ICPC Lecture Series
Algorithm 1

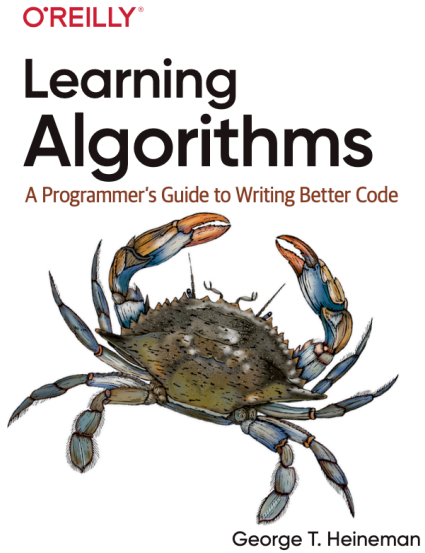
Dr. Samuel Cho, Ph.D.

NKU ASE/CS

The Book

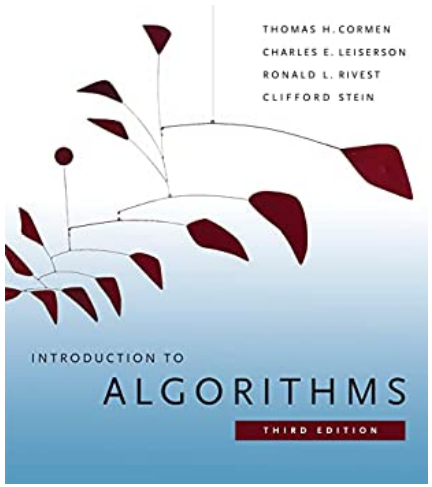
The Algorithm Book

- We will use George Heineman's Learning Algorithms book as a main reference.
- This book is an introduction-level algorithm book that uses Python.



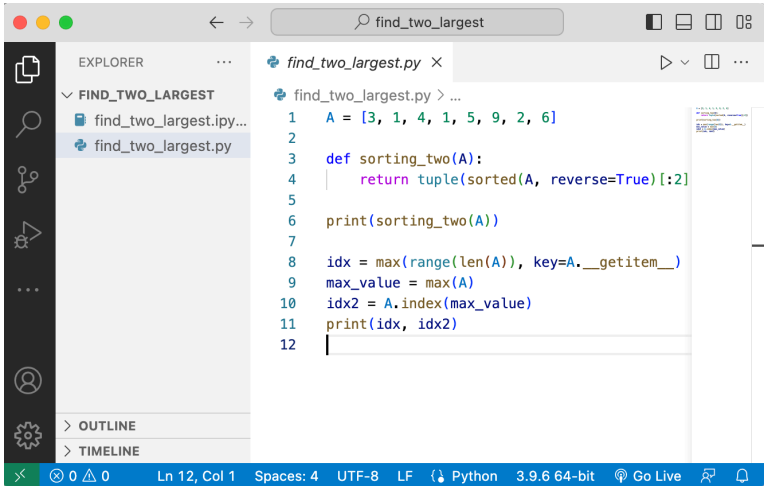
The Algorithm Book - Bible

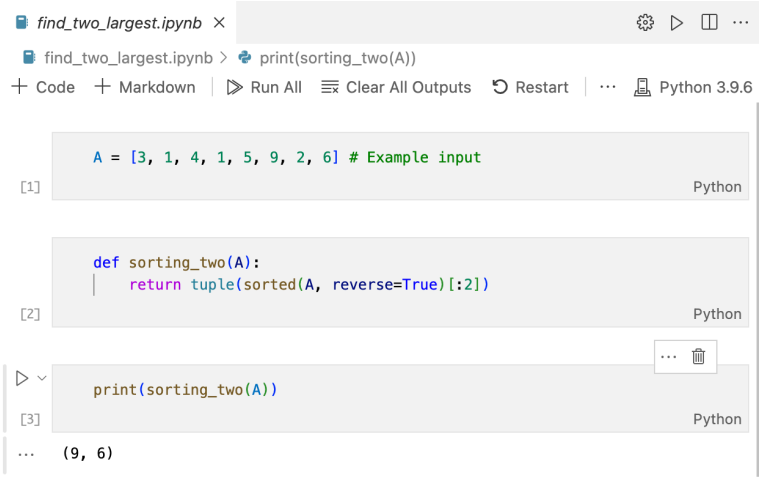
- The most famous algorithm book is Corman et al.'s Introduction to Algorithms.
- This book is an encyclopedia of algorithms, so you may need this book for learning various algorithms and problem-solving tools.



