Unit 24: Sorting

visualgo.net

Learning Objectives

After this unit, students should:

- be familiar with four sorting algorithms: counting sort, selection sort, bubble sort, and insertion sort
- be able to implement the above algorithms, argue that they are correct, and analyzing their running time
- be aware of the difference between comparison-based sorting algorithms and counting sort
- be aware of the situations where insertion sort performs better than selection sort and bubble sort.
- be aware of the situations where counting sort performs better than other sorting algorithms.

Sorting

most of the sorting are comparison based
- need to compare between 2 numbers to determine the allocation

Sorting is one of the most fundamental computational problems. Given a list of items, we want to rearrange the items in some order. We have seen in the previous unit that searching in a sorted list gives us tremendous improvement in efficiency.

In this unit, we assume that the items we wish to rearrange are integers, and we wish to rearrange them in increasing order. We could, however, in practice, sort any type of items.

We have seen two algorithms to sort in this module: counting sort and selection sort. We are going to see two more today.

Counting Sort

counting sort is not an (comparison based sorting algo)
- because of the assumption

Let's revisit counting sort. We are given an array of numbers, guaranteed to fall between a given range, say 0 to MAX. We can scan through this input array and count how many times each number appears in the input. We use another array of size $MAX + 1$ (let's call this the frequency array) to keep track of the counts. Once we are done, we loop through the frequency array and put the elements back into the output.

Here is the code that implements the counting sort algorithm.

```
/**
 * Perform counting sort on the input in[] and store the sorted
 * numbers in out[].
 *
 * @param[in] in The array containing numbers to be sorted.
 * @param[out] out The array containing the sorted numbers.
 * @param[in] len The size of the input and output array.
 *
 * @pre in[i] is between 0 and MAX for all i.
 * @post out[] is sorted
 */
void counting_sort(const long in[], long out[], long len)
{
  long freq[MAX + 1] = { 0 };

  for (long i = 0; i < len; i += 1) {
    freq[in[i]] += 1;
  }

  long outpos = 0;
  for (long i = 0; i <= MAX; i += 1) {
    for (long j = outpos; j < outpos + freq[i]; j += 1) {
      out[j] = i;
    }
    outpos += freq[i];
  }
}
```

Let's consider the running time of counting sort. We will break it down into three steps of counting sort.

First, initializing `freq` array to 0 takes $O(MAX)$ time.

Second, looping through `in` and counting takes $O(n)$ time (where n is `len`).

What about the third step: populating the output array with sorted numbers? This step involves a double for loop. You shouldn't jump to the conclusion that it takes $O(MAX^2)$ or $O(n^2)$. Let's analyze this more carefully.

We will go with a more intuitive/informal approach first. What we do here is to store the sorted numbers into the output, there are n numbers, so it seems reasonable to assume that this step takes $O(n)$ time.

Let's verify now with a more formal and systematic way. The inner loop:

```
    for (long j = outpos; j < outpos + freq[i]; j += 1) {
      out[j] = i;
    }
```

takes $O(f_i)$ time, where f_i is the `freq[i]` the number of times i appears. The outer loop loops through this for different i, from $i = 0, .. MAX$. So the total number of times is:

$$\sum_{i=0}^{MAX} f_i$$

This corresponds to the total number of times each number appears in the input, which is, well, just n. So indeed the third step takes $O(n)$.

The runnning time for counting sort is thus $O(n + n + MAX)$, which is just $O(n + MAX)$.

Note that since we do not know the relationship between n and MAX, we cannot simplify this term to either $O(n)$ or $O(MAX)$.

although max is constant, but because not determined then cannot take away

Selection Sort

Recall that selection sort repeatedly find the maximum element and move it to the back of the array (assuming we are sorting in increasing order). Here is an implementation of selection sort.

```
1  /**
2   * Find the index of the largest element among list[0..last].
3   *
4   * @param[in] last The last element to search.
5   * @param[in] list Input list
6   * @return The index of the max element among list[0..last].
7   *         Breaking ties by choosing the smaller index.
8   * @pre list is not NULL and list[0] .. list[last] are valid.
9   */
10 long max(long last, const long list[])
11 {
12   long max_so_far = list[0];
13   long max_index = 0;
14   for (long i = 1; i <= last; i += 1) {
15     if (list[i] > max_so_far) {
16       max_so_far = list[i];
17       max_index = i;
18     }
19   }
20   return max_index;
21 }
22
23 /**
24  * Sort a list using selection sort.
25  *
26  * @param[in] length The size of the list to sort.
27  * @param[in] list The input list
```

