2023-01-13

7.6.1
Selection Sort

Sorting Algorithms

On Wednesday, we introduced **binary search**, a search algorithm that only works on sorted arrays.

So, we need to be able to sort arrays. How? With a sorting algorithm!

In Unit 7, we discuss two sorting algorithms: **selection sort** and **insertion sort**.

These algorithms are relatively simple... but not very efficient! We will learn a much more efficient algorithm, **merge sort**, in Unit 9.

Selection Sort

- Today, we will just discuss selection sort and you'll implement your own.
- We'll cover **insertion sort** next week.

Selection Sort

- Selection Sort is an algorithm that relies on **swapping** array elements.
- We used swapping in section 6.4 to reverse the elements in an array.
- Java doesn't have a built-in "swap" capability, so you must use a temporary variable:

```
// Swap array elements array[i] and array[j]
int temp = array[i];
array[i] = array[j];
array[j] = temp;
```

Swapping variables is also called exchanging variables. Sort algorithms that rely on swapping/exchanging values are classified as **exchange sorts**. (Confusingly, there is also a specific sort algorithm called Exchange Sort.)

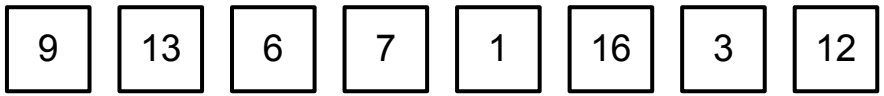
Selection Sort

For each index i in the array to be sorted:

- Find the index j of the minimum value in the range array[i] to array[length-1]
- Swap array[i] and array[j] (this is a "no-op" if i == j)

Selection Sort

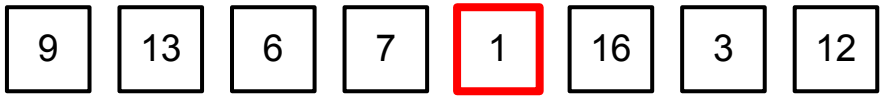
Let's work through a Selection Sort. We'll start with this unsorted array.



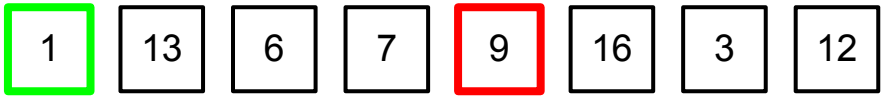
Selection Sort

Loop Iteration #1

Step 1. Identify minimum value



Step 2. Swap it into position 0

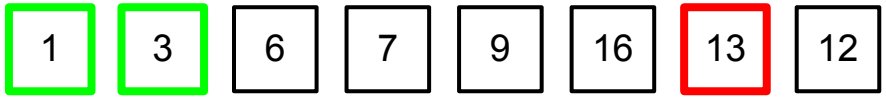


Selection Sort

Loop Iteration #2

Step 1. Identify minimum value in array [1..N-1]

Step 2. Swap it into position 1



Selection Sort

Loop Iteration #3

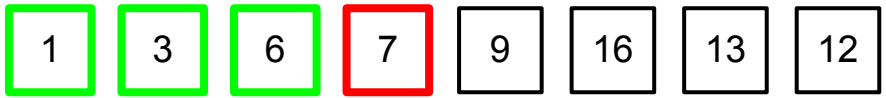
Step 1. Identify minimum value in array [2..N-1]

Step 2. Swap it into position 2

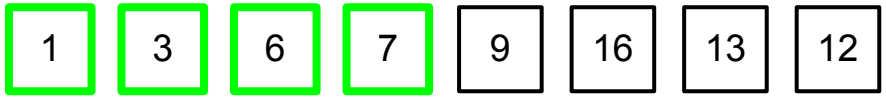
Selection Sort

Loop Iteration #4

Step 1. Identify minimum value in array [3..N-1]



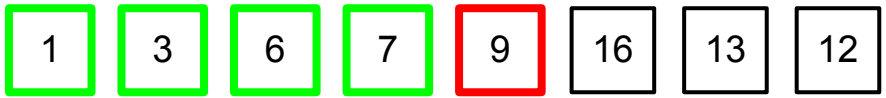
Step 2. Swap it into position 3



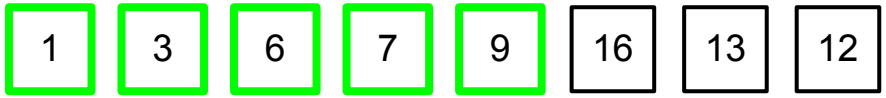
Selection Sort

Loop Iteration #5

Step 1. Identify minimum value in array [4..N-1]



