Sorting

August 7, 2020

1 Sorting

https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/InSort.html - visualize sorting algorithms and other data structures: - https://visualgo.net/en/sorting - https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

1.1 Table of Contents

1.2 Sorting - Introduction

- sort is a common task in daily lives: sorting playing cards; bills and piles of papers, books; jars of spices, etc.
- sort is one of the most frequently performed computing tasks
  - sort records in database
  - lot of problems require sorting to quickly find and shift through and work with data
- many algorithms have been devised
  - each may perform well on a particular type of data or problem
  - introduce us the technique of `divide and conquer`
  - introduce us the multiple ways to do the dividing

1.3 Sorting problem

- given a range of values or records $r_1, r_2, ..., r_n$ with associated key values $k_1, k_2, ..., k_3$, the sorting problem is to arrange the records into any order s such that the records $r_{s1}, r_{s2}, ..., r_{s3}$ have keys obeying the property $k_{s1} \leq k_{s2} \leq ... \leq k_{sn}$
- **stable sort** - if duplicate keys are allowed in the problem data then the initial ordering among duplicates are maintained

1.4 Analyzing sort algorithms

- it is traditional to measure the cost by counting the number of comparisons made between keys
  - measure is usually closely related to the actual running time for the algorithm
  - has the advantage of being machine and data-type independent
- if the records being sorted are too large to move, it may be appropriate to count the number of swap operations
- generally, we can assume that all records and keys are of fixed length, and that a single comparison or swap requires a constant amount of time regardless the data type of keys
- if applications require that a small number of records be sorted frequently, the constants in the running time equations that usually get ignored in asymptotic analysis now become crucial
- some sorting algorithm may require significant extra memory beyond the input sequence

1.4.1 Question:

In which case might the number of comparisons NOT be a good representation of the cost for a sorting algorithm? 1. When the CPU is really fast 2. When the amount of available space is small 3. When there are lots of records 4. When we are comparing strings of widely varying length

1.5 header includes required for this notebook & helper functions

[1]:
```
#include <iostream>
#include <string>
#include <vector>
#include <random>
#include <cstdlib>
#include <iterator>

using namespace std;
```

[2]:
```
// operator<< overloaded to print a vector
template<class T>
ostream& operator<<(ostream& out, const vector<T>& v) {
    char comma[3] = {'\0', ' ', '\0'};
    out << '[';
    for (auto& e: v) {
        out << comma << e;
        comma[0] = ',';
    }
    out << "]";
    return out;
}
```

[3]:
```
// function to generate random numbers
void generateRandomNumbers(vector<int> &rands, int start, int end) {
    // fill the vectors with random numbers
    random_device rd;
    //https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine
    // generates high quality random unsigned ints
    mt19937 mt(rd());
    uniform_int_distribution<> dis(start, end);
    // numbers between start and end inclusive
    generate(rands.begin(), rands.end(), bind(dis, ref(mt)));
}
```

1.6 Exchange Sorting Algorithms

- bubble sort, selection sort and insertion sort
- swapping adjacent records
- crucial bottleneck is that only *adjacent* records are compared

1.7 Bubble Sort

- https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/BubbleSort.html
- https://en.wikipedia.org/wiki/Bubble_sort
- simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order
- the pass through the list is repeated until the list is sorted
- also called sinking sort, where larger elements are "bubble" to the end of the list
- **advantages**:
  - simple
  - stable
  - practical if the input is in mostly sorted order with some out-of-order elements nearly in position
  - in-place; only requires a constant amount of additional memory space
- **disadvantages**:
  - too slow and impractical for most problems even when compared to **insertion sort**

[4]:
```
// 1st implementation of bubble sort
// implementation that works on integer vector
void bubbleSort0(vector<int> &v) {
    for(int pass=0; pass<v.size(); pass++) {
        for (int i=1; i<v.size(); i++)
            // if this pair is not in order, swap them
            if (v[i-1] > v[i]) {
                swap(v[i-1], v[i]);
            }
        cout << "pass # " << pass+1 << endl;
    }
}
```

