# Assignment 12, Data-Structures

Oleg Sivokon

*<2016-03-28 Mon>*

## Contents

# 1 Problems

## 1.1 Problem 1

Find tight bounds on the given recurrences. Assume $T(n)$ is constant for $n = 1$.

$$
\begin{aligned}
T(n) &= 8T\left(\frac{n}{2}\right) + n + n^3 \\
T(n) &= kT\left(\frac{n}{2}\right) + (k-2)n^3 \\
&\quad \textit{where } k \in \mathbb{Z} : k \geq 2 \\
T(n) &= 2T\left(\frac{n}{4}\right) + \sqrt{n} \times \lg n \\
T(n) &= T\left(n-1\right) + n \lg n + n \\
T(n) &= n^2\sqrt{n} \times T\left(\sqrt{n}\right) + n^5 \lg^3 n + \lg^5 n
\end{aligned}
$$

### 1.1.1 Answer 1

Using conclusion from master method theorem:

$$
\begin{aligned}
T(n) &= 8T\left(\frac{n}{2}\right) + n + n^3 \iff \\
f(n) &= n + n^3 \\
a &= 8 \\
b &= 2 \\
n^{\log_b a} = n^{\log_2 8} &= n^3 < n + n^3 \ .
\end{aligned}
$$

However, asymptotically, only the $n^3$ term matters, thus $T(n) = \Theta(n^3)$, viz. second case of master method.

### 1.1.2 Answer 2

$$
\begin{aligned}
T(n) &= kT\left(\frac{n}{2}\right) + (k-2)n^3 \\
&\quad \textit{where } k \in \mathbb{Z} : k \geq 2 \\
&= k\left(kT\left(\frac{n}{4}\right) + \frac{(k-2)n^3}{2}\right) + (k-2)n^3 \\
&= k^2T\left(\frac{n}{4}\right) + \frac{(k+2)(k-2)n^3}{2} \\
&= k^2\left(kT\left(\frac{n}{8}\right) + \frac{(k+2)(k-2)n^3}{8}\right) + \frac{(k+2)(k-2)n^3}{2} \\
&= k^3T\left(\frac{n}{8}\right) + \frac{(k^2+4)(k+2)(k-2)n^3}{8} \\
&= k^3T\left(\frac{n}{8}\right) + \frac{(k^2+4)(k^2-4)n^3}{8} \\
&\ldots \\
&= k^iT\left(\frac{n}{2^i}\right) + \frac{(k^i-2^i)n^3}{2^i}
\end{aligned}
$$

The recursion ends when $i = \lg n$, and at this point the $T(1)$ vanishes, and we are left with:

$$
\begin{aligned}
\frac{(k^i-2^i)n^3}{2^i} &= \left(\frac{k}{2}\right)^i n^3 - n^3 \\
&\approx \left(\frac{k}{2}\right)^i n^3 \\
&= \left(\frac{k}{2}\right)^{\lg n} n^3 \\
&\quad \textit{Using } 2^{\lg n} = n \\
&\approx n^4 \,.
\end{aligned}
$$

Thus $T(n) = O(n^4)$.

### 1.1.3 Answer 3

Using conclusion from master method theorem:

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{4}\right) + \sqrt{n}\lg n \iff \\
f(n) &= \sqrt{n}\lg n \\
a &= 2 \\
b &= 4 \\
n^{\log_b a} = n^{\log_4 2} &= \sqrt{n} < \sqrt{n}\lg n\ .
\end{aligned}
$$

This is the first case of master method, i.e. most work is done at the root (at each step we look at two of the four sub-problems). The amount of work done at each recursion step is $\sqrt{n} \times \lg n$ which is roughly the same as just $n$ since $\lg n = O(\sqrt{n})$. Hence $T(n) = O(n)$.

### 1.1.4 Answer 4

$$
\begin{aligned}
T(n) &= T(n-1) + n\lg n + n \\
&= \sum_{i=1}^{n}(i + i\lg i) \\
&= \sum_{i=1}^{n} i(1 + \lg i) \\
&= \frac{n(n+1)}{2} \times \sum_{i=1}^{n}(1 + \lg i) \\
&= \frac{n(n+1)}{2} \times \left(n + \sum_{i=1}^{n} \lg i\right) \\
&= \frac{n(n+1)(n + \lg n!)}{2} \\
&\textit{using Stirling approximation} \\
&\approx \frac{n(n+1)(n + n\lg n - n)}{2} \\
&= \frac{n^2 \lg n(n+1)}{2} \\
&\approx \frac{n^3 \lg n}{2}\ .
\end{aligned}
$$

Since constant factors are of no interest to us, we conclude $T(n) = O(n^3 \lg n)$.

