Post 2: Sorting Algorithms

Stat 133, Fall 2017

Andrew Peng

Sorting for Data Preparation

Motivation for Sorting data

Before we do any type of data analysis, it is important to process it in a way that is easy to read and analyze. One way of preparing data is to sort data. Any type of data that can be arranged in a sequence can be sorted. However, in order to sort data there needs to a concept of order such that any object X can be said to be greater than or less than another object Y. We can use many types of sorting algorithms in order to categorize our data. The most common ones used are comparison based sorts, including selection sort, quicksort, insertion sort, and mergesort. Sorting your data before hand can increase the execution time of certain algorithms commonly used, such as binary search.

Learning to sort data

There are built in ways in R to sort data, but it is important to understand what is going on behind the scenes. There are many naive ways to sort data, but before we dive into the algorithms themselves, we should first create some sample data to sort. Since we are going to get a sorted array at the end, we can create a sequence and shuffle it.

```
# A shuffled array of numbers from 1 to 20

dat <- seq(1:20)
dat <- sample(dat, replace = F)
dat
```

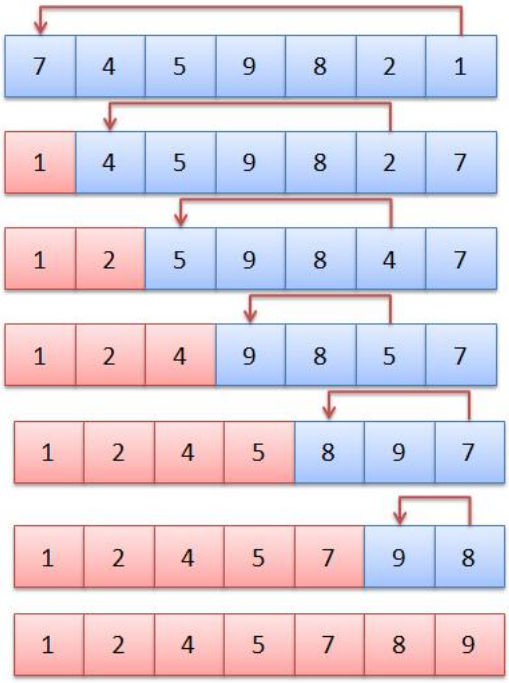
```
##  [1] 14 19 20 18 17  1  4 13 12  2  6 16  8 15  9  7 10 11  5  3
```

Here we are using integer numbers to be sorted, but any types of objects can be sorted as long as they can be compared. For example we can sort strings by length or cars by weight for example.

Selection sort

Selection sort is the most simple sort. It is usually the first thing that comes to mind when we sort things by hand. The algorithm goes like this. Say we have a vector of unsorted integers. For each iteration, we scan the vector and remove the smallest element and move it to the front and fix it.

1. Find the smallest non-fixed element in the array.
2. Swap it with the first element
3. Fix it
4. Repeat until the whole array is fixed (and sorted!)



Lets try it with our array

```
selection_sort <- function(arr) {
  for (i in 1:length(arr)) {
    curr_min = i
    for (j in i:length(arr)) {
      if (arr[j] < arr[curr_min]) {
        curr_min = j
      }
    }
    temp = arr[i]
    arr[i] = arr[curr_min]
    arr[curr_min] = temp
  }
  return(arr)
}

selection_sort(dat)
```

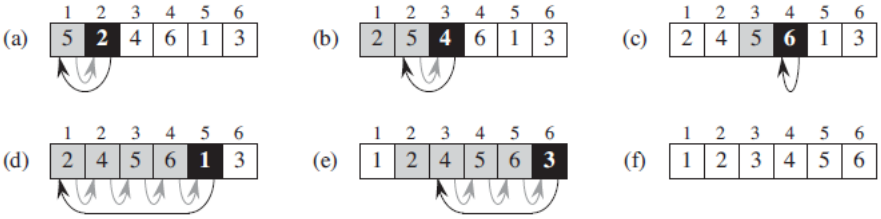
```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Insertion sort

Insertion sort is another common sorting algorithm. It works by shifting elements to their correct positions inside of the array. It turns out that this can be done extremely quickly, so insertion sort is very commonly used with smaller arrays.

For each element in the vector:

1. Until the item to it's left is less than it, swap them
2. Continue doing so until there are no more elements to swap



Lets try it with our data:

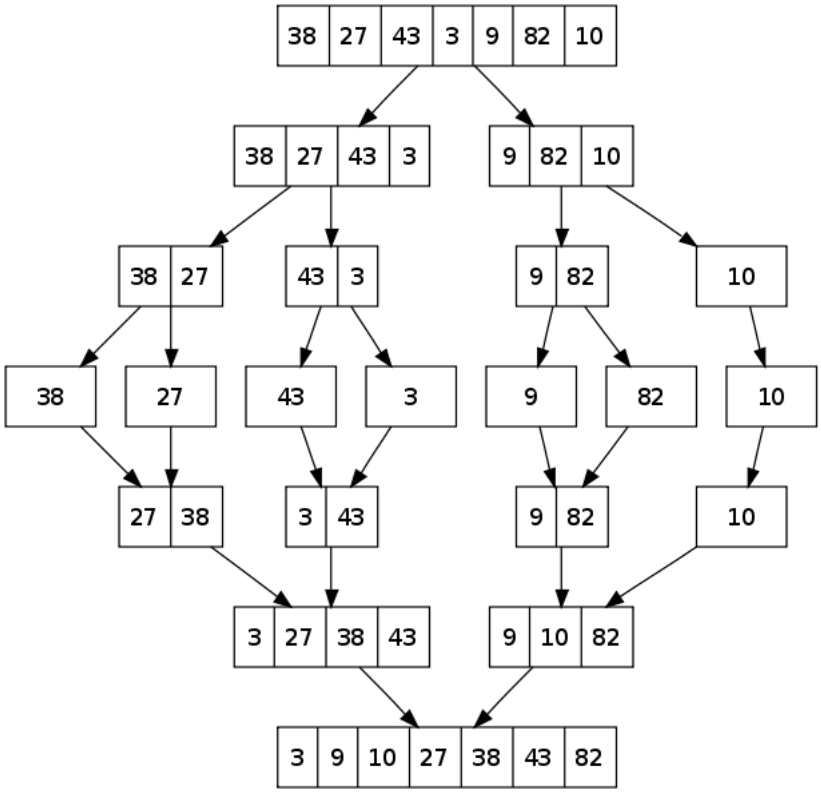
```
insertion_sort <- function(arr) {
  for (j in 1:length(arr)) {
    i=j
    while (i-1 >= 1 && (arr[i-1] > arr[i])) {
      # swap elements
      temp = arr[i]
      arr[i] = arr[i-1]
      arr[i-1] = temp
      i = i - 1
    }
  }
  return(arr)
}

