COSC 302: Analysis of Algorithms Lecture — Spring 2018
Prof. Darren Strash
Colgate University

Worksheet 4 — Divide and Conquer II and Average-Case Analysis (with solutions)

1. You are again hard at work doing quality assurance testing at the pinePhone factory. Suppose you now have two pinePhones. Can you find the highest safe rung by dropping the pinePhones asymptotically fewer times than $\Theta(n)$ (the number of drops for 1 pinePhone)? How many drops do you make with your algorithm in the worst case?

   We can decompose the ladder into $\sqrt{n}$ sections, each of which has $\sqrt{n}$ rungs. We drop the first pinePhone from the bottommost rung, then drop from rungs $i\sqrt{n}$ for $i = 1, 2, 3, \ldots \sqrt{n}$ until the pinePhone breaks. When it breaks, we are guaranteed that highest safe rung is within the $\sqrt{n}$ rungs below where the first pinePhone broke. We then test each one of the rungs from bottom to top using the iterative algorithm for one pinePhone. In total, we perform $2\sqrt{n} - 1 = \Theta(\sqrt{n})$ drops in the worst case.

2. Suppose you are given n numbers in an array A, divided into $\lceil n/\lg n \rceil$ equal-sized buckets $A_1, A_2, \ldots A_{\lceil n/\lg n \rceil}$, which are stored in order in an array. Further suppose that $\forall a \in A_i$, $\forall b \in A_{i+1}$, $a < b$, but we do not know the order of elements in the same bucket A_i. How fast can you search for a given element in the collection?

   It is possible to perform a binary search over the buckets in time $O(\lg n)$, which narrows the search to a constant number of buckets. From which we can perform a linear search in time $O(n/\lceil n/\lg n \rceil) = O(\lg n)$. See Algorithm 2.

```
BUCKET-SEARCH($A[1..n]$, $k$)
1: $p = 1$ // First bucket
2: $r = \lceil n/\lg n \rceil$ // Last bucket
3: bucket-size = $n/\lceil n/\lg n \rceil$
4: while $r - p > 1$
5:     mid-bucket = $\lfloor \frac{r+p}{2} \rfloor$
6:     bucket-element-index = mid-bucket $\cdot$ bucket-size // Last element in mid-bucket
7:     if $A$[bucket-element-index] $< k$ then
8:         $p$ = mid-bucket
9:     else if $A$[bucket-element-index] $> k$ then
10:        $r$ = mid-bucket
11:    else
12:        return true // Found $k$ in bucket mid-bucket
13: // Search through remaining values in buckets $p$ and $r$
14: found-k = LINEAR-SEARCH($A$, $(p-1) \cdot$ bucket-size $+ 1$, $r \cdot$ bucket-size)
15: return found-k
```

Solution continued on next page →

Correctness of this algorithm is based on the following loop invariant:

Invariant 1. *At the beginning of the while loop on line 2, k is either not in A, or k is in buckets $p...r$ (in subarray $A[(p-1)\cdot \textit{bucket-size}+1\ldots r\cdot \textit{bucket-size}]$).*

Initialization: Either k is in the $A[1..n]$ or it isn't. This is always true.

Maintenance: We assume that k is in A, otherwise k is never in the range and the invariant is trivially true. We therefore show maintenance for the case that k is in the array. At the beginning of the while loop. Then at the beginning of the loop k is in the range buckets p to r (stored in $A[(p-1)\cdot \text{bucket-size}+1\ldots r\cdot \text{bucket-size}]$). After comparing an element from bucket mid-bucket, then k is either in the current bucket or it is to the right or left. Without loss of generality, suppose the element compared from bucket mid-bucket is less than k. Then k cannot be in buckets mid-bucket$+1$ to r, since these contain values that are strictly greater than k. Therefore, k must be in buckets p to mid-bucket. Since $r =$ mid-bucket in the next loop, k is in buckets $p..r$ (stored in subarray $A[(p-1)\cdot \text{bucket-size}+1\ldots r\cdot \text{bucket-size}]$).

Termination: The loop either terminates at line 12, in which case k is found, or on line 15, in which case the value k is in the range (by the invariant) (and found on line 14) or not in the array and therefore is not found on line 14.

3. (a) Suppose you are given an array $A[1..n]$ with 3 inversions. Can you sort A asymptotically faster than $\Theta(n \lg n)$ using a comparison sort? If so, describe an algorithm.

   Solution: Yes, running insertion sort on the data will take $\Theta(n)$ time in the worst case. $\Theta(n)$ time to first look at each element and decide if that element is in order and $\Theta(1)$ time to test and swap the elements involved in each inversion.

