Algorithm

- Any well-defined computational procedure that takes some value, or set of values, as *input* and produces some value, or set of values, as *output*.

We use algorithms as a tool for solving a well-defined **computational problem**.

Example: the *sorting problem*.

- Input: Array $A[1 \dots n]$ of numbers.
- Output: Sorted array A. That is, the elements should be ordered such that

$$A[1] \le A[2] \le \cdots \le A[n].$$

Steps taken when designing and analyzing algorithms:

1. Define the computational problem to be solved.
   - What is the input and output of the problem?

2. Describe the algorithm.
   - Description of algorithm in English followed by pseudocode.

3. Proof of correctness.
   - Convince the reader of correctness.
   - Algorithm must be return the desired output for **any input**. Showing that the algorithm works for some specific example is **not** a proof of correctness of the algorithm.

4. Proof of runtime.
   - "Stopwatch" runtime depends heavily on things independent of the algorithm (e.g. hardware). We instead measure runtime in terms of the number of steps the algorithm takes.
   - Often interested in the **worst-case** runtime and space used (i.e. memory).

5. Proof of space (sometimes).
   - Prove bounds on the amount of memory the algorithm will need to produce the output.

We call a fixed input for a computational problem an **instance** of the problem.

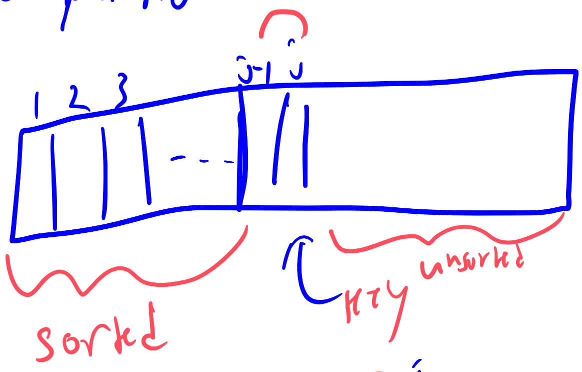
- For example, the input $[6, 2, 0, 1, 5, 2]$ is an instance of the sorting problem.

We say that an algorithm is **correct** if the algorithm terminates with the correct output for every instance of the problem.

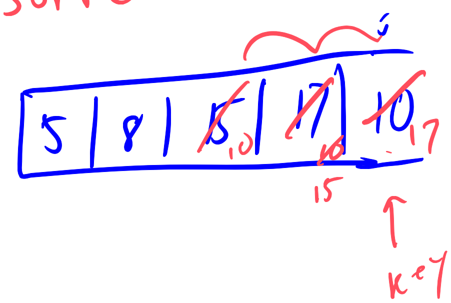
We say that a correct algorithm **solves** the given computational problem.

Example: Insertion Sort: Incremental Algorithm

Maintain a sorted portion at the beginning of the array and insert one number at a time into the sorted portion.



Ex:



We find the correct spot for "key" in sorted portion. This increases the size of the sorted portion by 1.

Pseudo-code:

		Steps	# of times executed
	InsertionSort(A[1 .. n])		
	{		
1.	for (j = 2; j ≤ n; j++)	c_1	n
	{		
2.	key = A[j]	c_2	$n-1$
3.	//Insert key into sorted order A[1 .. j-1]	0	
4.	i = j - 1	c_4	$n-1$
5.	while (i > 0 AND A[i] > key)	c_5	$\sum_{j=2}^{n} t_j$
	{		
6.	A[i+1] = A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7.	i --	c_7	$\sum_{j=2}^{n} (t_j - 1)$
	}		
8.	A[i+1] = key	c_8	$n-1$
	} //End for		
	}		

Proof of Correctness via Loop Invariant

Invariant: At the beginning of iteration j of the for loop: $A[1..j-1]$ consists of the numbers originally in $A[1..j-1]$ but sorted in non-decreasing order.

Base Case: $j=2$. $A[1..j-1] = A[1]$. One element is trivially sorted.

Inductive Step: Assume $A[1..j-1]$ satisfies the invariant, and we will show it is true for $A[1..j]$ after one iteration of the for loop.

- while loop shifts all elements larger than key one position to the right (so they are still sorted). After inserting key $A[1..j]$ satisfies the invariant.

Algorithm terminates when $j = n+1$. Therefore we have $A[1..j-1] = A[1..n]$ satisfies the invariant.

Thus the algorithm is correct.

Running Time

- Assume line i takes c_i time, where c_i is a constant (i.e., c_i is independent of the input)

- Count as a function of n,

running time on input of size n

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

Best Case: $t_j = 1 \quad \forall j$

$$T(n) = c_1 n + (c_2 + c_4 + c_5 + c_8)(n-1)$$
$$= c_1 n + c_2 n + c_4 n + c_5 n + c_8 n - c_2 - c_4 - c_5 - c_8$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8) n + (-c_2 - c_4 - c_5 - c_8)$$
$$= a n + b$$

constant constant

linear function.

$(n)' \to (2n)'$

$n \to 2n$

Worst Case: $t_j = j \quad \forall j$

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = 2 + 3 + 4 + 5 + \cdots + n = \frac{(n+1)n}{2} - 1$$
$$= \frac{n^2 + n}{2} - 1$$
$$= \frac{1}{2} n^2 + \frac{1}{2} n - 1$$

Quadratic

$(n)^2 \to (2n)^2$

$n^2 \to 4n^2$