Algorithm Analysis and Big Oh Notation

Courtesy of Prof. Ajay Gupta
(with updates by Dr. Leszek T. Lilien)

CS 1120 – Fall 2007

Department of Computer Science
Western Michigan University

Measuring the Efficiency of Algorithms

- Analysis of algorithms
  - Area of computer science
  - Provides tools for determining efficiency of different methods of solving a problem
    - E.g., the sorting problem - which sorting method is more efficient
  - Comparing the efficiency of different methods of solution.
    - Concerned with significant differences
    - E.g.:
      - n - the number of items to be sorted
      - Is the running time proportional to n or proportional to n^2?
      - Big difference: e.g., for n =100 it results in 100-fold difference; for n= 1000 it results in 1000-fold difference; for n = …

How To Do Algorithm Comparison?

- Approach 1:
  Implement the algorithms in C#, run the programs, measure their performance, compare
  - Many issues that affect the results:
    - How are the algorithms coded?
    - What computer should you use?
    - What data should the programs use?
- Approach 2:
  Analyze algorithms independently of their implementations
  - How?
  - For measuring/comparing execution time of algorithms
    - Count the number of basic operations of an algorithm
    - Summarize the count

The Execution Time of Algorithms

- Count the number of basic operations of an algorithm
  - Read, write, compare, assign, jump, arithmetic operations (increment, decrement, add, subtract, multiply, divide), open, close, logical operations (not/complement, AND, OR, XOR), …

The Execution Time of Algorithms

- Counting an algorithm's operations
  - Example: calculating a sum of array elements

```
int sum = item[0];     <- 1 assignment
int j = 1;             <- 1 assignment
while (j < n)          <- n comparisons
{
   sum += item[j];     <- n-1 plus/assignments
   ++j;                <- n-1 plus/assignments
}
                          Total: 3n operations
```

- Notice:

  Problem size = n = number of elements in an array

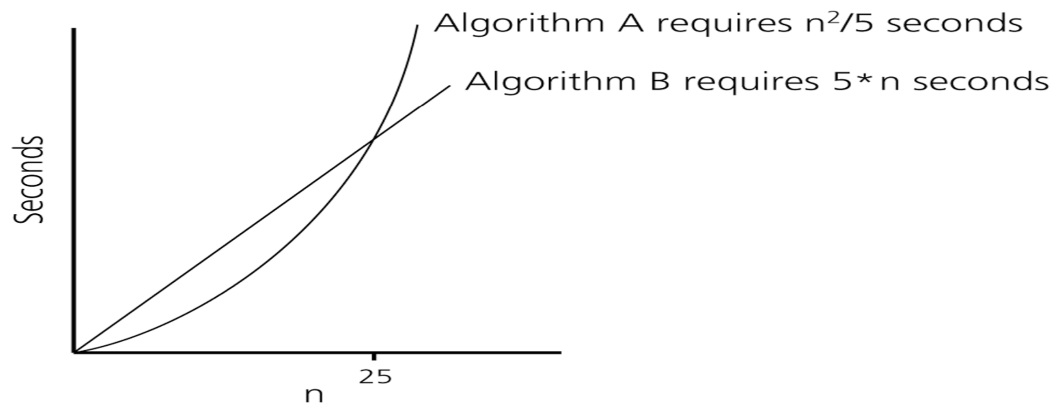
  This problem of size n requires solution with 3n operations

Algorithm Growth Rates

- Measure an algorithm's time requirement as a function of the problem size
  - E.g., problem size = number of elements in an array