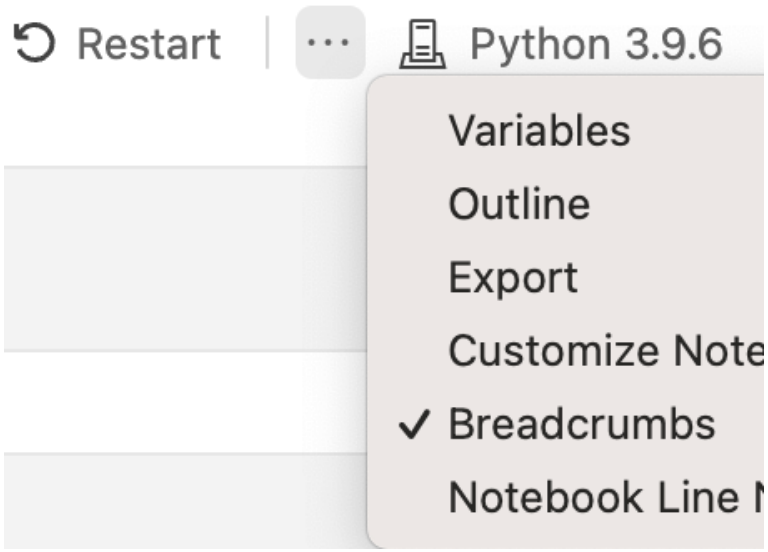
Run/Debug Code

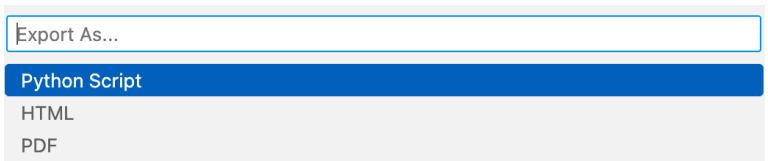
- Use Visual Code Studio (VCS) and extensions (Python and Code Runner).
- Consider using the Jupyter Notebook for documentation and learning code.



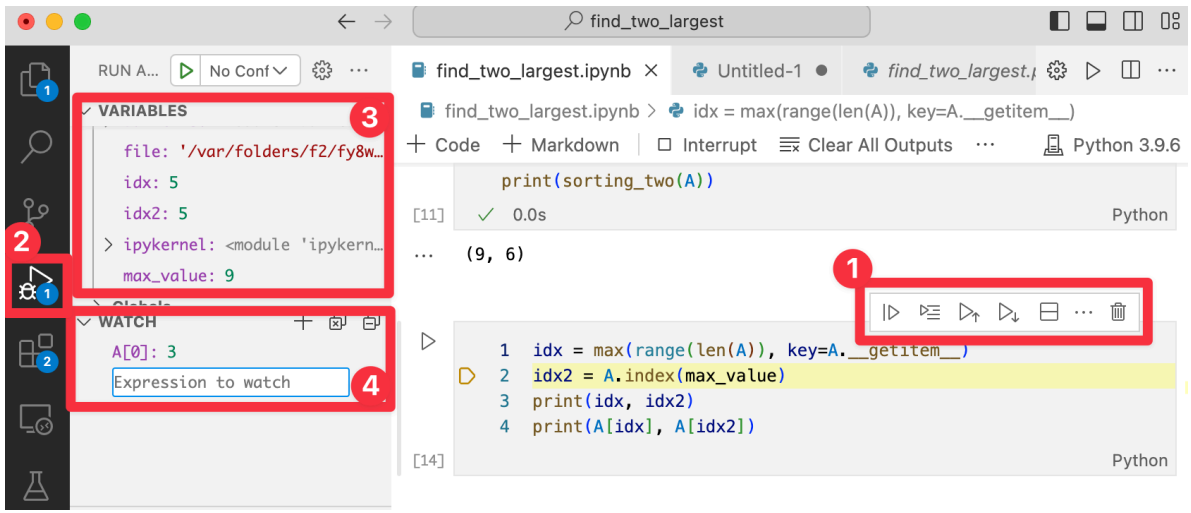


- You can use the export Python code from Jupyter, so using Jupyter to practice algorithms is a good idea.
- Google or ask Chat GPT with the keyword "install jupyter plugin vscode."