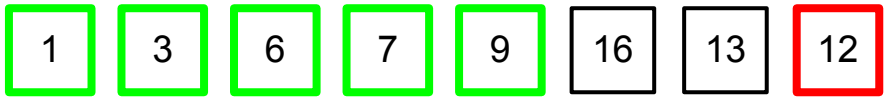
Step 2. Swap it into position 4



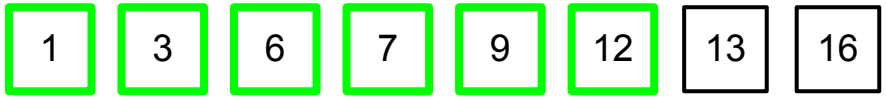
Selection Sort

Loop Iteration #6

Step 1. Identify minimum value in array [5..N-1]



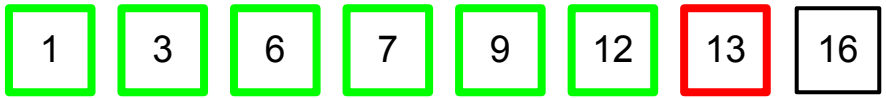
Step 2. Swap it into position 5



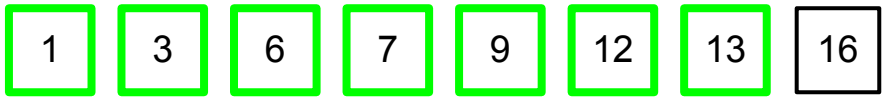
Selection Sort

Loop Iteration #7

Step 1. Identify minimum value in array [6..N-1]



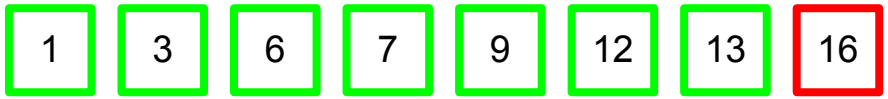
Step 2. Swap it into position 6



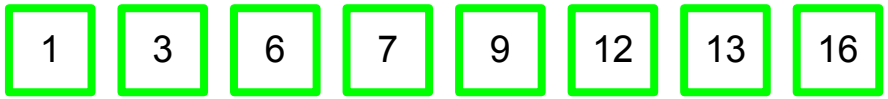
Selection Sort

Loop Iteration #8

Step 1. Identify minimum value in array [7..N-1]



Step 2. Swap it into position 7



So... Is Selection Sort... Good?

Selection Sort – an intuitive but slow algorithm

How many comparisons does Selection Sort do?

It examines N elements on the first pass,
N-1 elements on the second pass,
N-2 on your third… in other words,

$$1 + 2 + 3 + \ldots + N = \sum_{i=1}^{N} i$$

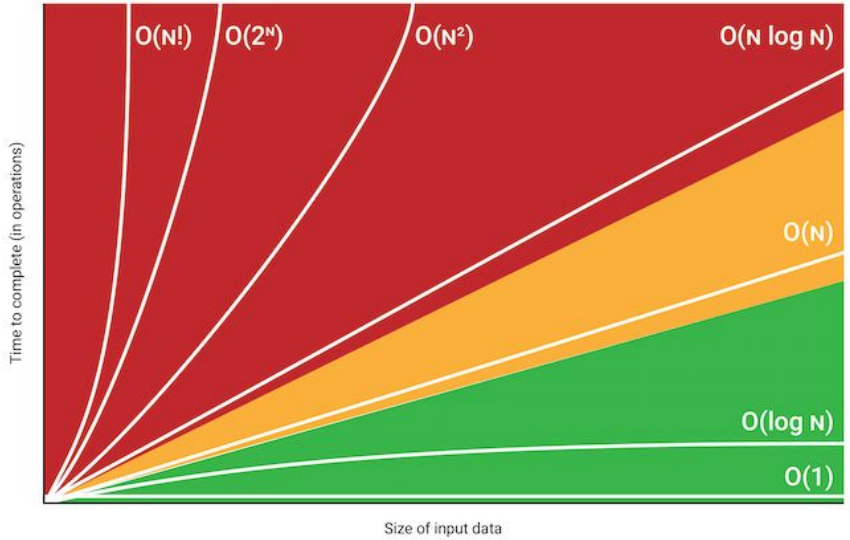
Loop #	Comparisons (N=8)
1	8
2	7
3	6
4	5
4	4
5	3
6	2
7	1

Selection Sort – our old friend $O(N^2)$

$$1 + 2 + 3 + \ldots + N = \sum_{i=1}^{N} i$$

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

$$\frac{N(N+1)}{2} = \frac{N^2 + N}{2} \approx \mathbf{O(N^2)}$$



Selection Sort – an intuitive but slow algorithm

$O(N^2)$ is also known as **quadratic time**. Many simple sort algorithms are quadratic time.

It can be OK for a small number of elements, but as N gets big, the algorithm's running time becomes very long.

In Unit 10, we will learn a sorting algorithm, **merge sort**, that has much better running time: O(N log N). But it's also more complicated to code!