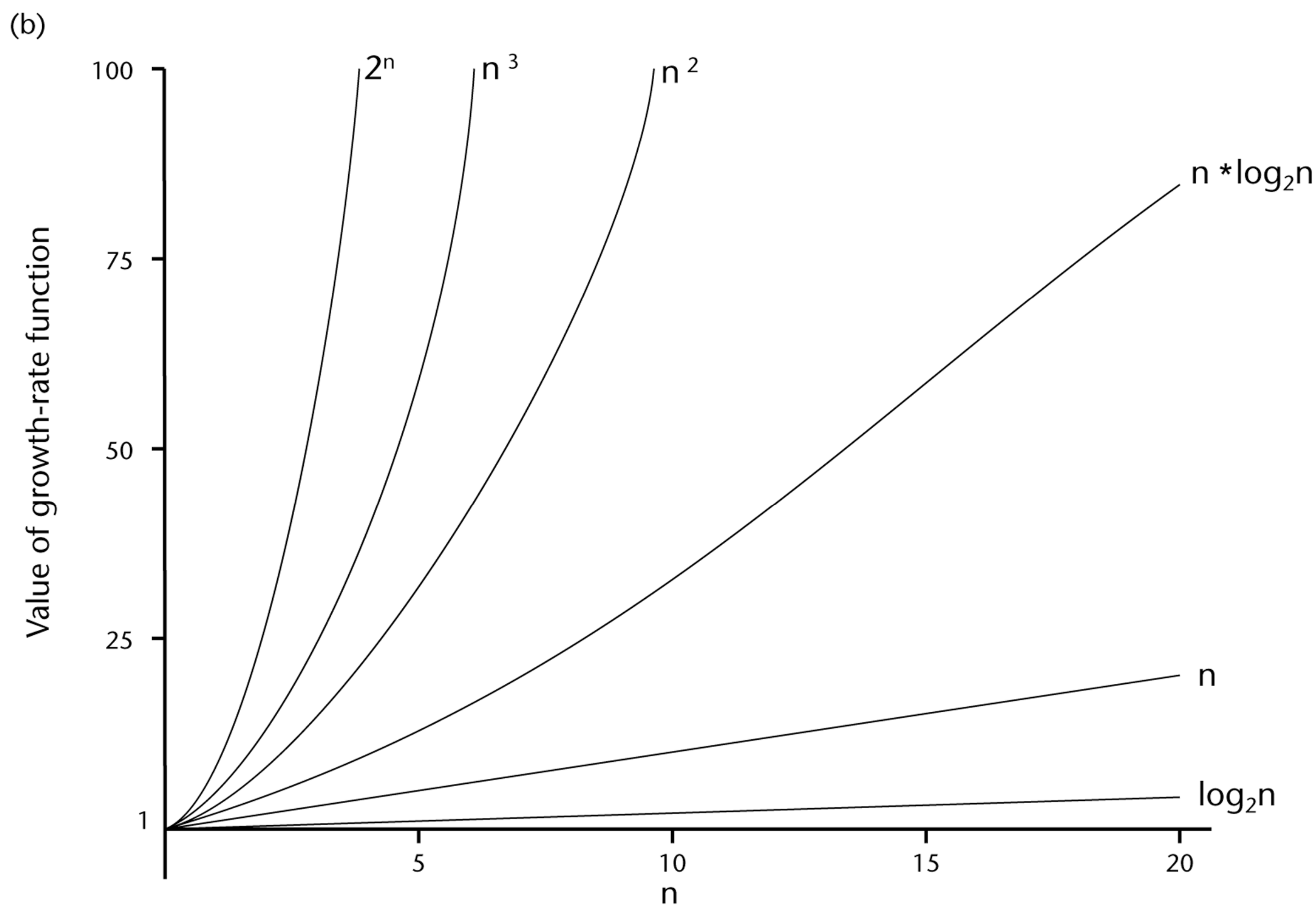
    `Algorithm A requires n²/5 time units`

    `Algorithm B requires 5n time units`



- Algorithm efficiency is a concern for large problems only
  - For smaller values of n, $n^2/5$ and 5n not "that much" different
    - Imagine how big is the difference for n > 1,000,000

Common Growth-Rate Functions - I



Common Growth-Rate Functions - II

(a)

Function	n = 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

- Differences among the growth-rate functions grow with *n*
  - See the differences growing on the diagram on the previous page
  - The bigger n, the bigger differences -
    - that's why algorithm efficiency is "concern for large problems only"

Similar table from the textbook:

n =	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
1	0	1	0	1
2	1	2	2	4
3	1	3	3	9
4	1	4	4	16
5	1	5	5	25
10	1	10	10	100
100	2	100	200	10000
1,000	3	1000	3000	10^6
1,000,000	6	1000000	6000000	10^{12}
1,000,000,000	9	1000000000	9000000000	10^{18}

Fig. 24.13 | Number of comparisons for common Big O notations.

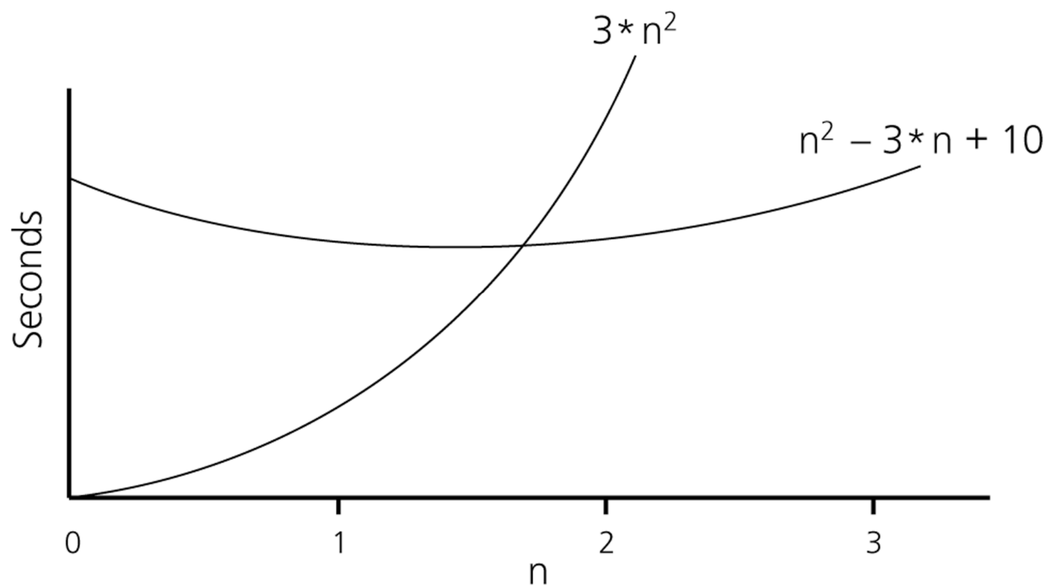
Big-Oh Notation

- Algorithm A is order f(n) —denoted O(f(n))— if there exist constants k and n_0 such that A requires <= k*f(n) time units to solve a problem of size n >= n_0