### 1.1.5 Answer 5

$$
\begin{aligned}
T(n) &= n^2\sqrt{n}T\left(\sqrt{n}\right) + n^5 \lg^3 n + \lg^5 n \\
&= n^{\frac{5}{2}}T\left(n^{\frac{1}{2}}\right) + n^5 \lg^3 n + \lg^5 n \\
&= n^{i\frac{5}{2}}T\left(n^{\frac{1}{i2}}\right) + \sum_{j=1}^{i}(n^5 \lg^3 n + \lg^5 n)^{\frac{1}{j}} \\
&\textit{Recursion terminates when } n^{\frac{1}{i2}} = 1 \ .
\end{aligned}
$$

## 1.2 Problem 2

Find upper and lower bounds on:

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + n^3 \\
T(n) &= T(\frac{9n}{10}) + n \\
T(n) &= 16T(\frac{n}{4}) + n^2 \\
T(n) &= 7T(\frac{n}{3}) + n^2 \\
T(n) &= 7T(\frac{n}{2}) + n^2 \\
T(n) &= 2T(\frac{n}{4}) + \sqrt{n} \\
T(n) &= T(n-1) + n \\
T(n) &= T(\sqrt{n}) + 1
\end{aligned}
$$

#### 1.2.1 Answer 6

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + n^3 \\
&\quad \textit{Using master method} \\
a &= 2 \\
b &= 2 \\
f(n) &= n^3 \\
n^{\log_2 2} &= n < n^3 \\
&\quad \textit{Third case of master method, hence} \\
T(n) &= \Theta(f(n)) = \Theta(n^3)
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= T(\frac{9n}{10}) + n \\
T(n) &= 9T(\frac{n}{10}) + n \\
&\quad \textit{Using master method} \\
a &= 9 \\
b &= 10 \\
f(n) &= n \\
n^{\log_9 10} &\approx n^{1.05} \approx n \\
&\quad \textit{Second case of master method, hence} \\
T(n) &= \Theta(n^{1.05} \lg n) \approx \Theta(n \lg n)
\end{aligned}
$$

$$T(n) = 16T(\frac{n}{4}) + n^2$$
$$\textit{Using master method}$$
$$a = 16$$
$$b = 4$$
$$f(n) = n^2$$
$$n^{\log_4 16} = n^2 = n^2$$
$$\textit{Second case of master method, hence}$$
$$T(n) = \Theta(n^2 \lg n)$$

$$T(n) = 7T(\frac{n}{3}) + n^2$$
$$\textit{Using master method}$$
$$a = 7$$
$$b = 3$$
$$f(n) = n^2$$
$$n^{\log_3 7} \approx n^{1.8} \approx n^2$$
$$\textit{Second case of master method, hence}$$
$$T(n) = \Theta(n^2 \lg n)$$

$$T(n) = 7T(\frac{n}{2}) + n^2$$
$$\textit{Using master method}$$
$$a = 7$$
$$b = 2$$
$$f(n) = n^2$$
$$n^{\log_2 7} \approx n^{2.8} > n^2$$
$$\textit{First case of master method, hence}$$
$$T(n) = \Theta(n^{2.8}) \approx \Theta(n^3)$$

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{4}) + \sqrt{n} \\
T(n) &= 2T(\frac{n}{4}) + n^{\frac{1}{2}} \\
&\quad \textit{Using master method} \\
a &= 2 \\
b &= 4 \\
f(n) &= n^{\frac{1}{2}} \\
n^{\log_4 2} &= n^{\frac{1}{2}} = n^{\frac{1}{2}} \\
&\quad \textit{Second case of master method, hence} \\
T(n) &= \Theta(\sqrt{n}\lg n)
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= T(n-1) + n \\
&\quad \textit{Suppose} \\
T(1) &= 1 \\
&\quad \textit{Then} \\
T(n) &= T(1) + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} \approx n^2 \\
T(n) &= \Theta(n^2)
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= T(\sqrt{n}) + 1 \\
T(n) &= T(n^{\frac{1}{2}}) + 1 \\
&= T(n^{\frac{1}{4}}) + 1 + 1 \\
&= T(n^{\frac{1}{8}}) + 1 + 1 + 1 \\
&= T(n^{\frac{1}{\lg i}}) + i \\
&= \lg n \\
&= \Theta(\lg n)
\end{aligned}
$$

## 1.3 Problem 3

Suggest a data-structure with the following properties:

1. Populate in $O(n)$ time.
2. Insert in $O(n \lg n)$ time.
3. Extract minimal element in $O(\lg n)$ time.
4. Extract median element in $O(\lg n)$ time.
5. Extract maximal element in $O(\lg n)$ time.

### 1.3.1 Answer 6

There is a simple, but impractical way of doing this—have four heaps:

- $A$ is a `min-heap` containing elements greater than median.
- $B$ is a `max-heap` containing elments smaller than median.
- $C$ is a `max-heap` tracking the heap $A$.
- $D$ is a `min-heap` tracking the heap $B$.

