HOMEWORK 2

Charles Haithcock, `cehaith2`

CSC 505 - Design and Analysis of Algorithms
Steffen Heber

Due: 27 February 2017

Question 1 (2 pts)

Solve Problem 7-4, a-c on page 188 of our textbook (quoted below)

> The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the left subarray and then it recursively sorts the right subarray. The second recursive call in QUICKSORT is not really necessary; we can avoid it by using an iterative control structure. This technique, called ***tail recursion***, is provided automatically by good compilers. Consider the following version of quicksort, which simulates tail recursion:
>
> TAIL-RECURSIVE-QUICKSORT(A, p, r)

```
1        while p < r
2            // Partition and sort left subarray.
3            q = PARTITION(A, p, r)
4            TAIL-RECURSIVE-QUICKSORT(A, p, q - 1)
5            p = q + 1
```

> ***a.*** Argue that TAIL-RECURSIVE-QUICKSORT(A, 1, $A.length$) correctly sorts the array A.
>
> Compilers usually execute recursive procedures by using a ***stack*** that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. Upon calling a procedure, its information is ***pushed*** onto the stack; when it terminates, its information is ***popped***. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The ***stack depth*** is the maximum amount of stack space used at any time during a computation.
>
> ***b.*** Describe a scenario in which TAIL-RECURSIVE-QUICKSORT's stack depth is $\Theta(n)$ on an n-element input array.
>
> ***c*** Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is $\Theta(\log_2 n)$. Maintain the $O(n \log_2 n)$ expected running time of the algorithm.

Purpose Often, recursive function calls use up precious stack space and might lead to stack overflow errors. Tail call optimization is one method to avoid this problem by replacing certain recursive calls with an iterative control structure. Learn how this technique can be applied to QUICKSORT.

Answer 1.a

For the modification to the algorithm, the change is fundamentally minimal; even though a recursive call was taken out, the control structure ends up calling the `tail_recursive_quicksort` recursive call on an implicity "left" and "right" subarray.

As defined in the text, `partition` chooses a pivot point, the index of which is returned as q in the pseudocode, and does partial sorting on the passed in array where all values in the subarrays are of the form $A[p..q-1] \leq A[q] < A[p+1..r]$. Then we recurse on the left subarray $A[p..q-1]$. However, to make up for the removal of the second recursive call and thus sorting on the right subarray $A[p+1..r]$, we simply update p to $q+1$. Thus, on the next iteration of the algorithm, we partially sort the right subarray via (partition) and recurse on the left side of the resulting subarray.

Alternatively, the algorithm can be thought of as paritioning and partially sorting the whole array but recursing on only the left subarray while the loop simply causes the next iteration to consider the right subarray. Since we end up recursing on the left subarray at each recursion and iteration and the loop update causes considering the right subarray on the next iteration, eventually all possible subarrays will be a left subarray to be sorted.

Answer 1.b

Such a scenario is similar to the requirements of worst case running time for `quicksort` wherein the pivot point chosen at each level of recursion causes a subarray of length $n-1$. If `partition` chooses a value at each level of recursion that splits the array into $A[p..r-1]$ and $A[r]$, the recursion then occurs on $n-1$ values meaning n stack frames must be pushed onto the stack before reaching the base case. The upside to this, however, is the loop will be ran only once at each level.

Answer 1.c

```
1  while p < r
2      // Partition and sort left subarray.
3      q = PARTITION(A, p, r)
4
5      // Recurse only on the smaller array
6      if ((q - p) < (r - q))
7          TAIL-RECURSIVE-QUICKSORT(A, p, q - 1)
8          p = q + 1
9      else
10         TAIL-RECURSIVE-QUICKSORT(A, q, r)
11         r = q - 1
```

The above maintains the running time of $\Theta(n \log_2 n)$ while ensuring the stack depth does not grow above $\Theta(\log_2 n)$. Because we recurse on the smallest subarray created from partitioning, at no point in time will we recurse on a subarray larger than $\frac{n}{2}$. Recursing on a subarray larger than that implicates we recurse on the larger of the two subarrays.