- Examples:
  - $n^2/5$
    - $O(n^2)$: $k=1/5$, $n_0=0$
  - 5*n
    - $O(n)$: $k=5$, $n_0=0$

More Examples

- How about $n^2-3n+10$?
  - It is $O(n^2)$ if there exist k and n_0 such that
    $$kn^2 \geq n^2-3n+10 \text{ for all } n \geq n_0$$
  - We see (fig.) that: $3n^2 \geq n^2-3n+10$ for all $n \geq 2$
  - So $k=3$, $n_0=2$
    - More k-n_0 pairs could be found, but finding just one is enough to prove that $n^2-3n+10$ is $O(n^2)$



Properties of Big-Oh

- Ignore low-order terms
  - E.g., $O(n^3+4n^2+3n)=O(n^3)$

- Ignore multiplicative constant
  - E.g., $O(5n^3)=O(n^3)$

- Combine growth-rate functions
  - $O(f(n)) + O(g(n)) = O(f(n)+g(n))$
  - E.g., $O(n^2) + O(n*\log_2 n) = O(n^2 + n*\log_2 n)$
    - Then, $O(n^2 + n*\log_2 n) = O(n^2)$

Worst-case vs. Average-case Analyses

- An algorithm can require different times to solve different problems of the same size.
- Worst-case analysis = find the maximum number of operations an algorithm can execute in all situations
  - Worst-case analysis is easier to calculate
  - More common
- Average-case analysis = enumerate all possible situations, find the time of each of the m possible cases, total and divide by m
  - Average-case analysis is harder to compute
  - Yields a more realistic expected behavior

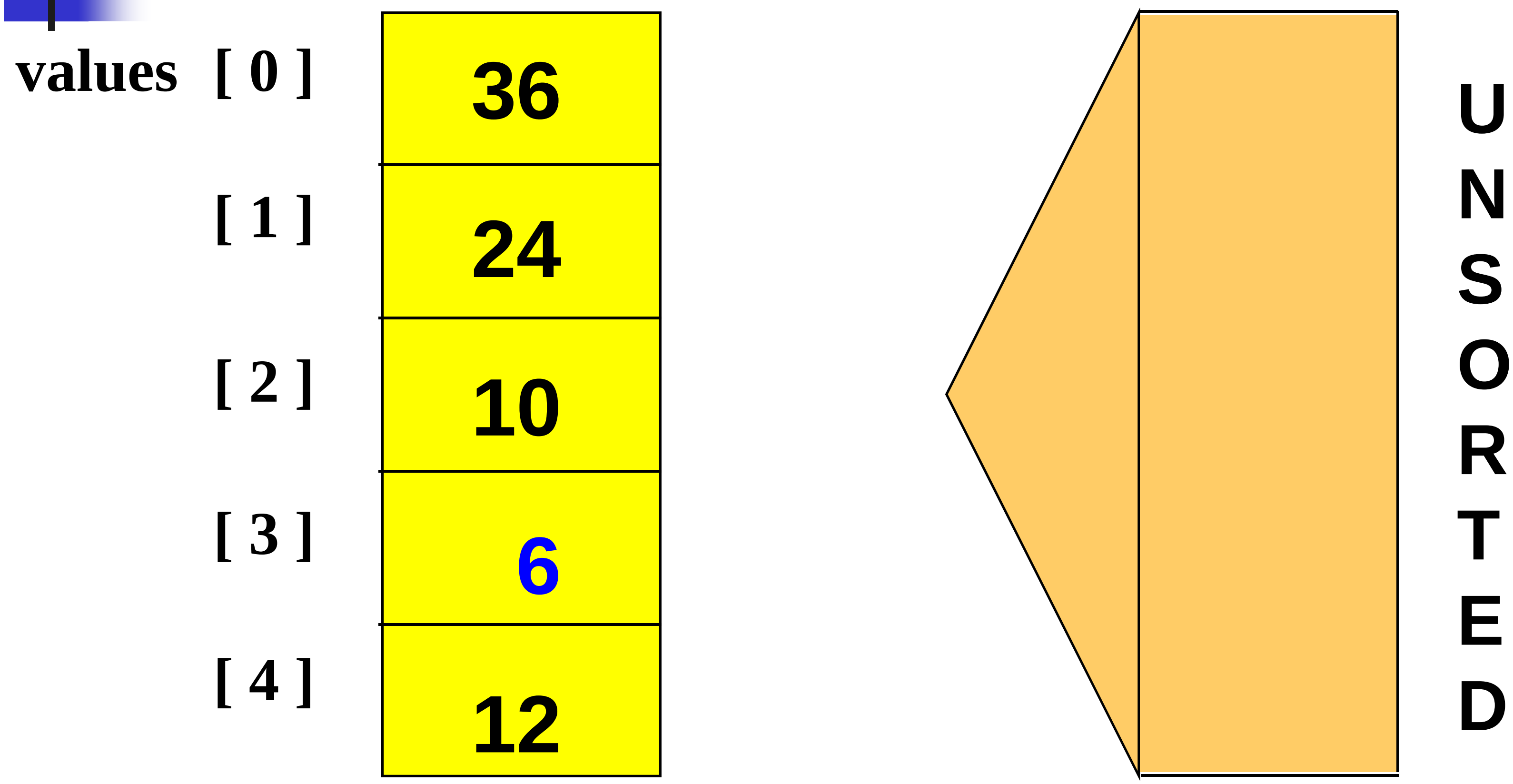
Bigger Example: Analysis of Selection Sort