- You can debug Python code in Jupyter.
- Click ① to run the code, click ② to open the debug window, ③ shows the variables in the program, and ④ shows we can give expression to get the results.



Problem-Solving in Algorithm

How Bill Gates Solve Problems

- *"I like to think the whole program through at a design level before I sit down and write any of the code."*
- *"Before I sit down to code something, most of the instructions have already run through my head."*
- *"And if there is a bug in the code, I feel pretty bad because if there's one bug, it says your mental simulation is imperfect, and there might be thousands of bugs in the program."*
- *"I really hate it when I watch some people's programs (with bugs), and I don't see them thinking."*

How We Apply His Trick

Only fools rush in; we solve the problem step by step.

- Step 1: Understand the Problem.
- Step 2: Think and Have a Sketch of the Solution.
- Step 3: Find the Data Structure.
- Step 4: Come up with the (Pseudo) Algorithm.
- Step 5: Write code.
- Step 6: Check, Debug, or Rewrite/Revise.

In other words:

- You should visualize the problem and solution.
- You should design the algorithm before coding.
- You should solve the problems multiple times (especially when the problem is a hard one).
- So, it's OK to solve the problem using the brutal force algorithm, as you can improve the first solution iteratively.

Finding the Largest Value in a List

Step 1: Understand the Problem

When given a problem, make sure you understand the problem domain.

What is an input?: A given list.

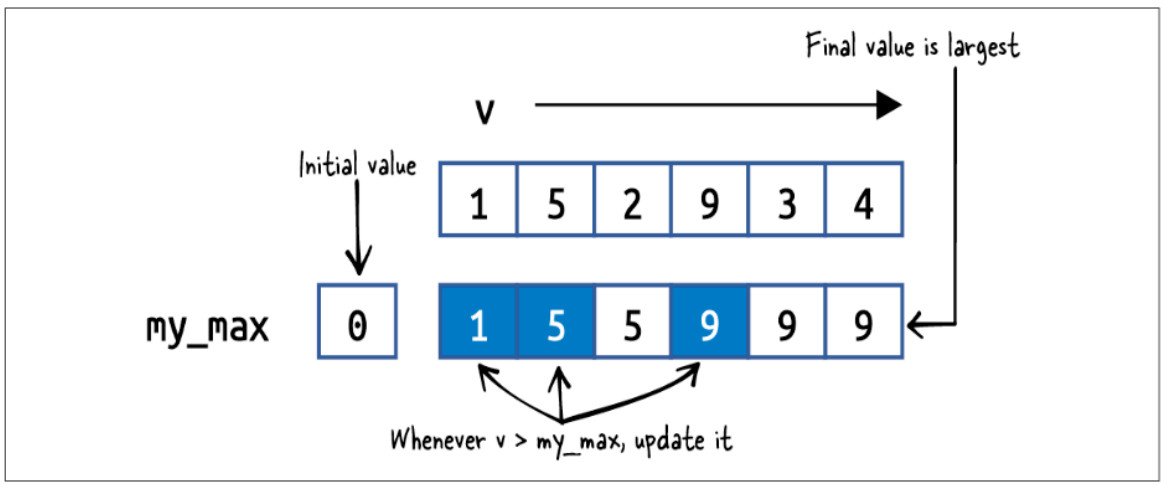
What is an output?: The largest value in the input list.

What is an example?: With an input `[1,5,4]`, the return value is 5 (index 1).

What are constraints?: The original list should not be modified.

Step 2: Think and Have a Sketch of the Solution

- When you can draw diagrams on paper using a pen, it means you know how to solve the problem.
- Use concrete and simple examples—you know the input and output—to verify that your solution works.



Step 3: Find the Data Structure

- We don't need a new data structure to solve this problem.

Step 4: Come up with the (Pseudo) Algorithm

- Python is a high-level language you can use to write pseudo code.
- You write the pseudo-code using a pen and paper.