[5]:
```
vector<int> nums(20);
```

[6]:
```
generateRandomNumbers(nums, 0, 20);
cout << nums << endl;
```

```
[5, 16, 18, 2, 11, 2, 14, 2, 9, 14, 12, 19, 0, 9, 12, 3, 17, 0, 7, 7]
```

[7]:
```
// copy nums to nums1 to use with other bubbleSort implementations
vector<int> nums1 = nums;
```

[8]:
```
// test bubbleSort0 with random nums
bubbleSort0(nums);
cout << nums << endl;
```

```
pass # 1
pass # 2
pass # 3
pass # 4
pass # 5
pass # 6
pass # 7
pass # 8
pass # 9
pass # 10
pass # 11
pass # 12
pass # 13
pass # 14
pass # 15
pass # 16
pass # 17
pass # 18
pass # 19
pass # 20
[0, 0, 2, 2, 2, 3, 5, 7, 7, 9, 9, 11, 12, 12, 14, 14, 16, 17, 18, 19]
```

[9]:
```
// test bubbleSort0 with sorted nums
bubbleSort0(nums);
cout << nums << endl;
```

```
pass # 1
pass # 2
pass # 3
pass # 4
pass # 5
pass # 6
pass # 7
pass # 8
pass # 9
pass # 10
pass # 11
pass # 12
pass # 13
pass # 14
pass # 15
pass # 16
pass # 17
pass # 18
pass # 19
pass # 20
[0, 0, 2, 2, 2, 3, 5, 7, 7, 9, 9, 11, 12, 12, 14, 14, 16, 17, 18, 19]
```

[10]:
```
// 2nd Algorithm of bubble sort
// bubble sort implementation that works on integer vector
void bubbleSort1(vector<int> &v) {
    bool sorted;
    int pass = 0;
    do {
        sorted = true;
        for (int i=1; i<v.size(); i++) {
            // if this pair is not in order, swap them
            if (v[i-1] > v[i]) {
                swap(v[i-1], v[i]);
                sorted = false;
            }
        }
        cout << "pass # " << ++pass << endl;
    } while (! sorted);
}
```

[11]:
```
// copy orginal random int vector back to nums
nums = nums1;
cout << nums << endl;
```

```
[5, 16, 18, 2, 11, 2, 14, 2, 9, 14, 12, 19, 0, 9, 12, 3, 17, 0, 7, 7]
```

[12]:
```
// bubble sort on random nums
bubbleSort1(nums);
cout << nums << endl;
```

```
pass # 1
pass # 2
pass # 3
pass # 4
pass # 5
pass # 6
pass # 7
pass # 8
pass # 9
pass # 10
pass # 11
pass # 12
pass # 13
pass # 14
pass # 15
pass # 16
pass # 17
[0, 0, 2, 2, 2, 3, 5, 7, 7, 9, 9, 11, 12, 12, 14, 14, 16, 17, 18, 19]
```

[13]:
```
// bubble sort on sorted nums
bubbleSort1(nums);
cout << nums << endl;
```

```
pass # 1
[0, 0, 2, 2, 2, 3, 5, 7, 7, 9, 9, 11, 12, 12, 14, 14, 16, 17, 18, 19]
```

[14]:
```
// 3rd implementation of bubble sort
// implementation that works on integer vector
void bubbleSort2(vector<int> &v) {
    bool sorted;
    for(int pass=0; pass<v.size()-1; pass++) {
        sorted = true;
        for (int i=1; i<v.size()-pass; i++) {
            // if this pair is not in order, swap them
            if (v[i-1] > v[i]) {
                swap(v[i-1], v[i]);
                sorted = false;
            }
        }
        cout << "pass # " << pass+1 << endl;
        if (sorted)
            break;
    }
}
```

[15]:
```
nums = nums1;
cout << nums << endl;
```

```
[5, 16, 18, 2, 11, 2, 14, 2, 9, 14, 12, 19, 0, 9, 12, 3, 17, 0, 7, 7]
```