Because the subproblem size recursed on is bounded between 1 and $\frac{n}{2}$, the recursion tree therefore still has only one child per subproblem with a max subproblem size of $\frac{n}{2}$. As such, the depth of the recursion tree will not exceed $O(\log_2 n)$ in depth. Since any node on the recursion tree effectively represents a stack push, the max possible depth of the stack in question will also not exceed $O(\log_2 n)$.

Question 2 (9 pts total)

In the US, coins are minted with denominations of 50, 25, 10, 5, and 1 cent. An algorithm for making change using the smallest possible number of coins repeatedly returns the biggest coin smaller than the amount to be changed until it is zero. For example, 17 cents will result in the series 10 cents, 5 cents, 1 cent, and 1 cent.

a. (4 pts) Give a recursive algorithm that generates a similar series of coins for changing n cents.

b. (4 pts) Give an $O(1)$ (non-recursive!) algorithm to compute the number of returned coins.

c. (1 pt) Show that the above algorithm does not always give the minimum number of coins in a country whose denominations are 1, 6, and 10 cents.

Purpose Apply recursion to solve a problem, practice formulating and analyzing algorithms.

Answer 2.a

```
def make_change(n, A)
    """n is assumed to be in cents and therefore an int
    A is an array representing the count of the coins as such [50, 25, 10, 5, 1]
    Assuming python3 so // means integer division
    """
    if n >= 50:
        return make_change(n % 50, A.append(n // 50))
    elif n >= 25:
        return make_change(n % 25, A.append(n // 25))
    elif n >= 10:
        return make_change(n % 10, A.append(n // 10))
    elif n >= 5:
        return make_change(n % 5, A.append(n // 5))
    else  # 0 <= n < 5
        return A.append(n)
```

Answer 2.b

```
def make_change(n)
    """n is assumed to be in cents and therefore an int
    A is an array representing the count of the coins as such [50, 25, 10, 5, 1]
    Assuming python3 so // means integer division
    """
    A = []
    A.append(n // 50)
    n %= 50
    A.append(n // 25)
    n %= 25
```

```
    A.append(n // 10)
    n %= 10
    A.append(n // 5)
    n %= 5
    A.append(n)
    return A
```

Answer 2.c

To allow a more generic solution, a modification to the proposed algorithm in **Answer 2.b** is provided below:

```
def make_change(n, denominations)
    """n is assumed to be in cents and therefore an int
    denominations is assumed to be a list of ints containing the denominations
    sorted from largest value to smallest
    Assuming python3 so // means integer division
    """
    A = []  # represents the counts of the denominations
    for denomination in denominations:
        A.append(n // denomination)
        n %= denomination

    return A
```

Because the amount of operations is directly dependent on the amount of unique denominations, the running time of this algorithm can be loosely represented as $o(m)$. However, realistically, the amount of denominations should be a small number or at least much smaller than the amount of money being converted. As such, length($denominations$) $\lll n$. Because of this, the running time of the above method is $\Theta(1)$.

The algorithm works to count the amount of coins needed to represent the input value in the respective denominations. This is carried out by effectively removing the maximum possible multiple of the i^{th} denomination from the remaining amount of change.

Conceptually, the algorithm can be represented via a recursion tree where each level has only one child. Because the maximum multiple of the i^{th} denomination is removed at level i, the subproblem size at level $i+1$ is then guaranteed to be smaller than the i^{th} denomination. Because of this, the subproblem must be represented in terms of the $i+1$ denomination. For example, in American denominations, 4 pennies can not be represented with a nickel, the next largest denomination from the penny.