insertion_sort(dat)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Merge sort

One of the most commonly used sorting algorithms used by programming languages is merge sort or some variation of mergesort. This sort uses a divide and conquer technique to recursively sort elements in each sub-array and then combine them at the end to merge them into one big sorted array. It sounds complicated but is relatively simple.

1. Divide array into halves
2. Run Mergesort recursively on each half (if the size is one, return itself)
3. Combine the two sorted arrays




```
merge_sort <- function(arr) {
  if (length(arr) <= 1) {
    return(arr)
  } else {
    mid = as.integer(length(arr)/2)
    # Recursive call
    left = merge_sort(arr[1:mid])
    right = merge_sort(arr[(mid+1):length(arr)])
    # Combine the arrays
    lp = 1
    rp = 1
    final = c()
    while (lp <= length(left) && rp <= length(right)) {
      if (left[lp] <= right[rp]) {
        final = c(final, left[lp])
        lp = lp + 1
      } else {
        final = c(final, right[rp])
        rp = rp + 1
      }
    }
    if (lp <= length(left)) {
      final = c(final, left[lp:length(left)])
    } else if (rp <= length(right)) {
      final = c(final, right[rp:length(right)])
    }
    return(final)
  }
}

merge_sort(dat)
```

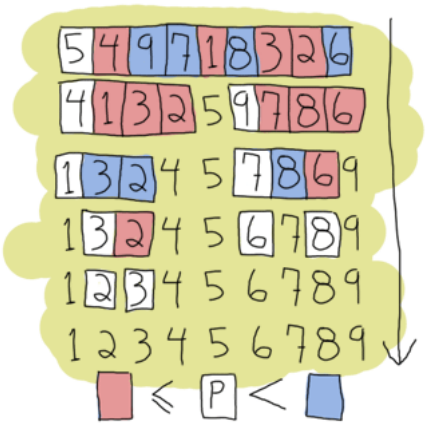
```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Quick sort

Quicksort is actually the most widely used variation of the comparison based sorting algorithm based on its execution speed compared to mergesort. It works in a similar divide and conquer strategy, but works a bit differently. Quick sort is based on the idea of partitioning the array. It picks a pivot and proceeds to sort all elements with values less than the pivot and all elements with values greater than the pivot and appends them together with the pivot in the middle.

1. Select a pivot
2. Place all items less than the pivot to the left. Place all items greater than the pivot to the right.
3. Run Quicksort on the left and right of the pivot recursively.

In the image the white element represents the pivot



Lets try it with our data

```
quick_sort <- function(arr) {
  if (length(arr) <= 1) {
    return(arr)
  } else {
    pivot = arr[1]
    left = c()
    right = c()
    for (i in 2:length(arr)) {
      if (arr[i] <= pivot) {
        left = c(left, arr[i])
      } else {
        right = c(right, arr[i])
      }
    }
    return(c(quick_sort(left),pivot,quick_sort(right)))
  }
}

quick_sort(dat)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Conclusion

In conclusion, sorting is a very valuable tool to have and it is important for engineers and data scientists to understand what goes on when using `sort()`. There are many types of sorting algorithms we used and learned. In practice, most built in implementations use some flavor of quick sort. Quick sort is used because although it has the same asymptotic speed as mergesort, in practice it runs faster because of cache coherency. On large datasets that cannot fit into memory (in big data scenarios), merge sort is more common. Sorting comes up in a lot of applications and algorithms so we need to make sure we do it efficiently and correctly which means we have to understand how basic sorting algoritms work

References

1. https://en.wikipedia.org/wiki/Sorting_algorithm
2. https://en.wikipedia.org/wiki/Quicksort
3. https://en.wikipedia.org/wiki/Merge_sort
4. https://en.wikipedia.org/wiki/Insertion_sort
5. http://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/04MergeQuick.pdf
6. https://www.crondose.com/2016/07/sorting-algorithms-important/
7. https://arxiv.org/ftp/arxiv/papers/1511/1511.03404.pdf