[16]:
```
// bubbleSort2 on random nums
bubbleSort2(nums);
cout << nums << endl;
```

```
pass # 1
pass # 2
pass # 3
pass # 4
pass # 5
pass # 6
pass # 7
pass # 8
pass # 9
pass # 10
pass # 11
pass # 12
pass # 13
pass # 14
pass # 15
pass # 16
pass # 17
[0, 0, 2, 2, 2, 3, 5, 7, 7, 9, 9, 11, 12, 12, 14, 14, 16, 17, 18, 19]
```

[17]:
```
// bubbleSort2 on sorted nums
bubbleSort2(nums);
cout << nums << endl;
```

```
pass # 1
[0, 0, 2, 2, 2, 3, 5, 7, 7, 9, 9, 11, 12, 12, 14, 14, 16, 17, 18, 19]
```

1.8 Bubble Sort (Algorithm #2 and #3) - Asymptotic Analysis

- **Worst-case complexity:** $\Theta(n^2)$ comparisons and $\Theta(n^2)$ swaps
- **Average-case complexity:** $\Theta(n^2)$ comparisons and $\Theta(n^2)$ swaps
- **Best-case complexity:** $\Theta(n)$ comparisons and $\Theta(1)$ swaps
- **Worst-case space complexity:** $\Theta(n)$ total and $\Theta(1)$ auxiliary
- see for details: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/BubbleSort.html

1.8.1 Exercise

1. What is the running time of Bubble Sort (Implementation #2 & #3) when the input is an array where all records key values are equal?

    1. $\Theta(nlogn)$
    2. $\Theta(n^2)$
    3. $\Theta(n)$
    4. $\Theta(logn)$

2. What is the running time of Bubble Sort when the input is an array where all records are sorted in descending order?

    1. $\Theta(nlogn)$
    2. $\Theta(n^2)$
    3. $\Theta(n)$
    4. $\Theta(logn)$

1.9 Selection Sort

- https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/SelectionSort.html
- consider sorting stack of phone bills from the past year by date:
    - look through the pile until you find the bill for January, and pull that out
    - look through the remaining pile until you find the bill February and put that behind January
    - proceed through the shrinking pile of bills to select the next one in order until you are done
    - this is the inspiration for **Selection Sort**
- selection sort finds the smallest key in an unsorted list, then the next smallest, and so on
- may also find find the largest key in and unsorted list, then the next largest, and so on (as in the ODSA Text)
- it's unique feature is that there are few record swaps
    - to find the next smallest key value requires searching through the entire unsorted portion of the list, but only one swap is required to put the record in place
- **advantages:**
    - simple
    - fewer swaps
- **disadvantages:**
    - not efficient for large data sets
    - generally performs worse than **Insertion Sort**
    - normally not stable
- for visualization use this: https://visualgo.net/en/sorting

[18]:
```
// implementation of Selection Sort
// sorts a vector of integers
void selectionSort(vector<int> &v) {
    for (int i=0; i<v.size()-1; i++) { // unsorted position
        int minIndex = i; // current smallest index
        for(int j=i+1; j < v.size(); j++) { // search for the smallest index
            if (v[j] < v[minIndex])
                minIndex = j;
        }
        if (v[minIndex] != v[i]) // no swap if v[minIndex] == v[i]; this makes␣
 ↪it stable
            swap(v[minIndex], v[i]);
    }
}
```

[19]:
```
// test selection sort
generateRandomNumbers(nums, 0, 20);
cout << nums1 << endl;
```

```
[5, 16, 18, 2, 11, 2, 14, 2, 9, 14, 12, 19, 0, 9, 12, 3, 17, 0, 7, 7]
```

[20]:
```
// selection sort vector
selectionSort(nums1);
cout << nums1 << endl;
```

```
[0, 0, 2, 2, 2, 3, 5, 7, 7, 9, 9, 11, 12, 12, 14, 14, 16, 17, 18, 19]
```