   (b) Suppose you change the base case of merge sort to run insertion sort when the array contains 10 or fewer elements. Does the asymptotic running time of merge sort change? *Hint: Come up with a recurrence, and solve it.*

   Solution: The asymptotic running time does not change, it is still $\Theta(n \lg n)$ in the worst case. We can see this in several ways. First: Running merge sort on a data set of size 10 takes time $\Theta(10 \lg 10) = \Theta(1)$ time. Running insertion sort on the same data takes $\Theta(10^2) = \Theta(1)$ time. Therefore, changing one for the other in merge sort only changes constant factors. Second: Look at the recursion tree. By stopping recursion when each subarray has 10 elements, we are only removing $\lg 10$ levels from the recursion tree. The recursion tree then has depth $\Theta(\lg n - \lg 10) = \Theta(\lg n)$. We are still doing a linear number of operations for each level in the recursion tree, taking $\Theta(n \lg n)$ time overall.

   (c) Define a *crossing* inversion to be the number of inversions between the left and right halves of an array. That is, given an array $A[1..n]$, crossing inversions only count inversions formed between indices i and j, where $i \in \{1, 2, \ldots, n/2\}$, and $j \in \{n/2+1, \ldots, n\}$.

      i. How fast can you sort an array $A[1..n]$ that has *only* crossing inversions and no other inversions? Describe an algorithm to sort with this running time.

      Solution: Since there are only inversions between the two halves, both the left and the right halves are themselves sorted. This is precisely the situation where we merge in merge sort. We can thus sort in worst-case $\Theta(n)$ time by merging the two halves of A into a new array B, then copying B back into A.

ii. Suppose that you are designing an algorithm to count the number of crossing inversions. You decide to use merge sort as a template, but you will only count the number of crossing inversions in each subarray—instead of merging or sorting. Consider the following questions:

A. Suppose that, for each recursive call of your merge-sort-like algorithm, its subarray $A[i..j]$ has 1 crossing inversion. Give an example input with 8 elements where this is the case. Under this restriction, how many *total* inversions will an n-element input array have?

Solution:

[2,0,4,1,6,3,7,5]

The total number of inversions T is equal to the number of crossing inversions C over all these subarrays. This can be seen by a two-part proof.

Proof. We show that $T \geq C$ and $T \leq C$. If both of these inequalities hold, then $T = C$, as T cannot be simultaneously smaller *and* larger than C.
Case 1: $T \geq C$

Notice that each crossing inversion, counted in C, is itself an inversion. Therefore, each such crossing inversion is counted in the total number of inversions, and therefore $T \geq C$.

Case 2: $T \leq C$

We show that an arbitrary inversion (which is counted in T) is counted in C. Let the pair of indices (i, j) be some inversion. Since we are recursively subdividing the array A until it reaches subarrays of single elements, there are two consecutive levels k and $k+1$ in the recursion tree, where indices i and j are in the same subarray at level k, and then in two different subarrays at level $k+1$. At level k, when they are part of the same subarray, (i, j) is a crossing inversion, as i is in the first half of the subarray, and j is in the second half. □

Further, there is a linear number of such inversions:

$$\sum_{i=0}^{\lg n - 1} 2^i = 2^{\lg n} - 1 = n - 1.$$

Which is derived by looking at the recursion tree for merge sort, and adding up the crossing edges at each level—and noticing that the last level, which only has subarrays of size 1 contains no crossing edges.

B. Suppose you are given an n-element input array with the properties described in A. Can you make simple adjustments to merge sort to sort such an input array asymptotically faster than $\Theta(n \lg n)$? *Hint: given a subarray with 1 inversion, where can the inverted elements be in the subarray?*

   Solution: Note that it is possible to sort in linear time with insertion sort, but now we can also do linear time sorting with merge sort. Instead of doing a full merge step, it is sufficient to swap the elements within 1 index from the middle. We maintain the invariant that after returning from a recursive call, the crossing inversion can only be between these two elements. Then, in order to sort, we swap these elements in the combine step. This takes constant time to sort each subarray. Thus, we get the recurrence $T(n) = 2T(n/2) + \Theta(1)$, which solves to $T(n) = \Theta(n)$.

4. Suppose you are given an array $A[1..n]$ of distinct integers in the range $a..b$ (inclusive). We say that an integer is *missing* from A if it is in the range $a..b$ and not present in array A.

   (a) Give pseudocode for a worst-case $O(1)$-time algorithm to determine how many integers in $a..b$ are missing from A.

