12. Sorting

Wednesday, June 1, 2022 5:30 PM

https://www.toptal.com/developers/sorting-algorithms

- Matters for big data sets
- Focusing on Bubble, Selection, insertion, Merge, Quick, Heap, radix, counting (bad; non-compare)
- Arrays.sort uses Timsort for object arrays (stable) (good)

http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/46c727d6ecc2/src/share/classes/java/util/DualPivotQuicksort.java

Quicksort for primitive arrays
insertion for small arrays
Merge for mostly sorted arrays
dual pivot Quicksort for everything else

Tradeoffs (when to use which)

- Bubble sort
  - never use, $O(n^2)$ average
  - double for loop, comparing $O(i)$ and $O(i+1)$ n^2 times

```
void bubbleSort(int arr[])
  {
     int n = arr.length;
     for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
           if (arr[j] > arr[j + 1]) {
              // swap arr[j+1] and arr[j]
              int temp = arr[j];
              arr[j] = arr[j + 1];
              arr[j + 1] = temp;
           }
  }
```

https://www.geeksforgeeks.org/bubble-sort/

- Selection Sort
  - slightly better than Bubble
    - $O(n^2)$ average, but selection has less swaps
  - Scans for smallest item, then swap the index

```
void sort(int arr[])
  {
     int n = arr.length;

     // One by one move boundary of unsorted subarray
```

https://www.geeksforgeeks.org/selection-sort/

```
    for (int i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first
        // element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Good sorts

Tradeoffs (when to use which)

https://www.geeksforgeeks.org/insertion-sort/

- Insertion sort
  - Good for small data sets and partially sorted data
  - O(n) best case, $O(n^2)$ avg/worse
  - split array into sorted and unsorted part, values from unsorted are picked and place in correct order in sorted part

GFG code

first pass:
- index 0 is unsorted section
  1 is sorted section

```
void sort(int arr[])
{
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int key = arr[i];   <- first unsorted element
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];   - swap
            j = j - 1;   - go back 1 index
        }
        arr[j + 1] = key;
    }
```

- repeat
  1) select first unsorted element
  2) swap other element to right to create correct pos & shift unsorted element
  3) advance pointer to right one.

$O(n^2)$

```
for ( ) {
    while ( ) {
    }
}
```

- Shifts all element to the right to create pos for unsorted element
- inserts unsorted element at correct pos

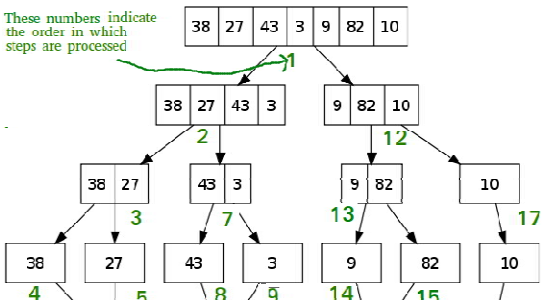
- Divide and conquer (Merge sort and Quick sort)
  - Merge: $O(n \log n)$ time, $O(n)$ space, stable sorting
  - Quick: $O(n \log n)$ time | $O(n^2)$ worse time, $O(\log n)$ space, not stable

https://www.geeksforgeeks.org/merge-sort/

Merge sort
- Divides array into 2 halves, calls itself for two halves (until it is 1 item), then merge sorted halves
- stable, if there are equivalent elements, the original order is preserved
- compares local list to each other instead of every element
- $O(n)$ space b/c we need to create divided arrays

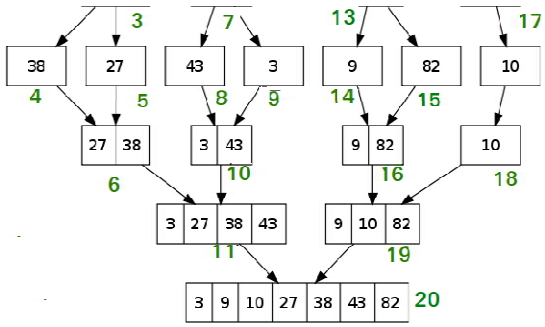
```
// Merges two subarrays of arr[].
  // First subarray is arr[l..m]
  // Second subarray is arr[m+1..r]
  void merge(int arr[], int l, int m, int r)
  {
    // Find sizes of two subarrays to be merged
    int n1 = m - l + 1;
    int n2 = r - m;