1.10 Selection Sort - Asymptotic Analysis

- **Worst-case performance:** $\Theta(n^2)$ comparisons, $\Theta(n)$ swaps
- **Average-case performance:** $\Theta(n^2)$ comparisons, $\Theta(n)$ swaps
- **Best-case performance:** $\Theta(n^2)$ comparisons, $\Theta(n)$ swaps
- **Worst-case space complexity:** $\Theta(n)$ total, $\Theta(1)$ auxiliary
- for details see: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/SelectionSort.html

1.10.1 Exercise

1. Selection Sort (as implemented in this notebook) is a stable sorting algorithm. Recall that a stable sorting algorithm maintains the relative order of records with equal keys
   1. True
   - False
2. What is the number of swap required in Selection Sort (as implemented in this notebook) to sort already sorted list?
   1. $\Theta(n^2)$
   - $\Theta(n)$
   - $\Theta(logn)$
   - $\Theta(1)$
   - 0

1.11 Insertion Sort

- consider again sorting a pile of bill for last year:
  - look at the first two bills and put them in order
  - take the third bill and put it into the right position with respect to the first two, and so on…
  - as you take each bill, you would insert it to the sorted pile that you have already made
  - this real-world example is the inspiration for **Insertion Sort**
- when sorting cards in a bridge hand, most use technique similar to insertion sort
- **advantages**:
  - simple
  - efficient for small data sets
  - nearly sorted list is always cheap to sort with **Insertion Sort**

  - more efficient in practice than **Bubble Sort** and **Selection Sort**
  - stable: does not change the relative order of elements with equal keys
  - in-place: only requires a constant amount $O(1)$ additional memory space
  - online: can sort a list as it receives it
- **disadvantages**:
  - much less efficient than quicksort, heapsort, or merge sort on large lists

- visualize it: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/InsertionSort.html

[21]:
```
// implementation of Insertion Sort
// sorts vecotor of integers
void insertionSort(vector<int> &v) {
    for(int i=1; i< v.size(); i++) {
        for(int j=i; (j > 0) && (v[j] < v[j-1]); j--)
            //if (v[j] < v[j-1])
                swap(v[j], v[j-1]);
            // else break;
    }
}
```

[22]:
```
// test insertionSort
vector<int> nums2(20);
```

[23]:
```
generateRandomNumbers(nums2, 0, 20);
cout << nums2 << endl;
```

```
[0, 5, 8, 11, 1, 1, 9, 5, 9, 9, 15, 9, 16, 13, 9, 11, 2, 19, 8, 11]
```

[24]:
```
// insertion sort nums
insertionSort(nums2);
cout << nums2 << endl;
```

```
[0, 1, 1, 2, 5, 5, 8, 8, 9, 9, 9, 9, 9, 11, 11, 11, 13, 15, 16, 19]
```

1.12 Insertion Sort - Asymptotic Analysis

- **Worst-case performance:** $\Theta(n^2)$ comparisons and swaps
- **Average-case performance:** $\Theta(n^2)$ comparisons and swaps
- **Best-case performance:** $\Theta(n)$ comparisons, $\Theta(1)$ swaps
- **Worst-case space complexity:** $\Theta(n)$ total, $\Theta(1)$ auxiliary
- see for details: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/InsertionSort.html

1.12.1 Exercise

What is the running time of Insertion Sort when the input is an array that has already been sorted? 1. $\Theta(n^2)$ 2. $\Theta(n)$ 3. $\Theta(nlogn)$ 4. $\Theta(logn)$

1.13 Summary of Cost of Exchange Sorting Implemented in this Notebook

Algorithms	Bubble	Selection	Insertion
Comparisons:	-	-	-
Best case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n)$
Average case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps:	-	-	-
Best case	0	0	0
Average case	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
Worst case	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$

1.14 Divide & Conquer

- natural approach to problem solving
- can be used to when sorting:
  - consider breaking the list to be sorted into pieces, process the pieces, and then put them back together somehow
- Shellsort, Mergesort, Quicksort, Binsort, Radix sort, etc.

