Unintuitive, Merging Piles, Overlapping Events

Authors: Rafael Almeida, Yeo Bondar

1A

The largest $\frac{n}{3}$ elements will be in the last $\frac{2}{3}$ of the array. Since the first $\frac{2}{3}$ are sorted, if any of the largest $\frac{n}{3}$ elements were in the first $\frac{2}{3}$, then they must be in the second half of it. In other words, they must be in the middle $\frac{1}{3}$. If they were not in the first $\frac{2}{3}$, then they are in the last (unsorted) $\frac{1}{3}$. Either way, they are either in the middle or last $\frac{1}{3}$, meaning they are in the last $\frac{2}{3}$ of the array

1B

The largest $\frac{n}{3}$ elements will be in their correct sorted positions (in the last $\frac{1}{3}$ of the array). Since we showed that the largest $\frac{n}{3}$ elements were in the last $\frac{2}{3}$ of the array after `line 6`, after `line 7` sorts the last $\frac{2}{3}$ of the array, the largest $\frac{n}{3}$ elements must be in their sorted positions

1C

$$T(n) = 3T\left(\frac{2n}{3}\right) + \Theta(1)$$

$$= 3(3T\left(\frac{4n}{9}\right) + \Theta(1)) + \Theta(1)$$

$$= 9T\left(\frac{4n}{9}\right) + 4\Theta(n)$$

$$= 9(3T\left(\frac{8n}{27}\right) + \Theta(1)) + 4\Theta(1)$$

$$= 27T\left(\frac{8n}{27}\right) + 13\Theta(1)$$

general form

$$T(n) = 3^iT\left(\frac{2^in}{3^i}\right) + \Theta(1)\sum_{j=1}^{i} j^2$$

depth

$$n\left(\frac{2}{3}\right)^i = 1$$

$$n = \left(\frac{3}{2}\right)^i$$

$$i = \log_{\frac{3}{2}} n$$

plugging in

$$T(n) = 3^{\log_{\frac{3}{2}} n}T(1) + \Theta(1)\sum_{j=1}^{\log_{\frac{3}{2}} n} j^2$$

$$= T(n) = n^{\log_{\frac{3}{2}} 3} + \Theta(1)\sum_{j=1}^{\log_{\frac{3}{2}} n} j^2$$

since the second term is logarithmic, $n^{\log_{\frac{3}{2}} 3}$ is dominant

$$T(n) = \Theta(n^{\log_{\frac{3}{2}} 3})$$

2A

When merging two sorted piles, A and B, with sizes a and b respectively, where $a < b$ the number of comparisons are in the range $[a, a+b]$. To achieve best case number of comparisons, a items in A must be less than the least item in B. You will only compare the first a items in both lists and then A will be empty. Worst case scenario, you have to compare every item from A to every item in B and the merging process will only end when both lists are empty

There are k piles, each with $\frac{n}{k}$ items. Therefore, comparisons for the first merge are in the range $[\frac{n}{k}, \frac{2n}{k}]$. As we continue merging piles, until the final kth merge, the upper bound for comparisons increases, $[\frac{n}{k}, \frac{3n}{k}]$, $[\frac{n}{k}, \frac{4n}{k}]$...$[\frac{n}{k}, \frac{kn}{k} = n]$. However, assuming a completely random and uniform distribution of the ID numbers in all the piles, most combinations of two piles will result in a number of comparisons that are a constant c away from the worst case scenario (similar to the argument for the analysis of QuickSort that most pivots result in $nlgn$ time). Therefore, the running time of approach 1 is

$$\sum_{x=1}^{k}\left(\frac{x}{c}\cdot\frac{n}{k}\right)$$

$$=\frac{n}{ck}\sum_{x=2}^{k}x$$

$$=\frac{n}{ck}\cdot\left(k\frac{k+1}{2}-1\right)$$

$$=\frac{nk+n}{2c}-\frac{n}{ck}$$

$$=\Theta(nk)$$

2B

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$
$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \Theta\left(\frac{n}{2}\right)$$
$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \Theta\left(\frac{n}{4}\right)$$
$$T(n) = 2(2(2T\left(\frac{n}{8}\right) + \Theta\left(\frac{n}{4}\right)) + \Theta\left(\frac{n}{2}\right)) + \Theta(n)$$
$$= 8T\left(\frac{n}{8}\right) + 4\Theta\left(\frac{n}{4}\right) + 2\Theta\left(\frac{n}{2}\right)) + \Theta(n)$$
$$= 8T\left(\frac{n}{8}\right) + 3\Theta(n)$$
$$= 2^iT\left(\frac{n}{2^i}\right) + i\Theta(n)$$