values	[0]	36
	[1]	24
	[2]	10
	[3]	6
	[4]	12

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

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

Selection Sort: Pass One



To find the smallest in UNSORTED:

indexMin = 0

comp. 1: check if values[1] = 24 < values[indexMin] = 36 - yes => indexMin = 1

comp. 2: check if values[2] = 10 < values[indexMin] = 24 - yes => indexMin = 2

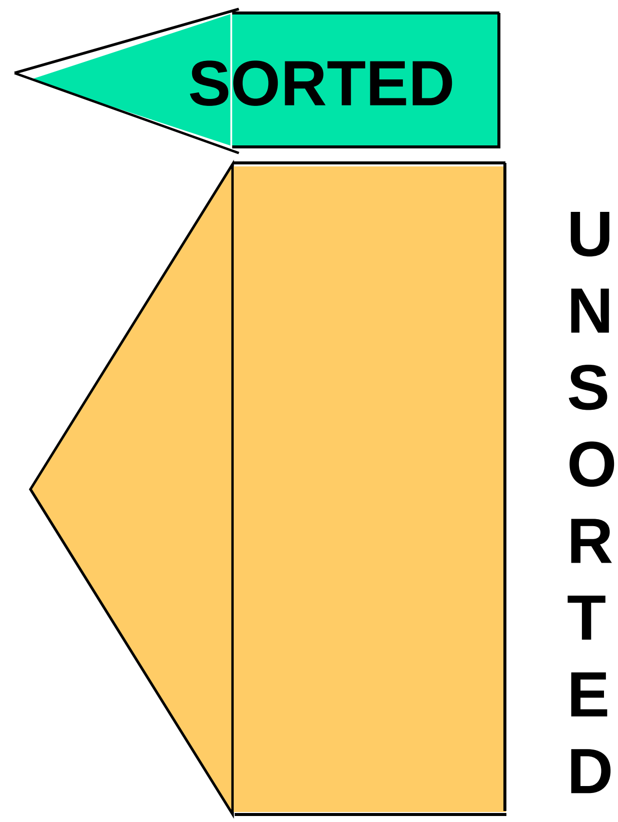
comp. 3: check if values[3] = 6 < values[indexMin] = 10 - yes => indexMin = 3

comp. 4: check if values[4] = 12 < values[indexMin] = 6 - NO

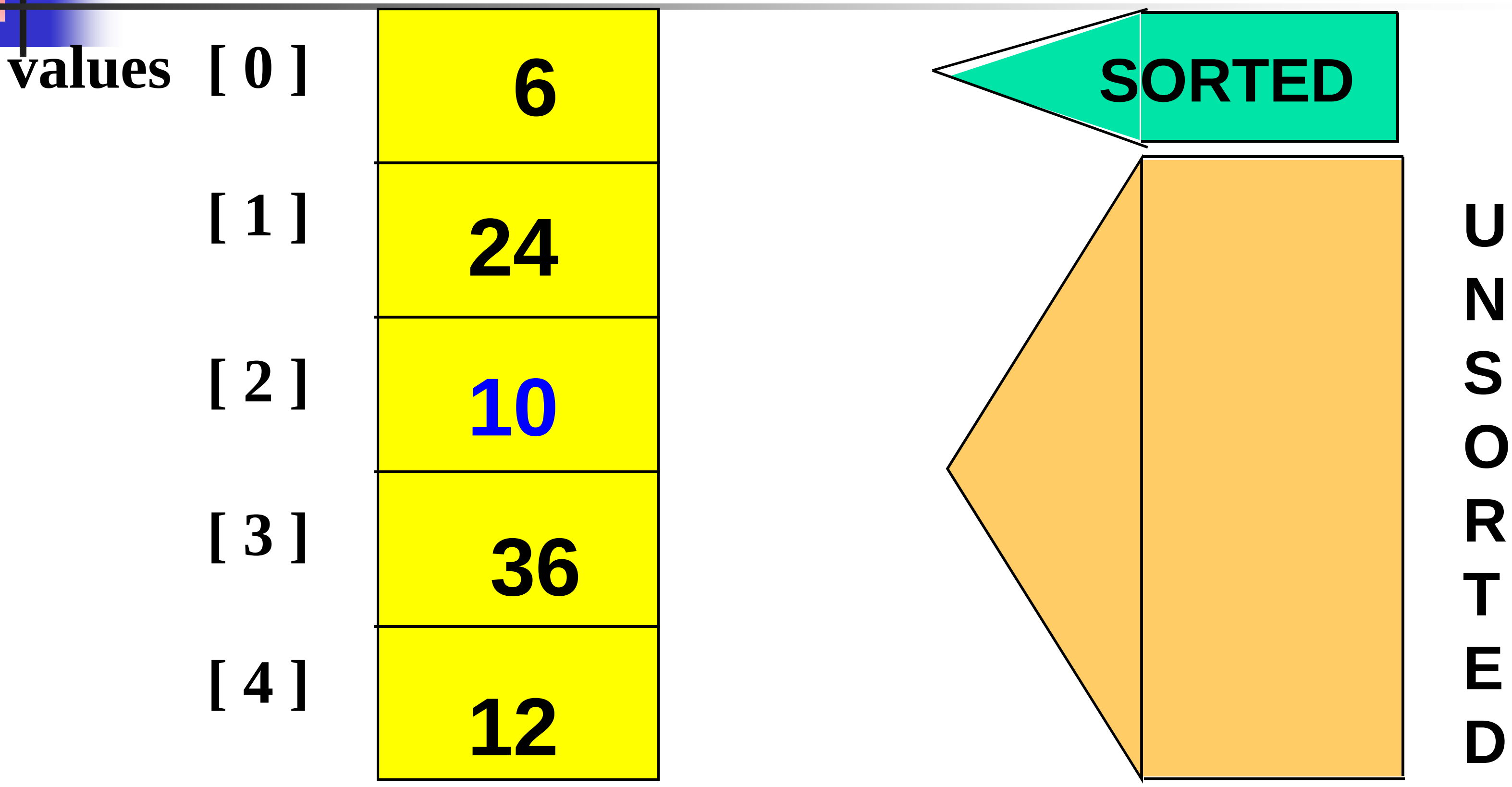
Thus indexMin = 3; swap values[0] = 36 with values[indexMin] = 6 – see next slide