1.15 Shellsort

https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/Shellsort.html#id2

- named after inventor, D. L. Shell (published in 1959)
- sometimes also called **diminishing increment sort**
- no real-life intuition to inspire Shellsort from
- faster but harder to implement compared to Insertion Sort and Selection Sort, e.g.
- the key idea is to exploit the best-case performance of Insertion Sort
  - when a list is sorted or nearly sorted, Insertion Sort runs in linear time
- the strategy is to quickly make the list "mostly sorted", so that a final Insertion Sort can finish the job in linear time
  - uses "virtual" sublists that are often not contiguous defined by increment, I, or **gap**
  - each sublist is sorted using the Insertion Sort
  - at each stage, Insertion Sort is either working on a small list (and is fast) or is working on a nearly sorted larger list (and again is fast)
  - each record in a given sublist is I positions apart
    - e.g., if the increment were 4, then each record in the sublist would be 4 positions apart
- **selecting increment,** I:
  - one possible implementation is to use increments that are all powers of two
  - e.g., pick largest power of two less than n
    - generate I sublists of 2 records each
    - e.g., if there were 16 records (indexed from 0-15), we initially have 8 (2^3) sublists, with each record in the sublist being 8 positions apart
    - the first sublist would be the records in positions 0 and 8
    - the second sublist would be the records in positions 1 and 9 and so on

        * subsequent increment will be half of the previous one with the final increment to be 1
    - overall, so long as each increment is smaller than the last, and the last increment is 1, Shellsort will work!
    - see visualizations here with different increments I: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/Shellsort.html#id2
- the following list has 9 items, with an increment of 3 (three sublists)
    - figure used from: http://interactivepython.org/runestone/static/pythonds/SortSearch/TheShellSort.ht incrementsc
- the following figure shows almost sorted list
- final insertion sort (increment of 1); only 4 swaps required
- see animation demos:
    - here: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/Shellsort.html#id2
    - or here: https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

[25]:
```
// modified insertionSort for shellSort to work with varying increments
void insertionSort2(vector<int> &v, int start, int incr) {
    for(int i=start+incr; i<v.size(); i+=incr)
        for(int j=i; (j >=incr) && v[j] < v[j-incr]; j-=incr)
            swap(v[j], v[j-incr]);
}
```

[26]:
```
// implementation of Shellsort
// Sorts vector of integers
void shellSort(vector<int> &v) {
    // begin with n/2 sublists and sort them
    // on the next pass, n/4 sublists are sorted and so on until size >= 3
    for (int i=v.size()/2; i>2; i/=2) // for each increment
        for(int j=0; j<i; j++) // Sort each sublist of that increment
            insertionSort2(v, j, i);

    insertionSort2(v, 0, 1); // can call regular insertionSort
}
```

[27]:
```
// test shellSort
vector<int> randNums(20);
```

[28]:
```
generateRandomNumbers(randNums, 0, 20);
cout << randNums << endl;
```

```
[9, 15, 7, 10, 8, 20, 15, 15, 9, 12, 12, 12, 4, 4, 7, 6, 20, 18, 9, 0]
```

[29]:
```
shellSort(randNums);
cout << randNums << endl;
```

```
[0, 4, 4, 6, 7, 7, 8, 9, 9, 9, 10, 12, 12, 12, 15, 15, 15, 18, 20, 20]
```

1.16 Shellsort - Asymptotic Analysis

- **Worst-case performance:** $O(n^2)$ - (if worst increment, I, is picked!)
- **Best-case performance:** $O(nlogn)$
- **Average-case performance:**- depends on increment, I
- **Worst-case space complexity:** $O(n)$ total; $O(1)$ auxiliary
- for details see: https://en.wikipedia.org/wiki/Shellsort

1.16.1 Exercise

Is the following a legal series of increments when running Shellsort on an array of 16 values? 10, 3, 2, 1 - Yes - No