Proof by contradiction Assume an input with a denomination set that would produce at some point a subproblem, S, which would not produce the minimum amount of coins. This means a subproblem would exist which does produce the minimum amount of coins, S' that has fewer coins in the solution than S. In order to achieve such a subproblem, the size of S' must be larger than the denomination used on its parent subproblem. This contradicts the algorithm since the algorithm, at each denomination, will take the maximum multiple of that denomination from the running total guaranteeing the updated running total will be smaller than the denomination in question.

Note Because of the discussions on the piazza board between myself and the instructors, I googled around to see what could be meant by the homework question "Show that the above algorithm does not always give the minimum number of coins in a country whose denominations are 1, 6, and 10 cents." In looking into it, an open problem exists on the finding the optimal denominations on finding the minimum amount of coins for change called the "Change-Making Problem". Given the wording on the question, I do not believe we are being asked about this however, but rather proving only that, given a denomination, you can not generate the minimum amount of coins possible for that denomination with certain input.

Question 3 (10 pts total)

For each of the following recurrences, use the Master Theorem to derive asymptotic bounds for $T(n)$, or indicate why the Master Theorem does not apply. If not explicitly stated, please assume that small instances need constant time c. Justify your answers, in particular, for case 3 of the Master Theorem show that the regularity condition is satisfied. (2 points each)

a. (2 pts) $T(n) = 7T(\frac{n}{3}) + n^3\sqrt{n}$

b. (2 pts) $T(n) = 8T(\frac{3n}{2}) + n^3$

c. (2 pts) $T(n) = 8T(\frac{n}{5}) + n^3$

d. (2 pts) $T(n) = 4T(\frac{n}{2}) + 100 - \sqrt{n}$

e. (2 pts) $T(n) = 3T(\frac{n}{3}) + \frac{1}{\sqrt{n}}$

Purpose Practice solving recurrences.

Answer 3.a

$$T(n) = 7T\left(\frac{n}{3}\right) + n^3\sqrt{n} \implies a = 7, b = 3, f(n) = n^{3+\frac{1}{2}} = n^{\frac{7}{2}} \tag{1}$$

$$\log_3 7 < \frac{7}{2}, \implies \text{Case } 3, \epsilon \approx 1.73 \tag{2}$$

In proving the regularity condition

$$\text{Condition: } af\left(\frac{n}{b}\right) \leq cf(n) \tag{3}$$

$$7\left(\frac{n}{3}\right)^{\frac{7}{2}} \leq cn^{\frac{7}{2}} \tag{4}$$

$$7\left(\frac{1}{3^{\frac{7}{2}}}\right)\cancel{n^{\frac{7}{2}}} \leq c\cancel{n^{\frac{7}{2}}} \tag{5}$$

$$\frac{7}{3^{\frac{7}{2}}} \leq c \tag{6}$$

Answer 3.b

$$T(n) = 8T\left(\frac{3n}{2}\right) = n^3 \implies a = 8, b = \frac{3}{2}, f(n) = n^3 \tag{7}$$

$$\log_{\frac{3}{2}} 8 > 3 \implies \text{Case } 1, \epsilon \approx 2.13 \tag{8}$$

$$T(n) = \Theta\left(n^{\log_{\frac{3}{2}} 8}\right) \tag{9}$$

Answer 3.c

$$T(n) = 8T\left(\frac{n}{5}\right) + n^3 \implies a = 8, b = 5, f(n) = n^3 \tag{10}$$

$$\log_5 8 < 3, \implies \text{Case } 3, \epsilon \approx 1.71 \tag{11}$$

In proving the regularity condition

$$\text{Condition: } af\left(\frac{n}{b}\right) \leq cf(n) \tag{12}$$

$$8\left(\frac{n}{5}\right)^3 \leq cn^3 \tag{13}$$

$$8\cancel{n^3}\left(\frac{1}{5^3}\right) \leq c\cancel{n^3} \tag{14}$$

$$\frac{8}{625} \leq c \tag{15}$$

Answer 3.d

$$T(n) = 4T\left(\frac{n}{2}\right) + 100 - \sqrt{n} \implies a = 4, b = 2, f(n) = -\sqrt{n} \tag{16}$$