Step 5: Write code

Code #

```
1 def max_ver1(A):
2     my_max = 0
3     for v in A:
4         if my_max < v:
5             my_max = V
6     return my_max
```

Check, Debug, or Rewrite/Revise

- Even though this code works with the sample example, this code has a bug.
- The initial value setup in line 2 is wrong; what if all the inputs are negative numbers?
- We need to fix this bug and rewrite the code.

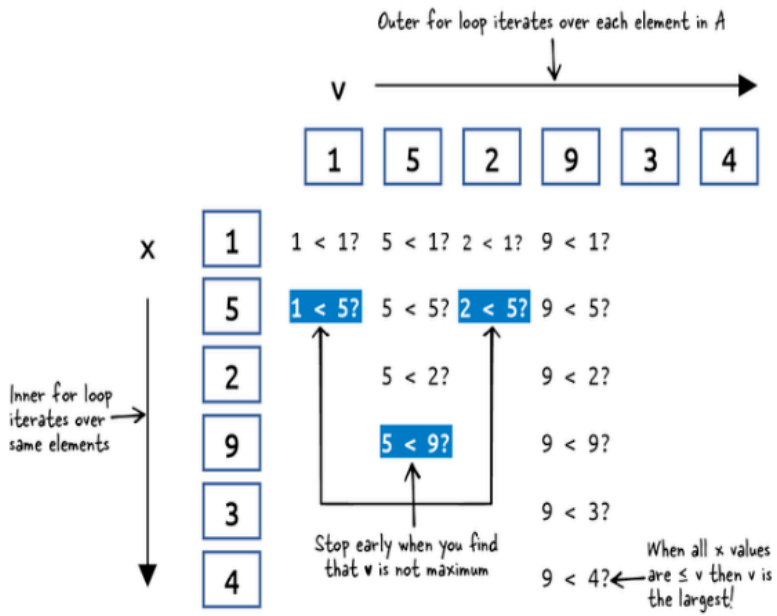
- We use the first element as the initial value (ver 2).
- Unless we need an index, we can use ver 1 to simplify the implementation (ver 3).
- Even though `A[0]` is compared with itself, it doesn't affect the performance.

Code #

```
1 def max_ver2(A):
2   my_max = A[0]
3   for i in range(1, len(A)): if my_max < A[i]: my_max = A[i]
4   return my_max
5
6 def max_ver3(A):
7   my_max = A[0]
8   for v in A: if my_max < v: my_max = V
9   return my_max
```

Correct, but Inefficient Solution

- What if we came up with the algorithm that checks the value in an array (v) has any other values (x) that are larger ($v < x$)?
- Novice programmers can make this kind of mistake.

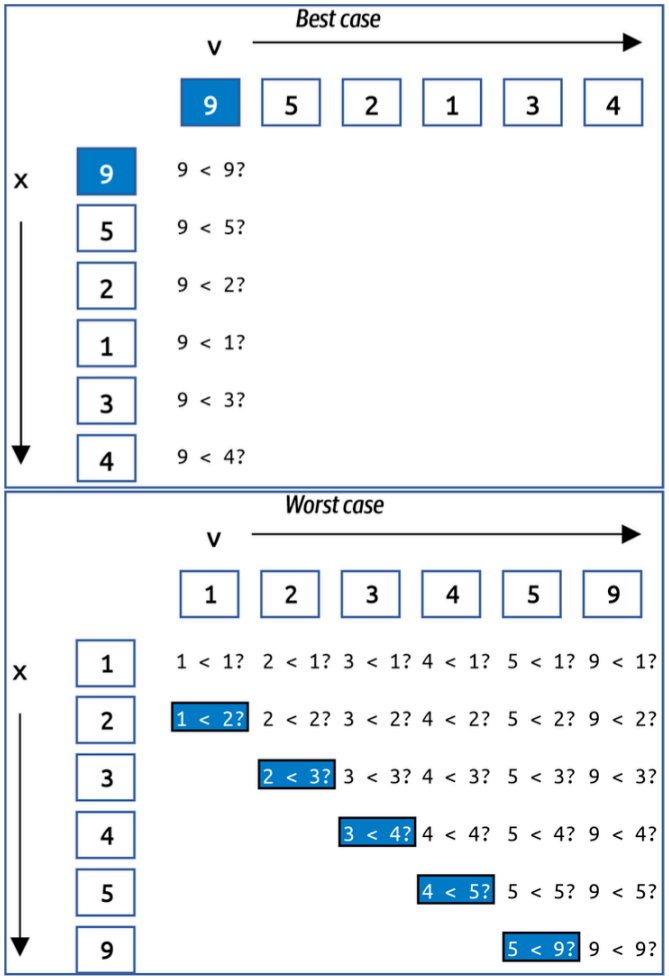


- We can use a boolean variable (v_is_largest) to check all the values are smaller than v (lines 4 – 6).