1.17 Mergesort

https://en.wikipedia.org/wiki/Merge_sort - split the list in half, sort the halves, and then merge the sorted halves together - one of the simplest sorting algorithms conceptually - has good performance both in asymptotic and in empirical running time - stable sort - relatively difficult to implement in practice - algorithm: 1. divide the unsorted list into n *sublist*, each containing one element (a list of one element is considered sorted) 2. repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining (sorted list) - sorting happens during merge - pseudocode:

```
List mergesort(List inlist) {
    if (inlist.length() <= 1) return inlist;;
    List L1 = half of the items from inlist;
    List L2 = other half of the items from inlist;
    return merge(mergesort(L1), mergesort(L2));
}
```

- see this for visualization: https://visualgo.net/en/sorting

[30]:
```
// implementation of merge sort
// sorts vector of integers

// merges two sub vectors v[0..mid] and v[mid+1..right]
void merge(vector<int> &v, int left, int mid, int right) {

    int i, j, k;
    int n1 = mid - left + 1;
    int n2 =  right - mid;

    vector<int> L(n1); // auxiliary vector
    vector<int> R(n2); // auxiliary vector

    // copy data to temp Left and Right vectors
    for(i=0; i<n1; i++)
        L[i] = v[left+i];

    for (j=0; j<n2; j++)
        R[j] = v[mid+1+j];
    // merge the temp vectors back into v
    i = 0;
    j = 0;
    k = left; // initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            v[k] = L[i];
            i++;
        }
        else {
            v[k] = R[j];
            j++;
        }
        k++;
    }
    // copy the remaining elements of L vector if there's any
    while(i < n1) {
        v[k] = L[i];
        i++;
        k++;
    }
    // copy the remaining elements of R vector if there's any
    while(j < n2) {
        v[k] = R[j];
        j++;
        k++;
    }
}
```

[31]:
```
// recursive Top-Down Merge Sort implementation
void mergeSort(vector<int> &v, int left, int right) {
    if (left < right) { // general case
        int mid = left+(right-left)/2; // same as (left+right)/2 but avoid␣
 ↪overflow
        mergeSort(v, left, mid);
        mergeSort(v, mid+1, right);
        // merge two sorted list
        merge(v, left, mid, right);
    }
}
```

[32]:
```
// test merge sort
vector<int> nums3(20);
```

[33]:
```
generateRandomNumbers(nums3, 0, 20);
cout << nums3 << endl;
```

```
[17, 12, 14, 9, 5, 16, 1, 5, 8, 10, 5, 6, 3, 13, 14, 5, 11, 8, 1, 2]
```

[34]:
```
mergeSort(nums3, 0, nums3.size()-1);
cout << nums3 << endl;
```

```
[1, 1, 2, 3, 5, 5, 5, 5, 6, 8, 8, 9, 10, 11, 12, 13, 14, 14, 16, 17]
```

1.18 Merge Sort - Asymptotic Analysis

- **Worst-case performance:** $\Theta(nlogn)$
- **Best-case performance:** $\Theta(nlogn)$
- **Average-case performance:** $\Theta(nlogn)$
- **Worst-case space complexity:** $\Theta(n)$ total; $\Theta(n)$ auxiliary; $\Theta(1)$ with linked lists
- for details see: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/Mergesort.html

1.18.1 Exercise

1. Implement merge sort for linked list. Merge operation can be done without extra memory space.

2. In the worst case, the total number of comparisons for Mergesort is closest to:

    1. n

    - $\frac{n^2}{2}$
    - $nlogn$
    - n^2

1.19 Quicksort

- aptly named, **quicksort** is the fastest known general-purpose in-memory sorting algorithm in the average case
- doesn't require the extra array require by the **Merge sort**
- widely used, typically implemented in library such as UNIX, C qsort, and C++ sort
- interestingly, quicksort is hampered by exceedingly poor worst-case performance, thus making it inappropriate for certain applications
  - depends on how pivot is chosen; when leftmost or rightmost element is chose as pivot:
    1. array is already sorted in same or reverse order
    2. all elements are same
  - worst case can always occur if maximum or minimum element is chosen as pivot
- algorithm steps:
  - find the pivot (usually mid element)
  - stick it to the last index
  - partition the list into two halves such that:
    * elements in the first half are less than the pivot
    * elements in the second half are strictly greater than or equal to the pivot
  - move the pivot to the right position (element is in the right place)
  - quicksort the first half
  - quicksort the second half
- visualize it: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/Quicksort.html