$$\log_2 4 > \frac{1}{2}, \implies \text{Case } 1, \epsilon = 1.5 \tag{17}$$

$$\Theta\left(n^{\log_2 4}\right) = \Theta(n^2) \tag{18}$$

Answer 3.e

$$T(n) = 3T\left(\frac{n}{3}\right)\frac{1}{\sqrt{n}} \implies a = 3, b = 3, f(n) = \sqrt{n} \tag{19}$$

$$\log_3 3 > \frac{1}{2}, \implies \text{Case } 1, \epsilon = 0.5 \tag{20}$$

$$\Theta\left(n^{\log_3 3}\right) = \Theta(n) \tag{21}$$

$$\tag{22}$$

Question 4 (5 pts)

Describe a non-recursive $\Theta(\log(n))$ algorithm which computes a^n, given a and n. Don't forget to justify the asymptotic running time of your algorithm. You may assume that n is a positive integer, but do not assume that n is always a power of 2.

Purpose More practice in algorithm design and algorithm analysis.

Solution

CALC-POWERS(a, n)

```
// Setup. Powers will hold the intermittent powers
powers = []
i = 0

// Build out powers.
while n > 1
    powers[i++] = ceil(n/2) - floor(n/2)  // 0 means even split, 1 is odd split
    n = floor(n/2)

// Start building product
product = a          // Base case, n = 1
i--                  // Go back to last power (powers[i] should be a^1)
while i >= 0         // Until we have calculated all powers...
    product *= product      // Multiply left and right sibling together
    if powers[i] == 1       // If the split was uneven...
        product *= a        // include an extra 'a'
    i--              // Go to parent power
return product
```

Justification

The proposed solution takes advantage of the following relationships; $a^x \times a^y = a^{xy}$; $0 < x < \infty, a^x \neq 0 \quad \forall x \in \mathbb{Z}^+$; and uses Dynamic Programming to store intermediate results.

Conceptually, a^n is broken into $a^{\lfloor \frac{n}{2} \rfloor}$ and $a^{\lceil \frac{n}{2} \rceil}$. From here, an implicit binary tree represents the subproblems of calculating the power of a^n where $a^{\lfloor \frac{n}{2} \rfloor}$ is the left child and $a^{\lceil \frac{n}{2} \rceil}$ is the right child. Because the left child is always $a^{\lfloor \frac{n}{2} \rfloor}$, the left child is always less than or equal to the right child with at most a difference of one: $\lceil \frac{n}{2} \rceil - \lfloor \frac{n}{2} \rfloor \in \{0, 1\}$. When the split is uneven, the split is marked as such, otherwise no mark is given. From here, the splits continue down the left child until $\lfloor \frac{n}{2} \rfloor = 0$. Such a condition implicates $n = 1$ which is a useful base case.

From here, a running product is kept by simply multiplying the product with itself causing an ascent in the implied tree discussed above towards the root, a^n wherein a is the starting value of the product. When an uneven split is detected, the running product is simply multiplied by a.

When fully displayed, the conceptual tree shows not only a large number of redundant

subproblems, but also a large number of subproblems which are extremely close in value to each other and require an additional, constant-time operation to calculate them. For example, any non-leave node will have two children where the work in each is either same or the same plus an additional multiplication. As such, all right children are effectively precalculated when under consideration and require no recursion. Recursion occurs on only left children, so every node has only one subproblem. Furthermore, because subproblems are always divided by two in size, the height of any tree is $\log_2 n$. And finally, since only one subproblem exists for each level, the running time for this algorithm is $\Theta(\log(n))$

Likewise, the data structure marking uneven splits is extremely compact since we need only ones and zeros.

