*EE360C: Algorithms*
University of Texas at Austin — Homework #3
Dr. David Soloveichik — Due: September 27, 2018 (*in-class quiz*)

# Homework #3

**You do not need to turn in these problems. The goal is to be ready for the in-class quiz that will cover the same or similar problems, and to prepare you for the exams.**

## Problem 1: Algorithm Analysis

Consider the following basic problem. You're given an array $A$ consisting of $n$ integers $A[1], A[2], \ldots, A[n]$. You'd like to output a two-dimensional $n$-by-$n$ array $B$ in which $B[i,j]$ (for $i < j$) contains the sum of array entries $A[i]$ through $A[j]$—that is, the sum $A[i]+A[i+1]+\cdots+A[j]$. (The value of array entry $B[i,j]$ is left unspecified whenever $i \geq j$, so it doesn't matter what is output for these values.) Below is a simple algorithm to solve this problem:

```
1  for $i = 1$ to $n$
2      for $j = i + 1$ to $n$
3      Add entries $A[i]$ through $A[j]$
4      Store the result in $B[i, j]$
```

**(a)** Give a function $f(n)$ that is an asymptotically tight bound on the running time of the algorithm above. Using the pseudocode above, argue that the algorithm is, in fact $\Theta(f(n))$.

**Solution**

$\Theta(n^3)$. Why? The two for loops each run on the order of $n$ times. Adding $\Theta(n)$ entries takes $\Theta(n)$ time. Therefore the total time is $\Theta(n^2) * \Theta(n)$ or $\Theta(n^3)$.

**(b)** Although the algorithm you analyzed above is the most natural way to solve the problem, it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem with an asymptotically better running time than the provided algorithm.

**Solution**

Notice that the original algorithm repeats the same additions multiple times. For any $i = k : 1 \leq k \leq n-1$ and $j = l : k+1 < l \leq n$, when we get to the step of finding the sum of entries $A[k]$ through $A[l]$, instead of naively making l-k additions, we can just get the result with one addition: $B[k, l-1]$, which has already been calculated in the previous step, plus $A[l]$:

```
1  for $i = 1$ to $n$
2      for $j = i + 1$ to $n$
3          $B[i, j] = B[i, j-1] + A[j]$
```

**(c)** What is the running time of your new algorithm?

**Solution**

$\Theta(n^2)$ Again, two for loops each run on the order of $n$ times, but this we only need $O(1)$ time to perform a single addition.

## Problem 2: Try sorting

Given a set $A$ of $n$ distinct positive integers and another integer $t$, describe an algorithm that determines whether or not there exists two elements in $A$ such that their product is exactly $t$. Come up with an $O(n \log n)$ algorithm to solve this problem. Hint: When you see a runtime target that looks familiar, try to think of other algorithms you can use as subroutines in your algorithm with the same time complexity to make your life easier. In this case, try sorting the list first.

**Solution**

Sorting takes $O(n \log n)$ time. Then we have an $O(\log n)$ operation (binary search among $O(n)$ sorted elements) and a constant amount of $O(1)$ operations, repeated $O(n)$ times in the loop. Thus the complexity of the algorithm is $O(n \log n) + n * O(\log n) + O(1) = O(n \log n)$.

```
1  Sort elements of A
2  for $i = 1$ to $n - 1$
3      $k = t/A[i]$
4      $j$ = BinarySearch($k$,$A[i+1:n]$)
5      if $j \neq -1$
6          return 'yes'
7  return 'no'
```

## Problem 3: Algorithms and decision trees

You are given 9 identical looking balls and told that one of them is slightly heavier than the others. Your task is to identify the defective ball. All you have is a balanced scale that can tell you which of two sets of balls is heavier.

**(a)** Show how to identify the heavier ball in just 2 weighings.

**(b)** Give a decision tree lower bound showing that it is not possible to determine the defective ball in fewer than 2 weighings.

**Solution**

(a) Enumerate the balls as $\{1, 2, \ldots, 9\}$. Measure $A = \{1, 2, 3\}$ and $B = \{4, 5, 6\}$. If $A$ is heavier, $A$ must contain the heavier ball. If $A$ is lighter, $B$ must contain the heavier ball. If $A$ has the same weight as $B$, then the heavier ball is in $C = \{7, 8, 9\}$. Therefore we know which set has the heavier ball; without loss of generality, let it be $A$. Now measure balls 1 and 2. If 1 is heavier, it is the heavy ball; if 1 is lighter, 2 is the heavier ball; if 1 and 2 have the same weight, then 3 is the heavier ball. If you think of this as a general algorithm, what is the (asymptotic) number of weighs you must make to find the heavier ball given $n$ balls?

(b)Any comparison in this tree has three possible outcomes; one of the two sets being weighed is heavier, or they have the same weight. Then each node in our decision tree has three children. A tree with branching factor 3 and height $h$ has at most $3^h$ leaves. Therefore $h$ must be at least 2 to have 9 leaves, so we must make at least two weighings.

## Problem 4: Heap algorithms

Given $k$ sorted lists, merge them into one sorted list in $\mathcal{O}(n \log k)$ time where $n$ is the sum of the lengths of all lists (i.e., the total number of elements in the input).

