CSM 61B
Spring 2024

Sorting I

Mentoring 12: April 1 - 5th, 2024

This week's video walkthroughs: https://tinyurl.com/CSMsp24-week12

1 Sorting Introduction

1.1 Perform selection, heap, and merge sort on the following sequence of numbers:
`398 182 400 574 795 503 019 280.`

Selection sort:

```
019 | 182 400 574 795 503 398 280
019 182 | 400 574 795 503 398 280
019 182 280 | 574 795 503 398 400
019 182 280 398 | 795 503 574 400
019 182 280 398 400 | 503 574 795 *
019 182 280 398 400 503 | 574 795 *
019 182 280 398 400 503 574 | 795 *
019 182 280 398 400 503 574 795 |
```

* min value already in right index

Heap sort:

Bottom-up heapify:

```
- 398 182 400 574 795 503 019 280
- 398 795 503 574 182 400 019 280
- 795 398 503 574 182 400 019 280
- 795 574 503 398 182 400 019 280
```

Remove max & sink:

```
- 280 574 503 398 182 400 019 | 795
- 574 398 503 280 182 400 019 | 795
- 019 398 503 280 182 400 | 574 795
- 503 398 400 280 182 019 | 574 795

- 019 398 400 280 182 | 503 574 795
- 400 398 019 280 182 | 503 574 795

- 182 398 019 280 | 400 503 574 795
- 398 280 019 182 | 400 503 574 795

- 182 280 019 | 398 400 503 574 795
- 280 182 019 | 398 400 503 574 795

- 019 182 | 280 398 400 503 574 795
- 182 019 | 280 398 400 503 574 795

- 019 | 182 280 398 400 503 574 795

- | 019 182 280 398 400 503 574 795
```

Merge sort:

Divide phase:

```
398 182 400 574 | 795 503 019 280
398 182 | 400 574 | 795 503 | 019 280
398 | 182 | 400 | 574 | 795 | 503 | 019 | 280
```

Conquer phase:

```
182 398 | 400 574 | 503 795 | 019 280
182 398 400 574 | 019 280 503 795
019 182 280 398 400 503 574 795
```

1.2 Given the initial unsorted array `5 0 2 9 6 7 1 4`, match each sequence of intermediate step(s) to **selection, heap, or merge sort**. For heap sort, assume the sorting is done with the 0th item blank.

(a) `0 1 2 9 6 7 5 4`
`0 1 2 4 5 7 6 9`

Selection sort

(b) `0 5 2 9 6 7 1 4`
`0 2 5 9 1 4 6 7`

Merge sort

(c) `5 9 7 0 6 2 1 4`
`0 6 7 4 5 2 1 9`
`7 6 2 4 5 0 1 9`

Heap sort

1.3 What is the best and worst case runtime of the above three sorts?

- Selection:

  Worst case: $\theta(N^2)$. Selection sort performs a linear search of the smallest element in the unsorted part of the array, then swaps the elements at the current index and the smallest element's index. This process first takes N time to scan the array and find the smallest element, $N-1$ time for the second smallest element, $N-2$ time for the third smallest element, ... , until we have sorted the entire array. We observe an arithmetic sum starting to form: $\sum_{i=1}^{N} i = \theta(N^2)$.
  Best case: $\theta(N^2)$. Regardless of how close an array is to being completely sorted, selection sort will always search for the next smallest element in the array, so the best and worst case runtimes are equal.

- Heap:

Worst case: $\theta(NlogN)$. Bottom-up heapification takes performs N number of sink operations of $O(logN)$ time each for a total of $O(NlogN)$. Once heapified, removing the largest item and sinking takes at worst $logN$ time; for N removals, the total runtime is $O(NlogN)$. Our simplified final runtime can thus be tightly bounded to $\theta(NlogN)$.
Best case: $\theta(N)$. If all items of the array are the same value, bottom-up heapification is not necessary to perform and each removal from the max-heap will take constant time (no sinking is required). Thus, the total removal time of N items will be $\theta(N)$.

- Merge:

  Worst case: $\theta(NlogN)$. Mergesort splits the array into a binary tree of $logN$ layers. Within each layer, N number of item comparisons are performed, yielding a total runtime of $NlogN$.
  Best case: $\theta(NlogN)$. Mergesort performs the same number of comparisons the same number of times, regardless of the sorted status of the array, so the best and worst case runtimes are equal.

2 Asymptotics

2.1 Provide the overall runtime of method `f1` as a function of `N` and `M`. `randArr()` takes in an integer argument and returns an unsorted integer array of that length; assume it runs in constant time.

```java
public static int f1(int N, int M) {
    if (N == 1) {
        return N;
    }
    for (int i = 0; i < M; i++) {
        int[] arr = randArr(N);
        selectionSort(arr);
    }
    return f1(N / 2, 4 * M);
}
```

$\theta(MN^2logN)$. On the first iteration of the for-loop, selection sort is performed on `M` number of unsorted arrays of length `N`, each taking N^2 time to sort: this yields MN^2 on the first call of `f1`. On the next call, `4M` selection sorts are done on arrays of length `N/2`, each taking $N^2/4$ time to sort - this sums to MN^2. Each subsequent call of `f1` will also take MN^2 time until the base case is reached, which occurs in $logN$ time since `N` is halved each call. And because the best and worst case runtime of selection sort is the same, we can tightly bound the overall runtime to $\theta(MN^2logN)$.

2.2 Provide the overall runtime of method `f2` as a function of N, the length of the input integer array `arr`. `shuffle` shuffles the elements of an input integer array, doing so in N time.

```java
public static int f2(int[] arr) {
    if (arr.length == 1) {
        return ;
    }
    shuffle(arr);
    mergeSort(arr);

    # create array of half length
    int[] halfArr = new int[arr.length / 2];
    System.arrayCopy(arr, 0, halfArr, 0, halfArr.length);

    return f2(halfArr);
}
```

$\theta(NlogN)$. The first call to `f2` contains one call to `shuffle`, `mergeSort`, and `System.arrayCopy`, totaling to $N + NlogN + \frac{N}{2}$ time. Since $NlogN$ is slower than $N + \frac{N}{2}$, we can ignore the linear terms. Subsequent calls to `f2` will result in runtime $\frac{N}{2} + \frac{N}{2}log\frac{N}{2} + \frac{N}{4}$, $\frac{N}{4} + \frac{N}{4}log\frac{N}{4} + \frac{N}{8}$, ..., until our base case is reached. Again, we can ignore the linear part of each sum since the corresponding $Nlog$ term runs in slower time compared to the linear terms. Thus, our simplifed sum is $NlogN + \frac{N}{2}log\frac{N}{2} + \frac{N}{4}log\frac{N}{4} + ... + 4log4 + 2log2$, this is resemblant of the sum $N + N/2 + N/4 + ... + 4 + 2 + 1 \in \theta(N)$, which only cares about the largest term N. Thus, our sum simplifies to $\theta(NlogN)$ time.