Selection Sort: End of Pass One

values	[0]	6
	[1]	24
	[2]	10
	[3]	36
	[4]	12



Selection Sort: Pass Two



To find the smallest in UNSORTED:

indexMin = 1

comp. 1: check if values[2] = 10 < values[indexMin] = 24 - yes => indexMin = 2

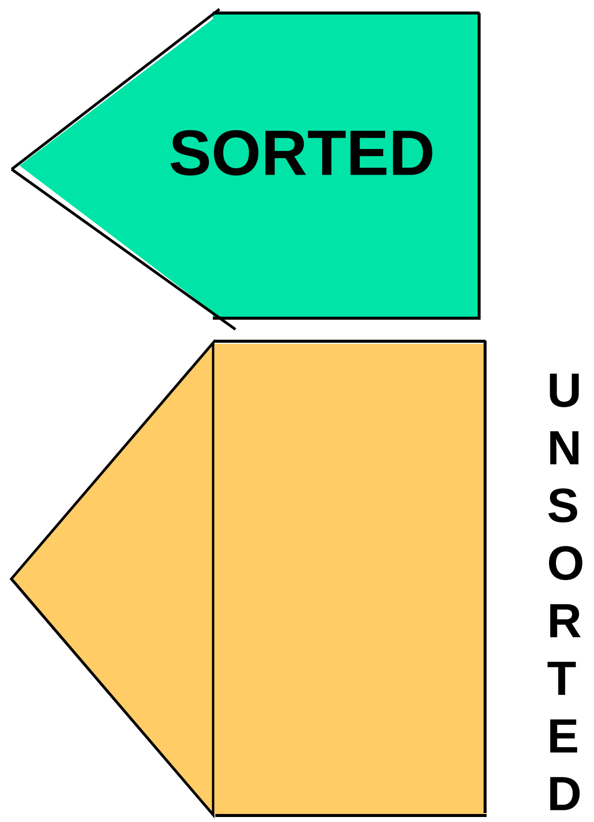
comp. 2: check if values[3] = 36 < values[indexMin] = 10 - NO

comp. 3: check if values[4] = 12 < values[indexMin] = 10 - NO

Thus indexMin = 2; swap values[1] = 24 with values[indexMin] = 10 – see next slide

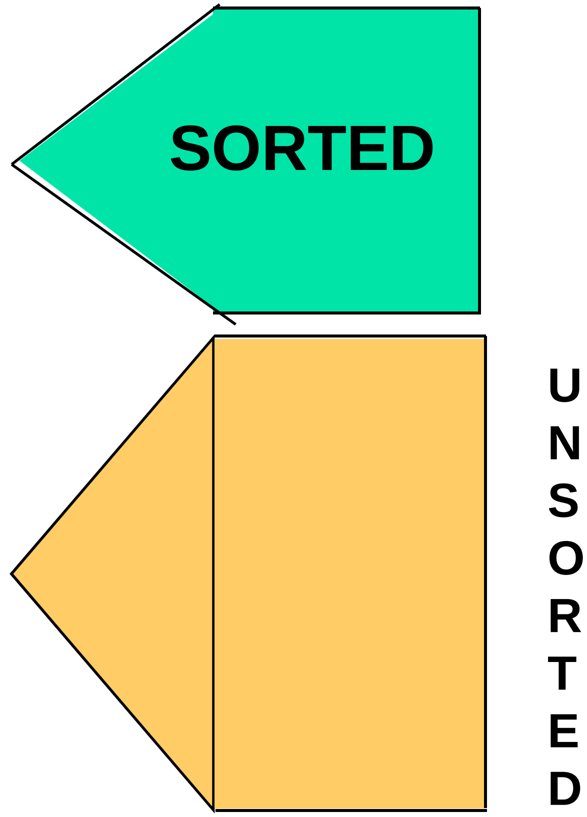
Selection Sort: End of Pass Two

values	[0]	6
	[1]	10
	[2]	24
	[3]	36
	[4]	12



Selection Sort: Pass Three

values	
[0]	6
[1]	10
[2]	24
[3]	36
[4]	12



To find the smallest in UNSORTED:

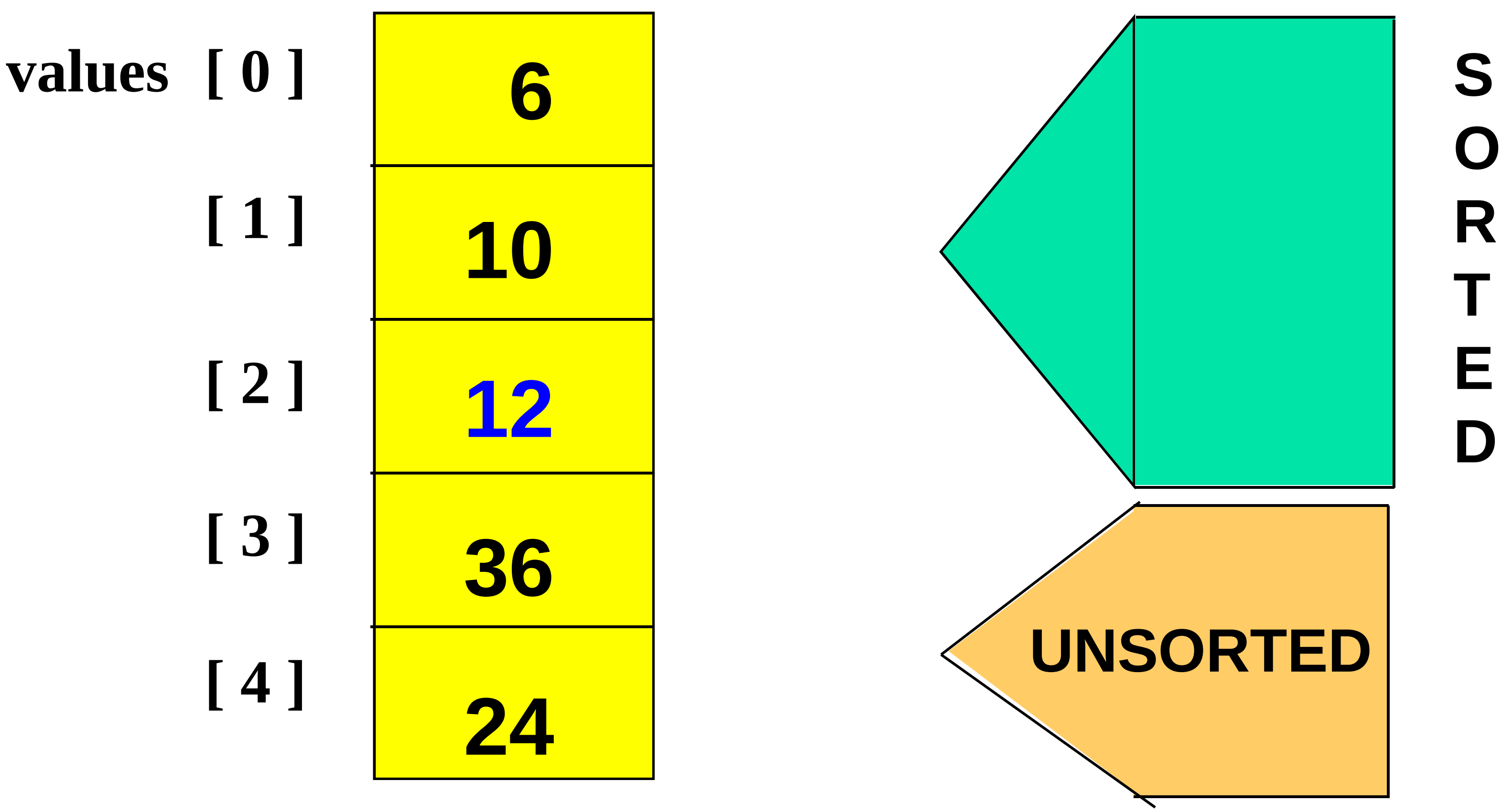
indexMin = 2

comp. 1: check if values[3] = 36 < values[indexMin] = 24 - NO

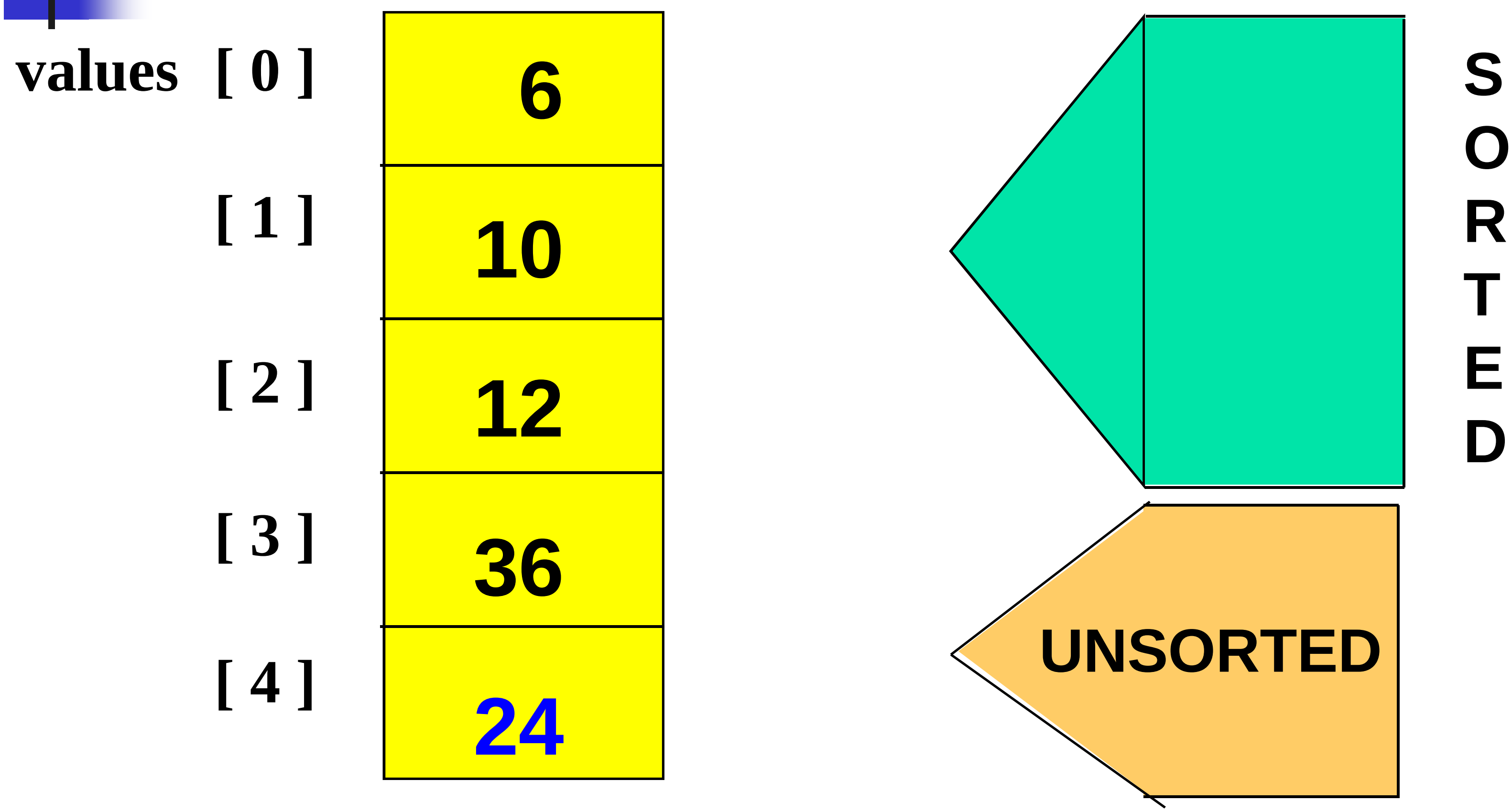
comp. 2: check if values[4] = 12 < values[indexMin] = 24 - yes => indexMin = 4

Thus indexMin = 4; swap values[2] = 24 with values[indexMin] = 12 – see next slide