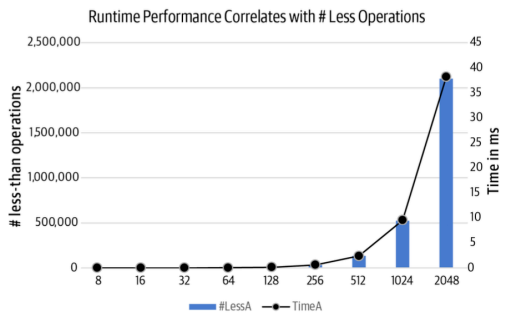
Code #

```
1 def max_ver4(A):
2   for v in A:
3     v_is_largest = True
4     for x in A:
5       if v < x:
6         v_is_largest = False; break
7     if v_is_largest:
8       return v
9   return None
```

- However, this is not only unnecessarily complex but also inefficient in performance due to its nested loop.
- In the worst case, the time to find the solution increases dramatically.



- This table shows the number of operations (and the time to execute the algorithm) when the input size increases.
- We can see that a bad algorithm results in (a) unnecessary complexity and (b) worse performance.
- So, we should revise the algorithm to improve iteratively whenever possible.



Finding Two Largest Values in a List

Problem

We are given a list A. Find the two largest values in a list.

What is an input?: A given list.

What is an output?: Two largest values in the input list.

What is an example?: With an input `[1,5,4]`, the return values are 5 and 4 (index 1 and 2).

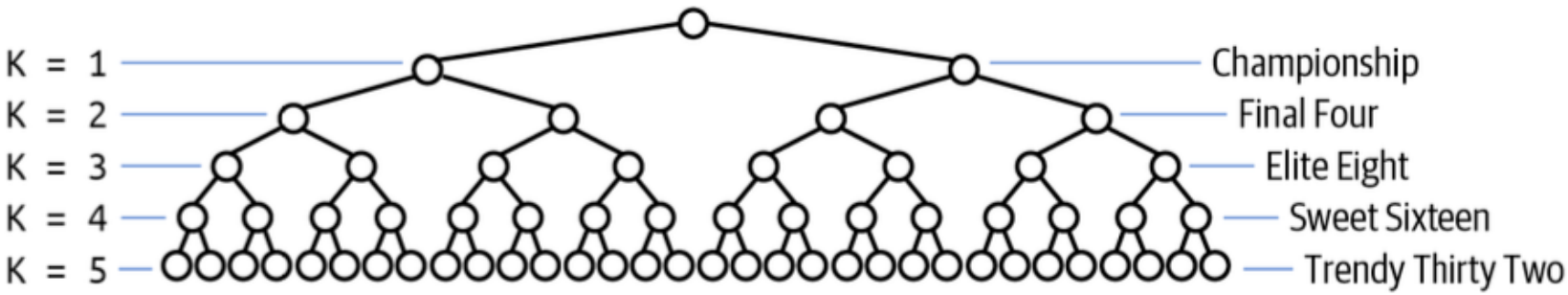
What are constraints?: The original list should not be modified.

Sketch of the Solutions

We can come up with multiple solutions.

1. **sorting_two:** Sort the list (descending) and select the first two elements.
2. **double_two:** Copy the list, find the max, remove the max value in a copied list, and find the max in the copied list.
3. **mutable_two:** Find the index of the max value and get the max value, remove the max, find the (2nd) max from the (modified) list, and insert the 1st max back into the array.

4. **largest_two:** Reuse the existing largest approach; select the first two values and make the first element the larger of the two, and then loop from 2 and replace the first or second element.
5. **tournament_two:** Make a single-elimination tournament to find the final winners.



Solution 1 (sorting_two)

Sort the list (descending) and select the first two elements.

- We need to understand OOP and FP approaches to write code; in this code, we use Python's sorted function that uses the FP approach.
- We can use `A.sort()` that uses the OOP approach, but in this case, A is modified to violate the requirements.