    /* Create temp arrays */
    int L[] = new int[n1];
    int R[] = new int[n2];

    /*Copy data to temp arrays*/
    for (int i = 0; i < n1; ++i)
      L[i] = arr[l + i];
    for (int j = 0; j < n2; ++j)
      R[j] = arr[m + 1 + j];

    /* Merge the temp arrays */

    // Initial indexes of first and second subarrays
    int i = 0, j = 0;

    // Initial index of merged subarray array
    int k = l;
    while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
        arr[k] = L[i];
        i++;
      }
```

compare elements of 2 subarrays and merges



```
        arr[k] = L[i];
        i++;
      }
      else {
        arr[k] = R[j];
        j++;
      }
      k++;
    }

    /* Copy remaining elements of L[] if any */
    while (i < n1) {
      arr[k] = L[i];
      i++;
      k++;
    }

    /* Copy remaining elements of R[] if any */
    while (j < n2) {
      arr[k] = R[j];
      j++;
      k++;
    }
}

// Main function that sorts arr[l..r] using
// merge()
void sort(int arr[], int l, int r)
{
    if (l < r) {
        // Find the middle point
        int m =l+ (r-l)/2;

        // Sort first and second halves
        sort(arr, l, m);
        sort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}
```

compare elements of 2 subarrays and merges them



sort() recursively halves array

merge() combines array

Main {

int arr[] = {...};

MergeSort ob = new MergeSort();

ob.sort(arr, 0, arr.length - 1);

}

recursively splits until left of arr = right of arr (only 1 element)

arr.length == 1

https://www.geeksforgeeks.org/quick-sort/

Quick sort

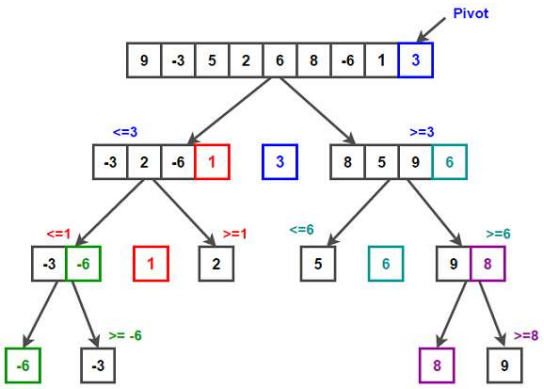
- picks an element as pivot (many ways) and partitions the array around the pivot

target of partition: given an element x as a pivot, place x in its correct position by putting all smaller elements before x and all larger elements after x

- O(log(n)) space

- O(n log n) time average, $O(n^2)$ worse if bad pivot

```
// A utility function to swap two elements
static void swap(int[] arr, int i, int j)
{
   int temp = arr[i];
   arr[i] = arr[j];
   arr[j] = temp;
}

/* This function takes last element as pivot, places
  the pivot element at its correct position in sorted
  array, and places all smaller (smaller than pivot)
  to left of pivot and all greater elements to right
  of pivot */
static int partition(int[] arr, int low, int high)
{

   // pivot
   int pivot = arr[high];

   // Index of smaller element and
   // indicates the right position
   // of pivot found so far
   int i = (low - 1);

   for(int j = low; j <= high - 1; j++)
   {

      // If current element is smaller
      // than the pivot
      if (arr[j] < pivot)
      {

         // Increment index of
         // smaller element
         i++;
         swap(arr, i, j);
      }
   }
   swap(arr, i + 1, high);
   return (i + 1);
}

/* The main function that implements QuickSort
        arr[] --> Array to be sorted,
        low --> Starting index,
        high --> Ending index
 */
static void quickSort(int[] arr, int low, int high)
{
   if (low < high)
   {

      // pi is partitioning index, arr[p]
      // is now at right place
      int pi = partition(arr, low, high);

      // Separately sort elements before
      // partition and after partition
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
```



```
    }
}
```

Merge vs Quick

- if worried about worse case : use merge
- if worried about memory : use Quick
  - unless the sorting is external, then use merge
- merge sort is stable

Radix sort and counting sort (non-comparison sort)

- only works with integers in a restricted range
- uses assumptions about how numbers are stored in memory

https://opendatastructures.org/newhtml/ods/latex/sorting.html#tex2htm-121