inner loop

```
 * @pre list is not NULL and list[0]..list[length-1] are valid
 * @post The list is sorted.
 */
void selection_sort(long length, long list[])
{
  for (long i = 1; i < length; i += 1) {
    long max_pos = max(length - i, list);
    if (max_pos != length - i) {
      swap(&list[max_pos], &list[length - i]);
    }
  }
}
```

What is the running time for selection sort? Let n is the length of the input list (i.e., `length`). The for loop iterates through $O(n)$ times.
In each loop, it finds the index of the maximum element among list[0] to list[length - i], and then perform a swap to put the maximum element into the rightful position. Finding the index of the maximum element takes more time than swapping, so let's focus on that (since for Big-O notation we are only concern with the higher-order terms).

The `max` function has another for loop in it. And it loops through the list $O(n-i)$ times. The running time is thus:

$$\sum_{i=0}^{n}(n-i) = n^2 - \frac{n(n+1)}{2} = \frac{n^2-n}{2} = O(n^2)$$

So selection sort takes O(n^2) time.

Selection vs Counting Sort

Comparing the running of counting sort $O(n+MAX)$ vs selection sort $O(n^2)$, it is clear that counting sort is more efficient -- we say that counting sort is a *linear time* algorithm and selection sort is a *quadratic time* algorithm.

What is the magic of counting sort? Why don't we just use counting sort all the time and why bother learning about other sorting algorithms?

It turns out that counting sort is special because it has an assumption that the input numbers fall into a certain range. If MAX is small, then counting sort is efficient. If MAX is say, the maximum long values, $2^{63}-1$, then counting sort is not necessarily more efficient (both in terms of time and space) in practice than selection sort.

Because of this assumption, counting sort does not need to compare the inputs during sorting, and thus it can achieve a linear time.

Selection sort, on the other hand, does not assume the range of the input numbers. It is a *comparison sort* since it compares the input numbers during sorting. It is therefore more general and has a wider range of applications.

We now look at two more comparison-based sorting algorithms.

Bubble Sort

Bubble sort is probably the most well known, under-performed sorting algorithm[1], but is taught in most CS classes because of its simplicity. The idea of bubble sort is to make multiple passes through the list. In each pass, we look for all possible adjacent pairs of items. Any adjacent pair that is out of order is swapped so that they are in order. This process repeats until everything is in order.

Let's look at an example. Suppose we have, as an input, the numbers `8 4 23 42 16 15`. In the first pass, we start from the first item and check from left to right. The pair `8 4` is out of order, so we swap them, and we get `4 8 23 42 16 15`. The pair `8 23` and `23 42` are in order, so we do not need to swap them. The pair `42 16` is out of order. We swap them and get `4 8 23 16 42 15`. The pair `42 15` is again out of order, so we swap them and get `4 8 23 16 15 42`.

The following sequence show the first pass through the array:

```
 8  4 23 42 16 15   <- swap
-- --
 4  8 23 42 16 15
   -- --
 4  8 23 42 16 15
      -- --
 4  8 23 42 16 15   <- swap
         -- --
 4  8 23 16 42 15   <- swap
            -- --
 4  8 23 16 15 42
```

After the first pass, notice that the largest element, 42, "bubbles" up through the list until it reaches the maximum position. We can now make the second pass, but we can exclude the last item since it is already in place.

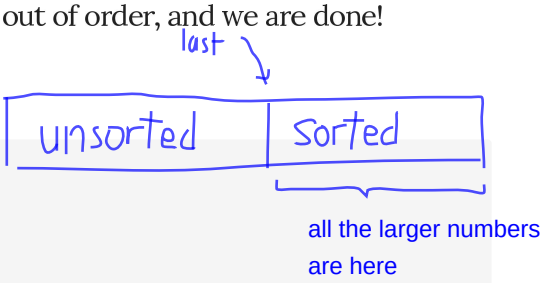
```
 4  8 23 16 15 42
-- --
 4  8 23 16 15 42
   -- --
 4  8 23 16 15 42   <- swap
      -- --
 4  8 16 23 15 42   <- swap
         -- --
```

```
 4  8 16 15 23 42
       -- --
```

After the second pass, the second largest element, 23, is in its position. So we can exclude this item in the subsequent pass.