Selection Sort: End of Pass Three



Selection Sort: Pass Four



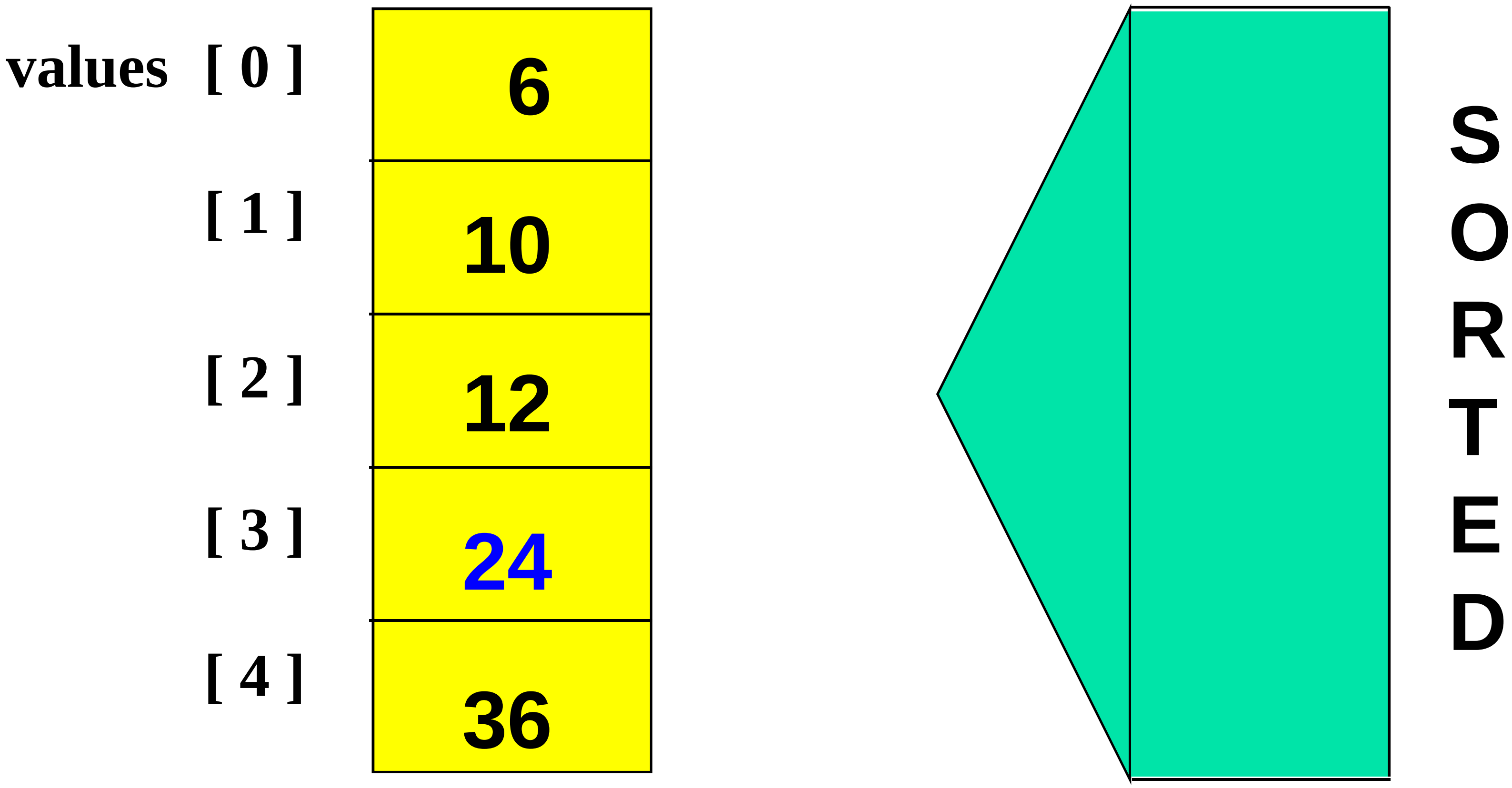
To find the smallest in UNSORTED:

indexMin = 3

comp. 1: check if values[4] = 24 < values[indexMin] = 36 - yes => indexMin = 4

Thus indexMin = 4; swap values[3] = 36 with values[indexMin] = 24 – see next slide

Selection Sort: End of Pass Four



Selection Sort: How Many Comparisons?

Values [0]	**6**
[1]	**10**
[2]	**12**
[3]	**24**
[4]	**36**

4 comparisons starting with indexMin = 0

3 comparisons starting with indexMin = 1

2 comparisons starting with indexMin = 2

1 comparison starting with indexMin = 3

0 comparisons starting with indexMin = 4

= 4 + 3 + 2 + 1 + 0 comparisons

In addition, we have <= 4 swaps

For Selection Sort in General

- Above: Array contained 5 elements

  4 + 3 + 2 + 1 + 0 comparisons and <= 4 swaps were needed

- Generalization for Selection Sort:

  When the array contains N elements, the number of comparisons is:

  $$(N-1) + (N-2) + \ldots + 2 + 1 + 0$$

  and the number of swaps is <= N-1

- Lets use:

  Sum = (N-1) + (N-2) + ... + 2 + 1 + 0

Calculating Number of Comparisons

$$
\begin{array}{rl}
\text{Sum} = & (N-1) + (N-2) + \ \ldots \ + \ 2 \ + \ 1 \\
+ \ \text{Sum} = & 1 \ + \ 2 \ + \ \ldots \ + (N-2) + (N-1) \\
\hline
= 2 * \text{Sum} = & N \ + \ N \ + \ \ldots \ + \ N \ + \ N = \\
= & N * (N-1)
\end{array}
$$

- Since:

$$2 * \text{Sum} = N * (N-1)$$

then:

$$\text{Sum} = 0.5\, N^2 - 0.5\, N$$

- This means that we have $0.5\, N^2 - 0.5\, N$ comparisons

And the Big-Oh for Selection Sort is…

- $0.5\ N^2 - 0.5\ N$ comparisons = $O(N^2)$ comparisons
  $N-1$ swaps = $O(N)$ swaps

- This means that complexity of Selection Sort is $O(N^2)$
  - Because $O(N^2) + O(N) = O(N^2)$

Pseudocode for Selection Sort

```
void  SelectionSort (int values[], int numValues)

// Post: Sorts array values[0 . . numValues-1 ]
// into ascending order by key value
{
  int  endIndex = numValues - 1 ;
  for (int current=0; current<endIndex; current++)
    Swap (values, current,
          MinIndex(values,current,endIndex));

}
```

Pseudocode for Selection Sort (contd)

```
int  MinIndex(int values[ ], int  start, int end)
//  Post: Function value = index of the smallest value
//  in values [start]  . . values [end].
{
  int  indexOfMin = start ;
  for(int index =start + 1 ; index <= end ; index++)
    if  (values[ index] < values [indexOfMin])
      indexOfMin = index ;
  return indexOfMin;
}
```