Creation and insertion are essentially the same as they are in the regular `max-heap` and `mean-heap`. Median element is either the root of $A$ or the root of $B$, depending on which heap has more elements. Maximum element is the root of $C$ and minimal element is the root of $D$, so their extraction is just the glorified `extract-max` and `extract-min` correspondingly.

Tracking is achieved using the following mechanism: Each node in each heap has an additional field that has a position of the tracked node in the other heap in it. Once the position of the node is modified, in addition to `heapify-min` or `heapify-max`, the procedure also updates the index in the tracking node (this takes only constant time).

Whenever a node is deleted, it also needs to be deleted from the tracking heap. In this case, the rightmost element in the heap is placed in the cell previously occupied by the node being deleted. Then `heapify-min` or `heapify-max` is performed, depending on the kind of heap it was.

Note that this solution is impractical since it requires saving a lot of additional information, but if we were to relax the requirement of $O(n)$ allowing $O(n \lg n)$ for population, then we could use something like order-statistic tree.

## 1.4 Problem 4

1. Given binary heap $A$ of size $n$ prove that `extract-max` requires roughly $2 \lg n$ comparisons.
2. Write an alternative `extract-max` which only uses $\lg n + \lg \lg n + O(1)$ comparisons.
3. Improve the previous `extrac-max` s.t. its running time is $\lg n + \lg \lg \lg n + O(1)$ wrt. comparisons.
4. Is it possible to improve this procedure further? Is it worth it wrt. the amount of code that it requires?

### 1.4.1 Answer 7

First, recall what `extract-max` looks like:

The comparisons all happen inside the `heapify-max`, notice that it is called recursively on the problem of size $n$, splitting it into two equally-sized portions, and only working on the selected subtree. It will only look once at a node at the $h^s$ level $h$ being the heights of the heap. At each such level it will do five comparisons: two to ensure that all reads fall within the valid range, two more to find the maximal element of the parent and its two sibling nodes, and the last one to figure out whether an additional `heapify-max` call is required.

**Algorithm 1** Running time of extract-max

**procedure** *extract-max*($heap$)
    $max \leftarrow heap_0$
    $size \leftarrow size(heap)$
    $last \leftarrow heap_{size-1}$
    $heap_{size-1} \leftarrow nil$
    *heapify-max*($heap, size - 1$)
    **return** $max$
**end procedure**
**procedure** *heapify-max*($heap, child$)
    $left \leftarrow child * 2 - 1$
    $right \leftarrow child * 2$
    $parent \leftarrow child$
    $size \leftarrow size(heap)$
    **if** $left < size \wedge heap_{left} > heap_{parent}$ **then**
        $parent \leftarrow left$
    **end if**
    **if** $right < size \wedge heap_{right} > heap_{parent}$ **then**
        $parent \leftarrow right$
    **end if**
    **if** $parent \neq child$ **then**
        $heap_i, heap_{parent} \leftarrow heap_{parent}, heap_i$
        *heapify-max*($heap, parent$)
    **end if**
**end procedure**

Thus, somewhat contrary to conjectured, the number of comparisons required is actually $5 \lg n$, but only $2 \lg n$ of them are between the members of the heap (the rest is borders checking).

### 1.4.2 Answer 8

The idea is borrowed from Gonnet and Munro:

> Observe that the elements on the path from any node to the root must be in sorted order. Our idea is simply to insert the new element by performing the binary search on the path from location $n+1$ to 1. As this path contains $\lceil \log(n+1) \rceil$ old elements the algorithm will require $\lceil \log(\lceil 1 + \log(n+1) \rceil) \rceil = \lceil \log(\log(2+1)) \rceil$ comparisons in the worst case. We note that the number of moves will be the same as those required in carefully coded standard algorithm.

### 1.4.3 Answer 9

Again, quoting Gonnet and Munro:

> This bound can, however, be improved as follows. For simplicity assume we are removing the maximum and simultaneously inserting a new element.

```
Remove the maximum, creating a "hole" at the top of the heap.

Find the path of the maximum sons down r levels to some location, say A(i)

If New element > A(i) Then
  Perform perform a binary search with the new element along the path of
  length r
Else
  Promote each element on the path to the location of its father and
  recursively apply the method starting at the location A(i).
```

### 1.4.4 Answer 10

The questions of "practical usefulness" are very subjective. The answer will depend on multiple factors and the ability to predict the future in minor details. However, it has been shown many times since Williams, Gonnet and Munro, Carlsson and many others who worked on optimizing priority queues, that binary heaps aren't the best choice of the data-structure for this purpose. Typical requirement for a binary queue is that it perform an `insert` in constant time, this already disqualifies binary heaps, where one can only hope for amortized constant time.

The reality of working with large datasets are such that continuous arrays are difficult to allocate and access. Persistency becomes increasingly important and so does concurrency. Binary heaps implemented as arrays don't fare very well in this emerging market, so the question of this specific optimization is rather pointless.