The rest of the passes operates similarly. In the i-th pass, we scan through array item 0 to $n-i$, swapping any adjacent element that is out of order, until $i = n-1$, in which case we only have two elements, we swap them if we are out of order, and we are done!

The code for bubble sort can be written as follows:




```
void bubble_pass(long last, long a[])
{
  for (long i = 0; i < last; i += 1) {
    if (a[i] > a[i+1]) {
      swap(a, i, i+1);
    }
  }
}

void bubble_sort(long n, long a[]) {
  for (long last = n - 1; last > 0; last -= 1) {
    bubble_pass(last, a);
  }
}
```

How many steps does it take to bubble-sort an array of n elements? Since the i-th pass scans through $n-i$ elements, and there are a n passes in total, the analysis is similar to the one we did for the algorithm to compute the Kendall-tau distance -- bubble sort takes $O(n^2)$ steps.

Insertion Sort

The next sorting algorithm we are going to discuss is the insertion sort. This is another classic algorithm, that could perform better than bubble sort in some scenarios. The idea of insertion sort is simple: we partition the input list into two, a sorted partition, and an unsorted partition. Then we repeatedly take the first element from the unsorted partition, find its rightful place in the sorted partition, and *insert* it into place. We start with a sorted partition of one element, and we end if the sorted partition contains all the elements.

Take `8 4 23 42 16 15` as an example. I will use `|` to partition the array into a left sorted partition, and a right, unsorted, partition.

```
8 | 4 23 42 16 15
```

We pick the first element on the unsorted partition, 4, and insert it into the sorted partition. This involves shifting the elements in the sorted partition to the right until we find the rightful place for `4`. After this step, the sorted partition grows by 1 and the unsorted partition shrinks by 1.

```
4 8 | 23 42 16 15
```

In the next round, we take `23`, and finds its rightful place. It turns out `23` is already in its correct place.

```
4 8 23 | 42 16 15
```

In the next step, `42` is also in its correct place.

```
4 8 23 42 | 16 15
```

`16` is the next element, and we insert it between 8 and 23.

```
4 8 16 23 42 | 15
```

Finally, we insert `15` and we are done, as there is no more element in the unsorted partition.

```
4 8 15 16 23 42
```

The code for insertion sort can be written as follows:

```
1  void insert(long a[], long curr)
2  {
3    long i = curr - 1;
4    long temp = a[curr];
5    while (temp < a[i] && i >= 0) {
6      a[i+1] = a[i];
7      i -= 1;
8    }
9    a[i+1] = temp;
10 }
11
12 void insertion_sort(long n, long a[]) {
13   for (long curr = 1; curr < n; curr += 1) {
14     insert(a, curr);
15   }
16 }
```

checking if list hasnt ended yet &&
checking if they are bigger

swapping/shifting the elements
- impt if not the old element will be
over written

Animation

Animations for various sorting algorithms, including some which you will learn in CS2040C, are available online on VisuAlgo

Problem Set 24

Problem 24.1

In the implementation of bubble sort above, we always make $n - 1$ passes through the array. It is, however, possible to stop the whole sorting procedure, when a pass through the array does not lead to any swapping. Modify the code above to achieve this optimization.

Problem 24.2

(a) Suppose the input list to insertion sort is already sorted. What is the running time of insertion sort?

(b) Suppose the input list to insertion sort is inversely sorted. What is the running time of insertion sort?

Problem 24.3

What is the loop invariant for the loop in the function `insert`?

Problem 24.4

In certain scenarios, comparison is more expensive than assignment. For instance, comparing two strings is more expensive than assigning a string to a variable. In this case, we can reduce the number of comparisons during insertion sort by doing the following:

repeat

- take the first element X from unsorted partition
- use binary search to find the correct position to insert X
- insert X into the right place

until the unsorted partition is empty.

Implement the variation to insertion sort above. You may use your solution from Problem 23.1.

---

1. https://www.youtube.com/watch?v=k4RRi_ntQc8 ↵