**Solution**

Remove the first element from each list and insert them into a min-heap $H$. Let $L$ be a list to store the merged sorted list. Until all input lists are empty, repeat: let $m$ be the result of extract min from $H$; delete $m$ from $H$, add $m$ to $L$, and add to the heap the next number from the list that originally contained $m$, and remove it from its list (note: in an implementation, when storing values in the heap, you must store a tuple $(a, b)$ where $a$ is the value and $b$ is a pointer to the list which $a$ came from). Note that $H$ does not exceed size $k$ because an element is removed before a new one is added.

**Runtime**: The initial construction of the heap takes $\mathcal{O}(k \log k)$ time using insert (or, since it is a new heap, $\mathcal{O}(k)$ time using the more advanced heap construction algorithm). There are $n$ total elements, the loop eliminates one element from the input lists in each iteration, and the loop halts when all $n$ elements have been removed, so the loop iterates $n$ times. Note that the size of the heap does not exceed $k$ since it starts at size $k$ and an element is deleted before next is added. Each iteration of the loop does the following: extract min ($\mathcal{O}(1)$), delete ($\mathcal{O}(\log k)$), insert ($\mathcal{O}(\log k)$), and deletes first element from a list ($\mathcal{O}(1)$). So the runtime of each iteration is $\mathcal{O}(\log k)$, and the loop iterates $n$ times, so the total runtime is $\mathcal{O}(n \log k)$.

**Proof of correctness**[a] Let $L$ be the final merged sorted list. We will prove $L$ is sorted. Let $L_d$ be the input list which originally contained element $d$. For each $L[i] = a, L[j] = b$ with $i < j$, there are two cases. We will show in both cases that $a < b$.

Case 1: $a$ came from the same input list as $b$; i.e., $L_a = L_b$. $a$ comes before $b$ in $L$, so $a$ was inserted into $L$ before $b$ was removed from $L_b$, and $L_b$ was sorted, so $a < b$.

Case 2: $a$ came from a different input list than $b$; i.e., $L_a \neq L_b$. Let $c$ be the element from $L_b$ which was in the heap when $a$ was moved from the heap to $L$ (note that it is possible $c = b$). Since $a$ was removed via extract min while $c$ was in the heap, $a < c$. Since $b$ and $c$ originated from $L_b$, and there is only one element per input list in the heap at any time, if $c \neq b$ then $b$ was not in the heap when $a$ was added to $L$. Further, since $b$ comes after $a$ in $L$, then $b$ was not in $L$ when $a$ was added to $L$. Since $b$ was not in $L$ and not in the heap, $b$ was in $L_b$. Since $c$ and $b$ are both from $L_b$, and $c$ was inserted into the heap before $b$, and $L_b$ is sorted, then $c < b$. By transitivity, since $a < c$ and $c < b$, $a < b$.

Then every pair $L[i] = a, L[j] = b$ with $i < j$ in $L$ satisfy $a < b$, so $L$ is sorted.

[a] Proof of correctness is not required on quizzes or exams **unless you are explicitly asked to provide a proof of correctness**. Nonetheless, it is good to practice proofs of correctness.

## Problem 5: Median-heap data structure

Design a "Median-heap" data structure, which supports find-median operation in $\mathcal{O}(1)$ time and insert/delete operations in $\mathcal{O}(\log n)$ time. Recall the definition of median (intuitively, the median of a list is the middle element when that list is sorted). If the list has even length, you can choose the larger of the two middle elements as the median.

**Solution**

Construct two heaps, a max-heap $M$ and a min-heap $m$. $M$ will store the "smaller half" of our data, and $m$ will store the "larger half". We will maintain the following median-heap property: $||m| - |M|| \leq 1$. This way, depending on the relative sizes of the two heaps, the median will be given by find max from $M$ or find min from $m$; e.g., if $M$ has five elements and $m$ has four, then the median is find max $M$. The operations are as follows:

- Find median: if $|M| = |m|$ or $|M| = |m| + 1$, return find max from $M$; otherwise by our median-heap property $|M| = |m| - 1$, so return find min from $m$.
- Resize: If $|M| > |m| + 1$, find max from $M$, delete it from $M$, and insert it into $m$. Else, if $|m| > |M| + 1$, find min from $m$, delete it from $m$, and insert it into $M$.
- Insert $a$: let $x$ be find median. If $a \geq x$, $a$ belongs in the "larger half" of the data, so insert $a$ into $m$. Otherwise, insert $a$ into $M$. Call resize.
- Delete $a$: let $x$ be find median. If $a = x$, delete $a$ from $M$ or $m$ depending on which contains the median; else if $a < x$, delete $a$ from $M$; else if $a > x$, delete $a$ from $m$. Call resize.

Calling resize after insertion and deletion maintains the median-heap property. Let $n$ be the total number of elements in the median-heap (i.e., $|m| + |M|$). Extract median does one comparison and runs either find max or find min from $M$ or $m$, each of which takes $\mathcal{O}(1)$ time, thus find median takes $\mathcal{O}(1)$ time. Resize does one comparison, calls find max or find min from $M$ or $m$ ($\mathcal{O}(1)$), deletes from $m$ or $M$ ($\mathcal{O}(\log n)$), and inserts into $M$ or $m$ ($\mathcal{O}(\log n)$), so in total the time is $\mathcal{O}(\log n)$. Insert/delete each do one comparison followed by an insertion or deletion from $m$ or $M$ ($\mathcal{O}(\log n)$), followed by a call of resize ($\mathcal{O}(\log n)$) for a total time of $\mathcal{O}(\log n)$.