recursion depth

$$\frac{n}{2^i} = 1$$
$$n = 2^i$$
$$lgn = i$$

substitute

$$T(n) = nT(1) + lgn\Theta(n)$$
$$= \Theta(nlgn)$$

2C

initially assuming the two methods are equivalent

$$nk = nlgn$$
$$k = lgn$$

if all values are equally likely, the average value of k is

$$\frac{1}{n}\sum_{x=1}^{n} k$$
$$= \frac{1}{n} \cdot \frac{n(n+1)}{2}$$
$$= \frac{(n+1)}{2}$$

it is now clear

$$\frac{(n+1)}{2} > lgn$$

asymptotically, method 2 is better

2D

```
HeapMerge(piles, n, k)
    heap = heapify(piles)

    mergedPile = []
    For i = 0 while i < n
            minList = peak heap
            minItem = remove first item from minList
            Add minItem to mergedPile
            HeapifyDown(heap, 0)

    Return mergedPile
```

1. Initialize a min heap with the given piles as lists and use the first item of each list as the key
2. Initialize an empty array `mergedPile`
3. For each item in all piles
    1. Remove the first item (`minItem`) from the list at the root of the heap (`minList`) and add it to `mergedPile`
    2. Re-heapify the heap
4. `mergedPile` is all items sorted

The algorithm uses a min heap to merge sorted linked lists into a single sorted list. The following three operations: peaking the list at the root of the heap, removing the smallest item from that list, then adding that item to the `mergedPile` is performed n times, where n is the total number of items in all piles. `HeapifyDown` ensures `heap` maintains the heap property and takes $O(logk)$ because there are k lists in `heap` and is also performed n times. Therefore, the overall time complexity of the algorithm is $T(n) = 3n + nlogk = \Theta(n) + O(nlogk) = O(nlogk)$

3A

If all students are in the room at the same time, there will be $\sum_{n=1}^{n-1} n = \frac{n(n-1)}{2}$ pairs. One student will form a pair with every other, creating $n-1$ pairs. The subsequent student will only create an extra $n-2$ unique pairs, then $n-3...1$

Using combinatorics, this is

$$
\begin{aligned}
&\binom{n}{2} \\
&= \frac{n!}{(n-2)! \cdot 2!} \\
&= \frac{n(n-1)}{2}
\end{aligned}
$$

3B

```
CountPairs(a, b, n)
        pairs = 0
        For int i = 0 while i < n
                For int j = i+1 while j < n
                        If a[i] <= b[j] and b[i] >= a[j]
                                pairs++

        Return pairs
```

1. Initialize a counter `pairs` to 0
2. For each student `i` in the array
   - For each subsequent student `j`, check if the arrival and departure times of `i` and `j` overlap
   - If they overlap, increment `pairs`
3. `pairs` is the total number of pairs formed

The algorithm uses two nested loops, resulting in a quadratic number of comparisons. The outer loop executes n times and the inner loops runs $n-1$, $n-2$, $n-3...1$. Specifically, for the first student, the algorithms does $n-1$ comparisons, then $n-2$ and so on. The number of comparisons will be similar to the sum given in `3A`, which is quadratic, giving $\Theta(n^2)$

3C

```
Event
        Attributes
                time
                type

CountPairs(a, b, n)
        events = []
        For int i = 0 while i < n
                Add Event(a, Arrival) to events
                Add Event(b, Departure) to events

        Sort events by time

        pairs = 0
        peopleInRoom = 0
        For event in events
                If event type is Arrival
                        peopleInRoom++

                If event type is Departure
                        peopleInRoom--
                        pairs += peopleInRoom

        Return pairs
```

3C Continuation

2. Create an array `events` to store `Event` objects, each having a `time` and `type`
3. For each student, add an `arrival` and `departure` event to `events`
4. Sort `events` by `time`
5. Initialize `pairs` and `peopleInRoom` to 0
6. For each `event` in `events`
    1. If `event` is `arrival`, increment `peopleInRoom`
    2. If `event` is `departure`, decrement `peopleInRoom` and increment `pairs` by `peopleInRoom`
7. `pairs` is the total number of pairs formed

The algorithm creates `Event` objects for each student's arrival and departure, sorts these events, and then counts the pairs of students in the room at the time of each departure. Creating the `Event` objects takes $\Theta(n)$ time, sorting takes $O(nlgn)$ time, and then counting the pairs takes $\Theta(n)$ time. The time complexity of the algorithm is $T(n) = \Theta(n) + O(nlgn) + \Theta(n) = O(nlgn)$

#homework