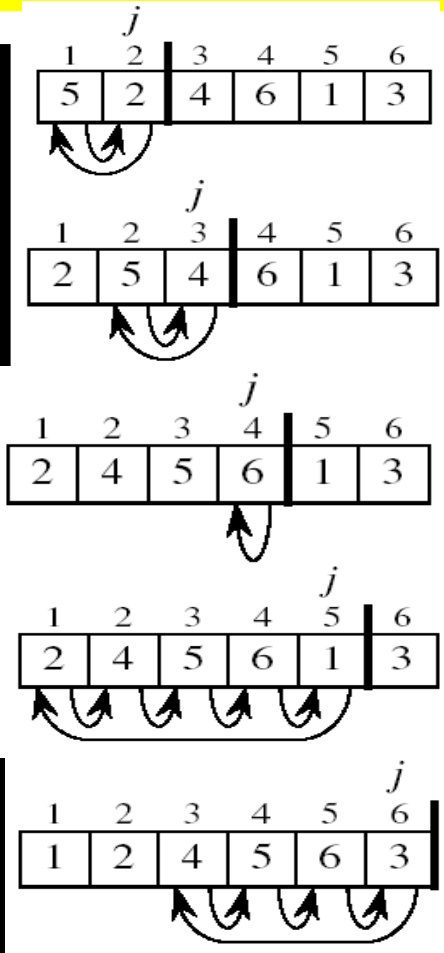
5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:



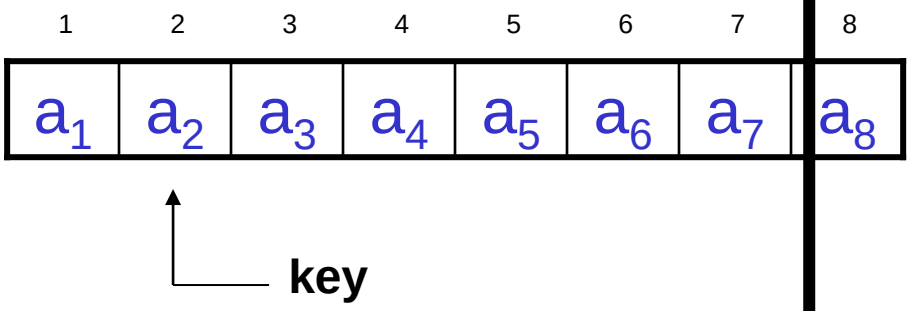
Insertion Sort





INSERTION-SORT

Alg.: INSERTION-SORT(*A*)




```
for j ← 2 to n
    do key ← A[ j ]
        ▷ Insert A[ j ] into the sorted sequence A[1 . . j -1]
        i ← j - 1
        while i > 0 and A[i] > key
            do A[i + 1] ← A[i]
                i ← i - 1
        A[i + 1] ← key
```

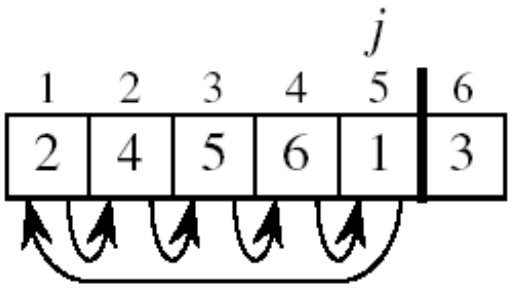
- Insertion sort – sorts the elements in place

Loop Invariant for Insertion Sort

Alg.: INSERTION-SORT*(A)*

for j ← 2 **to** n

do key ← A[j]



Insert A[j] into the sorted sequence A[1 . . j -1]

i ← j - 1

while i > 0 and A[i] > key

do A[i + 1] ← A[i]

i ← i – 1

A[i + 1] ← key

Invariant: at the start of the **for** loop the elements in A[1 . . j-1] are in sorted order

Analysis of Insertion Sort

INSERTION-SORT*(A)*	cost	times
for j ← 2 **to** n	c_1	n
do key ← A[j]	c_2	n-1
▷ Insert A[j] into the sorted sequence A[1 . . j -1]	0	n-1
i ← j - 1	c_4	n-1
while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key	c_8	n-1

t_j: # of times the while statement is executed at iteration j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Best Case Analysis

- The array is already sorted "**while** i > 0 and A[i] > key"
  - $A[i] \leq key$ upon the first time the **while** loop test is run (when $i = j - 1$)
  - $t_j = 1$
- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$

  $= (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8)$

  $= an + b = O(n)$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Worst Case Analysis

- The array is in reverse sorted order **"while** i > 0 and A[i] > key"
  - Always **A[i] > key** in **while** loop test
  - Have to compare **key** with all elements to the left of the **j**-th position
    ⇒ compare with **j-1** elements ⇒ t_j = **j**

using $\sum_{j=1}^{n} j = \frac{n(n+1)}{2} \Rightarrow \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \Rightarrow \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$ we have:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1)$$

$= an^2 + bn + c$ a quadratic function of n

- **T(n) = O(n²)** order of growth in **n²**

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Comparisons and Exchanges in Insertion Sort

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

≈ $n^2/2$ comparisons

≈ $n^2/2$ exchanges

Insertion Sort - Summary

- Advantages
  - Good running time for "almost sorted" arrays
- Disadvantages
  - $O(n^2)$ running time in worst and average case
  - $\approx n^2/2$ comparisons and exchanges

Running Time of Insertion Sort

Worst Case: $O(N^2)$

Best Case: $O(N)$

Good for sorted or almost sorted arrays!

Bubble, Selection and Insertion Sort

		Comparisons	Swaps
Bubble	Worst	n(n-1)/2	n(n-1)/2
	Best	n-1	n-1
Selection		n(n-1)/2	n-1
Insertion	Worst	n(n-1)/2	n(n-1)/2
	Best	n-1	n-1

Bubble Sort

- **<u>Advantage</u>**:
- It is the simplest sorting approach.
- Best case complexity is of **O(N)** [for optimized approach] while the array is sorted.
- Using **optimized approach**, it **can detect already sorted array in first pass** with time complexity of **O(1)**.
- **<u>Disadvantage</u>**:
- Bubble sort is comparatively slower algorithm.

Selection Sort

- **<u>Advantage</u>**:
- It can also be used on list structures that make add and remove efficient, such as a linked list. Just remove the smallest element of unsorted part and end at the end of sorted part.
- Best case complexity is of **O(N)** while the array is already sorted.
- Number of swaps reduced. **O(N)** swaps in all cases.
- **<u>Disadvantage</u>**:
- Time complexity in all cases is **$O(N^2)$**, no best case scenario.

Insertion Sort

- **Advantage**:
- It can be easily computed.
- Best case complexity is of **O(N)** while the array is already sorted.
- Number of swaps reduced than bubble sort.
- For smaller values of N, insertion sort performs efficiently like other quadratic sorting algorithms.
- Adaptive: total number of steps is reduced for partially sorted array.
- **Disadvantage**:
- It is generally used when the value of N is small.
  For **larger values of N**, it is **inefficient**.