Code #

```
1 def sorting_two(A):
2   return tuple(sorted(A, reverse=True)[:2]) # only 0, 1
```

Solution 2 (double_two)

Copy the list, find the max, remove the max value in a copied list, and find the max in the copied list.

- We need to be sure that the original input should not be modified by copying the list (line 3).

Code #

```
1 def double_two(A):
2   my_max = max(A)
3   copy = list(A)
4   copy.remove(my_max)
5   return (my_max, max(copy))
```

Solution 3 (mutable_two)

Find the index of the max value and get the max value, remove the max, find the (2nd) max from the (modified) list, and insert the 1st max back into the array.

- We need to restore the original list by inserting the deleted element back (line 6).

Code #

```
1 def mutable_two(A):
2     idx = max(range(len(A)), key=A.__getitem__)
3     my_max = A[idx]
4     del A[idx]
5     second = max(A)
6     A.insert(idx, my_max)
7     return (my_max, second)
```

- Line 1 is a Pythonic way to find the index that; it generates a range `[0, 1, ..., 7]`, and uses the key (`A.__getitem__`) to get the value that matches the range to find the max value.
- However, this is for advanced users; we can use a simple approach to get the same results (lines 2 – 3).

Code #

```
1 idx = max(range(len(A)), key=A.__getitem__)
2 max_value = max(A)
3 idx = A.index(max_value)
```

Solution 4 (largest_two)

Reuse the existing largest approach; select the first two values and make the first element the larger of the two, and then loop from 2 and replace the first or second element.

Code #

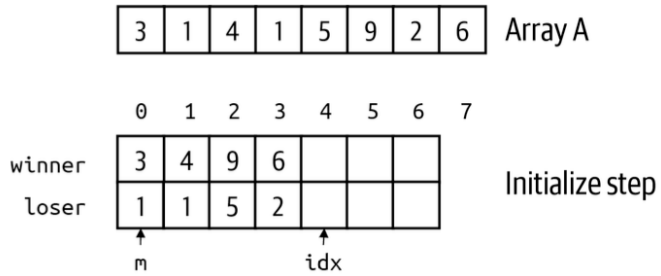
```
1  def largest_two(A):
2    my_max,second = A[:2]
3    if my_max < second:
4      my_max,second = second,my_max
5    for idx in range(2, len(A)):
6      if my_max < A[idx]:
7        my_max,second = A[idx],my_max
8      elif second < A[idx]:
9        second = A[idx]
10   return (my_max, second)
```

Solution 5 (tournament_two)

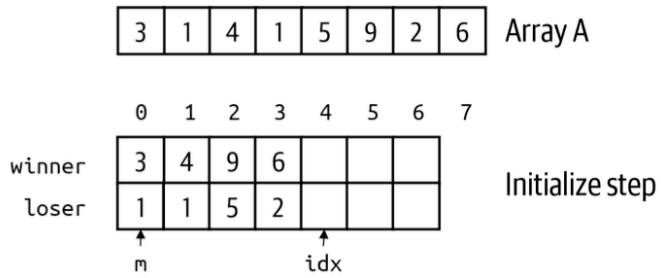
Make a single-elimination tournament to find the final winners.

- Think about the data structure and algorithm to use this idea.