Question 5 (5 pts)

This problem was an interview question! Consider a situation where your data is almost sorted—for example you are receiving time-stamped stock quotes and earlier quotes may arrive after later quotes because of differences in server loads and network traffic routes. Focus only on the time-stamps. To simplify this problem assume that each time-stamp is an integer, all time-stamps are different, and for any two time-stamps, the earlier time-stamp corresponds to a smaller integer than the later time-stamp. The time-stamps arrive in a stream that is too large to be kept in memory completely. The time-stamps in the stream are not in their correct order, but you know that every time-stamp (integer) in the stream is at most hundred positions away from its correctly sorted position. Design an algorithm that outputs the time-stamps in the correct order and uses only a constant amount of storage, i.e., the memory used should be independent of the number of time-stamps processed. Tip: map the problem to a data structure covered in class.

Purpose Practice algorithm design and the use of data structures.

The stream is assumed to provide integers one at a time and not in any batch form and the time between providing integers is an non-deterministic.

Min Heap Method

If space must be minimized as much as possible, a Min Heap would allow sorting of data as it is provided via the stream while minimizing space as much as possible. Since the integers are provided one at a time, a Min Heap will be implicitly built from incoming integers via a `min_heapify` function similar to the `max_heapify` function defined in the text (p. 154). To minimize space requirements, a size limit is imposed on the heap of 100 elements. This size can be some attribute of the heap itself to make checking the limit a constant time operation.

Upon initialization, no integers are yet received, so the heap is empty and the size is 0. Upon the first integer presented from the stream, `min_heapify` inserts the integer into the root of the tree. When the second, and eventually i^{th} integer is presented, `min_heapify` works to insert the integer into the appropriate location in the tree. Upon hitting the size limit of 100 integers, the root of the tree will be removed into the output. The new integer will be set as the root of the tree and `min_heapify` is called to percolate down the new integer. Since the timing of the next integer input from the stream is unknown and the next integer is not guaranteed to be in a specific spot of the tree upon arrival, the tree can not be emptied out until program closure where some tear- down function will effectively run a `heapsort` method against the tree using `min_heapify`.

Because any heap will be a complete tree, the depth of the tree at any time will be $\lfloor \log_2(n) \rfloor + 1$. Furthermore since the amount of elements in the tree is bounded above by 100, the height of the tree is then bounded by $\lfloor \log_2(100)+1 \rfloor = 7$ meaning the maximal amount of operations for any `min_heapify` call will be 7. This ends up modifying the worst case scenario of heapsorting the entirety of input from the stream from $O(n \log_2(n))$ to $O(7n) = O(n)$

Due to the Min Heap property of the root of any sub-tree being the smallest value in the tree, selection of the next value to provide to the output is a simple selection of the root of the tree and therefore a constant time operation, $\Theta(1)$.

The limitation of a size of 100 for the tree further satisfies the requirement of constant storage size while allowing the data to be sorted. To explain, consider the three possible states the tree can be in, empty, partially filled, and full where full means 100 values reside in the tree while partially filled means the amount of values in the tree is $0 < n < 100$. When the tree is empty, inserting the next integer automatically is the root of the tree and the minimum value in the tree. When partially filled, the next integer will end up percolating to the next available spot satisfying the requirements of the `min_heapify` function. When the tree is full the first time, the 101-st element has come in and a value needs to be taken from the tree. Since the root of the tree has the smallest value in the tree out of all 100 values, the root of the tree must be the smallest element from the stream. To prove this, assume the 101-st element is actually the smallest element from the stream. This means it is smaller than all 100 elements before it contradicting the partially sorted rule where any element is within 100 spaces of its correct position; in this scenario, the 101-st element is over 100 elements away from its sorted position. This is maintained throughout execution; at input $i > 100$ from the stream, the sorted output will contain the $i - 100$ smallest elements that the stream will ever produce in sorted order, the tree will be filled with elements between i and $i - 100$ with the root being the next value to go to the sorted output.