   Solution:

```
NumMissing($A$, $a$, $b$)
1: // number of elements between $a$ and $b$ inclusive,
2: // minus the number of elements present in $A$
3: return $b - a + 1 - A$.length
```

   (b) Briefly describe, and give pseudocode for, an efficient algorithm to compute the smallest integer missing from A. *Hint: Your algorithm should have worst-case running time $O(n)$.*

   Solution: First note that it would be simple to solve the problem if the input array was first sorted. Thus there is a simple $\Theta(n \lg n)$ time algorithm: sort the array, then iterate through and check for the first value missing (either a is missing, or find the first value such that x is present, but $x + 1$ is not).

   However, since sorting in linear-time is prohibited by the problem, let's try to solve using a divide and conquer strategy. In particular, if we could divide the array in half and recursive on one of the halves, then we would get a recurrence

$$T(n) \leq T(n/2) + \Theta(n)$$

   for the running time, which solves to $O(n)$.

   Using linear-time selection as a tool, we can compute the median element (at index $n/2$) in A by calling Select(A, 1, n, $n/2$). We can then partition A on this element and if the left half is missing an element, recurse on it. If the left half is not missing an element, but the right half is missing an element, then we recurse on the right half. We do this until we are left with a constant number of elements, which can be checked in constant time. See rough pseudocode on next page.

Continued on next page →

```
SmallestMissing($A$, $p$, $r$, $a$, $b$)
numleft = $p - r + 1$
// If the number left is constant, check if $a$, $a+1$, $\ldots$ are present, and return smallest
if numleft $\leq 3$ then
    for $i = 0$ to numleft
        if $(a+i) \notin A[p..r]$ then
            return $(a+i)$

// Otherwise, we need to divide and conquer.
pivot = Select($A$, $p$, $r$, $(p+r)/2$) // Get median element between indices $p$ and $r$
swap pivot $\leftrightarrow A[r]$
$q$ = Partition($A$, $p$, $r$)
if NumMissing($A[p..q]$, $a$, pivot) $\geq 1$ then // Recurse on left half
    return SmallestMissing($A$, $p$, $q$, $a$, pivot)
else// Recurse on left half
    return SmallestMissing($A$, $q$, $r$, pivot, $b$)
```

The final solution is found by calling SmallestMissing(A, 1, n, a, b).

5. Iterated functions and the tower of twos.

   (to be discussed in a future recitation.)

6. *(Rolling a fair die)*

   (a) What is the expected value of rolling a fair die one time?

   Solution: The sample space of a die is $\{1, 2, 3, 4, 5, 6\}$, and since the die is fair, the probability of getting a given value is equal, $1/6$.

   Let X be the value of rolling a die, then the expected value of X is

$$\mathrm{E}[X] = \sum_{D=1}^{6} D \cdot \Pr\{X = D\} = \sum_{D=1}^{6} \frac{D}{6} = \frac{1}{6} \sum_{D=1}^{6} D = \frac{1}{6} \cdot \frac{6(6+1)}{2} = \frac{7}{2} = 3.5.$$

   (b) Compute the expected number of 3's you get when rolling a fair die one time. Be sure to use an indicator random variable.

   Since we want to count the number of 3's, we need an indicator that counts 1 if we get a 3, and 0 otherwise. Let D be the value of rolling a die. Then we create an indicator random variable X_D as follows.

$$X_D = I\{D = 3\} = \begin{cases} 1 & D = 3, \\ 0 & \text{otherwise.} \end{cases}$$

   Let X be the number of 3's rolled. Then the *exact* number of times a 3 is rolled is

$$X = X_D.$$

   We compute the expected value of X as

$$\mathrm{E}[X] = \mathrm{E}[X_D] = \sum_{i=1}^{6} X_i \cdot \Pr\{D = i\} = 1 \cdot \Pr\{D = 3\} = \frac{1}{6}.$$

7. *(Sometimes insertion sort)* You believe that you can devise a randomized sorting algorithm that has expected-time $\Theta(n \lg n)$ by randomly selecting whether or not to run insertion sort on the input array $A[1..n]$.

   (a) Suppose you you design an algorithm that, with probability 1/2, calls merge sort, and with probability 1/2, calls insertion sort. Compute the expected running time of your algorithm.

   (to be discussed in a future recitation.)

   (b) What if the probabilities change to $\frac{n-\lg n}{n}$ for merge sort, and $\frac{\lg n}{n}$ for insertion sort?

   (to be discussed in a future recitation.)