Solution 4 - Data Structure and Algorithm

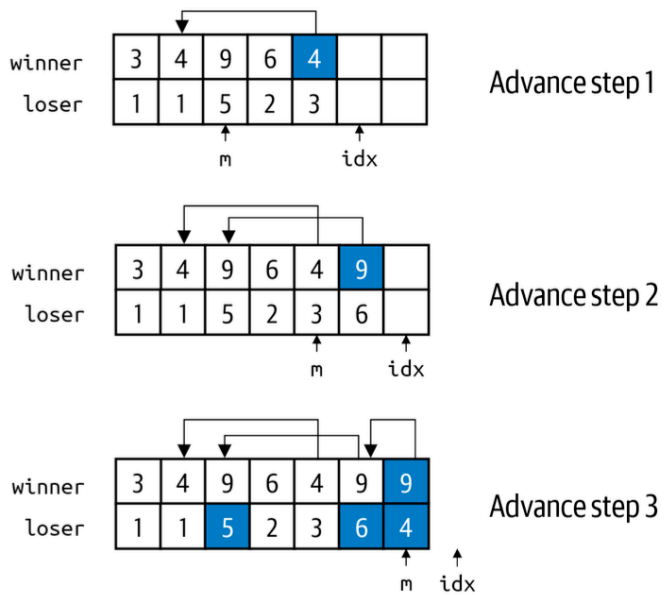


- Given an input array A, we can use two arrays — winner and loser.
- The size of the array is $\frac{1}{2}len(A) + \frac{1}{4}len(A) + \ldots + 1$; In this example, $4 + 2 + 1 = 7$; We can prove that it is always $len(A) - 1$ from $\sum_{k=0}^{k=n} 2^k = 2^n - 1$.
- Initially, we select two values from the array and find the winner and lower to store them into corresponding arrays.



- We need a variable 'idx' to keep track of the location in the winner array to store values; it starts at $\frac{1}{2}len(A)$, is increased by 1, and stops at $len(A)$.
- We need a variable 'm' to keep track of the values in the winner list; 'm' starts at the index 0, is increased by 2, and stops at $idx-1$.

- We also need to add one more process to get the 2nd winner.
- As this example shows, the 2nd winner can be in the loser list.
- To find the 2nd winner quickly, we need to keep track of the prior list that points to the previous m position.



- Write code in Python.

Time Complexity

- In this example, we implement the Python methods to add values to the variable ct.

```python
def f0(N):
  ct = 0
  ct = ct + 1
  ct = ct + 1
  return ct
```

```python
def f1(N):
  ct = 0
  for i in range(N):
    ct = ct + 1
  return ct
```

```python
def f2(N):
  ct = 0
  for i in range(N):
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
    ct = ct + 1
  return ct
```

```python
def f3(N):
  ct = 0
  for i in range(N):
    for j in range(N):
      ct = ct + 1
  return ct
```

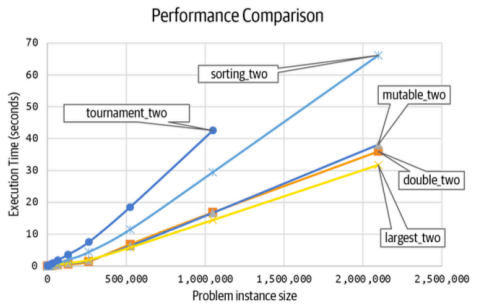
- The performance shows that f0 is not impacted by the input size, but f3 has a double loop to be dramatically impacted by the input size.
- There are no big differences between f1 and f2.

N	f0	f1	f2	f3
512	2	512	3,584	262,144
1,024	2	1,024	7,168	1,048,576
2,048	2	2,048	14,336	4,194,304

Time Complexity (Performance) of Algorithms

- As we expected, the tournament_two is the worst due to its loops, and as sorting is expensive, its performance is not that good.
- All the other algorithms show similar performance.

N	double_two	mutable_two	largest_two	sorting_two	tournament_two
1,024	0.00	0.01	0.01	0.01	0.03
2,048	0.01	0.01	0.01	0.02	0.05
4,096	0.01	0.02	0.03	0.03	0.10
8,192	0.03	0.05	0.05	0.08	0.21
16,384	0.06	0.09	0.11	0.18	0.43
32,768	0.12	0.20	0.22	0.40	0.90
65,536	0.30	0.39	0.44	0.89	1.79
131,072	0.55	0.81	0.91	1.94	3.59
262,144	1.42	1.76	1.93	4.36	7.51
524,288	6.79	6.29	5.82	11.44	18.49
1,048,576	16.82	16.69	14.43	29.45	42.55
2,097,152	35.96	38.10	31.71	66.14	…



Space Complexity of Algorithms

we also have to consider space complexity, which accounts for extra memory required by an algorithm based on the size, N, of a problem instance.

- largest_two() has minimal space requirements: it uses two variables, my_max and second, and an iterator variable, idx.
- No matter the size of the problem instance, its extra space never changes.
- This means the space complexity is independent of the size of the problem instance or constant;
- mutable_two() has similar behavior.

- In contrast, tournament_two() allocated three arrays winner, loser, and prior all of the size $N-1$.
- As N increases, the total extra storage increases in a manner directly proportional to the size of the problem instance.
- Both double_two() and sorting_two() make a copy of the input, A, which means their storage usage is much more like tournament_two() than largest_two().