N	N^2
1	1
10	100
100	10,000
1000	1,000,000
10000	100,000,000
100000	10,000,000,000
1000000	10^{12}

Flashback: Unit 4.3

In computer science, the computational complexity or simply complexity of an algorithm is the amount of resources required to run it.

Time complexity is the amount of time that an algorithm takes to run, but in an abstract sense of how many operations need to be executed, not a quantity of seconds or milliseconds. The usual units of time (seconds, minutes etc.) are not used because they are too dependent on the choice of a specific computer and on the evolution of technology.

Space complexity is how much memory an algorithm consumes. A lot of our algorithms so far allocate no memory, or allocate one or two things. This is constant space complexity. Constant space complexity means that the amount of space that your algorithm uses is independent of the input parameters. Linear space complexity means you probably allocated an array of size N, or for, quadratic, N^2, etc.

What is the space complexity of Selection Sort?

First sorting algorithm I learned: Bubble Sort

Pseudocode implementation [edit]

In pseudocode the algorithm can be expressed as (0-based array):

```
procedure bubbleSort(A : list of sortable items)
    n := length(A)
    repeat
        swapped := false
        for i := 1 to n-1 inclusive do
            /* if this pair is out of order */
            if A[i-1] > A[i] then
                /* swap them and remember something changed */
                swap(A[i-1], A[i])
                swapped := true
            end if
        end for
    until not swapped
end procedure
```

Also $O(N^2)$. Mostly of historical interest... although bubble sort has some properties that make it useful in some use cases, like computer graphics.

Selection sort

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ comparisons, $O(n)$ swaps
Best-case performance	$O(n^2)$ comparisons, $O(1)$ swap
Average performance	$O(n^2)$ comparisons, $O(n)$ swaps
Worst-case space complexity	$O(1)$ auxiliary

The Jargon File, which famously calls bogosort "the archetypical [sic] perversely awful algorithm", also calls bubble sort "the generic bad algorithm".[5] Donald Knuth, in *The Art of Computer Programming*, concluded that "the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems", some of which he then discusses.[6]

Bubble sort

Static visualization of bubble sort[1]

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ comparisons, $O(n^2)$ swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons, $O(n^2)$ swaps
Worst-case space complexity	$O(n)$ total, $O(1)$ auxiliary

Bogosort

From Wikipedia, the free encyclopedia

In computer science, **bogosort**[1][2] (also known as **permutation sort**, **stupid sort**,[3] slowsort or bozosort) is a sorting algorithm based on the generate and test paradigm. The function successively generates permutations of its input until it finds one that is sorted. It is not considered useful for sorting, but may be used for educational purposes, to contrast it with more efficient algorithms.

Description of the algorithm [edit]

The following is a description of the randomized algorithm in pseudocode:

```
while not sorted(deck):
    shuffle(deck)
```

Bogobogosort

is an algorithm that was designed not to succeed before the heat death of the universe on any sizable list. It works by recursively calling itself with smaller and smaller copies of the beginning of the list to see if they are sorted. The base case is a single element, which is always sorted. For other cases, it compares the last element to the maximum element from the previous elements in the list. If the last element is greater or equal, it checks if the order of the copy matches the previous version, and if so returns. Otherwise, it reshuffles the current copy of the list and restarts its recursive check.[7]

Exercise: Build your own selection sort (not bogosort)

- Replit: SelectionSort
- We'll reconvene near the end of class to review the solution together.

```java
private void selectionSort(ArrayList<Record> records) {
  for (int i = 0, n = records.size(); i < n; i++) {
    int jMin = i;
    String companyMin = records.get(i).getCompany();
    for (int j = i + 1; j < n; j++) {
      String company = records.get(j).getCompany();
      if (company.compareToIgnoreCase(companyMin) < 0) {
        jMin = j;
        companyMin = company;
      }
    }
    if (jMin != i) {
      Record temp = records.get(jMin);
      records.set(jMin, records.get(i));
      records.set(i, temp);
    }
  }
}
```

```java
public static int smallestValIdx(ArrayList<Record> records, int startIdx) {
  int smallestValIdx = -1;
  for (int idx = startIdx; idx < records.size(); idx++) {
    if (-1 == smallestValIdx) {
      smallestValIdx = idx;
    } else {
      if (records.get(idx).getCompany().compareToIgnoreCase(records.get(smallestValIdx).getCompany()) < 0) {
        smallestValIdx = idx;
      }
    }
  }
  return smallestValIdx;
}

private void selectionSort(ArrayList<Record> records) {
  /* No need to do the last element in the ArrayList */
  for (int idx = 0; idx < records.size() - 1; idx++) {
    int smallestValIdx = smallestValIdx(records, idx);
    if (-1 != smallestValIdx) {
      Record swapVal = records.get(idx);
      records.set(idx, records.get(smallestValIdx));
      records.set(smallestValIdx, swapVal);
    }
  }
}
```