[35]:
```
// picking middle index as pivot may avoid worst-case performance issue
int findPivot(int left, int right) {
    return (left+right)/2;
}
```

[36]:
```
// partition the array in such a way that
// elements before the pivot are smaller than the elements after the pivot␣
↪element
// left and right are indices, pivot is the pivot element
int partition(vector<int> &V, int left, int right, int pivot) {
    while (left <= right) { //move bounds inward until they meet
        // elements to the left sublist will be strictly less than pivot
        while (V[left] < pivot)
            left++;
        // elements to the right sublist will be strictly greater or equal to␣
↪pivot
        while ((right >=left ) && (V[right] >= pivot))
            right--;
        if (right > left)
            swap(V[left], V[right]);
    }
    return left; // return first position in right partition
}
```

[37]:
```
// implementation of Quicksort
// sorts vector of integers, first and last are indices
void quickSort(vector<int> &V, int first, int last) {
    if (first < last) {
        int pivotIndex = findPivot(first, last); // pick a pivot
        swap(V[pivotIndex], V[last]); // stick pivot at end
        // k will be the first position in the right sublist
        int k = partition(V, first, last-1, V[last]); // V[last] is pivot
        swap(V[k], V[last]); // put pivot in place; now pivot is in right␣
↪sorted position!
        quickSort(V, first, k-1); // sort left partition
        quickSort(V, k+1, last); // sort right partition
    }
}
```

[38]:
```
// test quickSort
vector<int> nums4(20);
```

[39]:
```
generateRandomNumbers(nums4, 0, 20);
cout << nums4 << endl;
```

```
[8, 0, 15, 5, 18, 3, 15, 3, 14, 3, 19, 12, 14, 19, 18, 10, 19, 0, 20, 18]
```

[40]:
```
quickSort(nums4, 0, nums4.size()-1);
cout << nums4 << endl;
```

```
[0, 0, 3, 3, 3, 5, 8, 10, 12, 14, 14, 15, 15, 18, 18, 18, 19, 19, 19, 20]
```

1.20 Quicksort - Asymptotic Analysis

- **Worst-case performance:** $\Theta(n^2)$ rarely when pivot yields a bad partitioning of the array
- **Base-case performance:** $\Theta(nlogn)$
- **Average-case performance:** $\Theta(nlogn)$
- **Worst-case space complexity:** $\Theta(n)$ total; $\Theta(1)$ auxiliary
- for detail see: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/Quicksort.html

1.20.1 Exercise

- What is the running time of quicksort when the input is an array where all record values are equal (with the above implementation)?
  1. $\Theta(n^2)$
  2. $\Theta(n^n)$
  - $\Theta(n)$
  - $\Theta(logn)$
  - $\Theta(nlogn)$

1.21 Summary of Cost of Divide & Conquer Sorting Implemented in this Notebook

Algorithms	Shellsort	Mergesort	Quicksort
Performance Class:	-	-	-
Best case	$\Theta(nlogn)$	$\Theta(nlogn)$	$\Theta(nlogn)$
Average case	depends on gap	$\Theta(nlogn)$	$\Theta(nlogn)$
Worst case	$\Theta(n^2)$	$\Theta(nlogn)$	$\Theta(n^2)$
Auxiliary Space	$\Theta(1)$	$\Theta(n)$; $\Theta(1)$ with linked list	$\Theta(logn)$

[ ]:
```
```