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

Worksheet 3 — Recurrences and Divide and Conquer I (with solutions)

1. Give pseudocode for a divide and conquer algorithm to compute a minimum (or maximum) of an array $A[1..n]$ in $O(n)$ time.

FIND-MIN($A[1..n]$)

```
1: if A.length = 1 then
2:     return A[1]
3: min_left = FIND-MIN(A[1..n/2])
4: min_right = FIND-MIN(A[n/2 + 1..n])
5: return min{min_left, min_right}
```

The asymptotic running time for FIND-MIN is represented by the recurrence

$$T(n) = 2T(n/2) + \Theta(1),$$

which solves to $T(n) = \Theta(n)$ by case 2 of the master theorem.

2. Solve the recurrence $T(n) = T(n/2) + \lg n$ using the direct method.

   Assume that $T(1) = 1$ and n is a power of 2.

   Proof.

$$\begin{aligned}
T(n) &= T(n/2) + \lg n \\
&= T(n/2^2) + \lg\left(\frac{n}{2}\right) + \lg n \\
&= T(n/2^3) + \lg\left(\frac{n}{2^2}\right) + \lg\left(\frac{n}{2}\right) + \lg n \\
&= T(n/2^i) + \sum_{j=0}^{i-1} \lg\left(\frac{n}{2^j}\right) \\
&= T(1) + \sum_{j=0}^{\lg n - 1} \lg\left(\frac{n}{2^j}\right) \\
&= 1 + \sum_{j=0}^{\lg n - 1} \left(\lg n - \lg 2^j\right) \\
&= 1 + \sum_{j=0}^{\lg n - 1} \lg n - \sum_{j=0}^{\lg n - 1} \lg 2^j \\
&= 1 + \sum_{j=0}^{\lg n - 1} \lg n - \sum_{j=0}^{\lg n - 1} j \\
&= 1 + \lg^2 n - \frac{(\lg n - 1)(\lg n)}{2} \\
&= 1 + \lg^2 n - \frac{\lg^2 n}{2} + \frac{\lg n}{2} \\
&= 1 + \frac{\lg^2 n}{2} + \frac{\lg n}{2}.
\end{aligned}$$

   □

   Then $T(n) = \Theta(\lg^2 n)$.

3. Give an asymptotic upper bound for the recurrence $T(n) = T(n/2) + \lg n$ using the substitution method.

   From the previous problem, we know that $T(n) = O(\lg^2 n)$. Therefore, we show that $T(n) \leq c \lg^2 n$ for sufficiently large values of n.

   Proof. We inductively assume that $T(n/2) \leq c \lg^2(n/2)$, and show that $T(n) \leq c \lg n$.

$$\begin{aligned} T(n) &= T(n/2) + \lg n \\ &\leq c \lg^2(n/2) + \lg n && \text{I.H.} \\ &= c(\lg n - \lg 2)^2 + \lg n \\ &= c \lg^2 n - 2c \lg n + 1 + \lg n \\ &= c \lg^2 n - ((2c-1) \lg n - 1) \\ &\leq c \lg^2 n && \text{when } (2c-1) \lg n - 1 \geq 0. \end{aligned}$$

   Where $(2c-1) \lg n - 1 \geq 0$ for $c = 1$ and all $n \geq 2$. □

4. Can the recurrence $T(n) = T(\sqrt{n}) + \lg n$ be solved using the master method? Explain.

   No. To apply the master method, the recurrence must have the form $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$ are constants. In this case, there is no constant b that allows this recurrence to be solved directly with the master method. Instead, it has the form

$$T(n) = T(n/\sqrt{n}) + f(n),$$

   and therefore b is not constant. However, it is possible to first change variables and apply the master method.

5. Use the master method to show that $T(n) = 2T(n/2) + \Theta(n)$ solves to $T(n) = \Theta(n \lg n)$.

   Proof. Our recurrence is of the form $T(n) = aT(n/b) + f(n)$ and therefore the master theorem applies. Notice that $n^{\log_b a} = n = \Theta(n) = \Theta(f(n))$. Therefore, by case 2 of the master theorem, we have that $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n \lg n)$. □

6. Use the master method to solve the following recurrences. Explain the case you apply from the master theorem, and justify your choice.

   (a) $T(n) = 9T(n/3) + n$

   Case 1: $f(n) = n = O(n^{\log_3 9 - 1}) = O(n^{2-\epsilon})$ for $\epsilon = 1 > 0$.
   Therefore: $T(n) = \Theta(n^{\log_3 9}) = \Theta(n^2)$

   (b) $T(n) = 9T(n/3) + n^2$

   Case 2: $f(n) = n^2 = \Theta(n^{\log_3 9}) = \Theta(n^2)$.
   Therefore: $T(n) = \Theta(n^{\log_3 9} \lg n) = \Theta(n^2 \lg n)$.

   (c) $T(n) = 9T(n/3) + n^3$

   Case 3: $f(n) = n^3 = \Omega(n^{\log_3 9 + 1}) = \Omega(n^{2+\epsilon})$ for $\epsilon = 1 > 0$. Let's also check the regularity condition:

$$\begin{aligned} af(n/b) &= 9(n/3)^3 \\ &= 1/3n^3 \\ &\leq 1/3f(n) \end{aligned}$$

   Thus, the regularity condition holds for $c = 1/3 < 1$.
   Therefore: $T(n) = \Theta(f(n)) = \Theta(n^3)$.

7. Can the master method be applied to the recurrence $T(n) = 27T(n/3) + n^3 \lg n$? Why or why not?

   No, the master method does not apply. Case 3 is the closest, but $n^3 \lg n \neq \Omega(n^{3+\epsilon})$, since $\lg n \leq n^\epsilon$ for any constant $\epsilon > 0$. That is, $\lg n$ grows slower than any polynomial factor n^ϵ for $\epsilon > 0$. We can see this by comparing the first derivative of each.

$$\frac{d}{dn} \lg n = \frac{d}{dn} \frac{\ln n}{\ln 2} = \frac{1}{n \ln 2}$$
$$\frac{d}{dn} n^\epsilon = \epsilon n^{\epsilon - 1} = \epsilon \frac{n^\epsilon}{n}$$

   When is $\frac{1}{n \ln 2}$ strictly smaller than $\epsilon \frac{n^\epsilon}{n}$?

$$\begin{aligned} \frac{1}{n \ln 2} &< \epsilon \frac{n^\epsilon}{n} \\ \frac{1}{\ln 2} &< \epsilon n^\epsilon && \text{(multiply by } n > 0) \\ \frac{1}{\epsilon \ln 2} &< n^\epsilon && \text{(divide by } \epsilon > 0) \\ \left(\frac{1}{\epsilon \ln 2}\right)^{1/\epsilon} &< n && \text{(to the power of } 1/\epsilon). \end{aligned}$$

   Therefore $\lg n$ grows slower than n^ϵ for all $n > \left(\frac{1}{\epsilon \ln 2}\right)^{1/\epsilon}$, which is a constant.

8. Suppose you are given an array $A[1..n]$ of n distinct numbers, and you are told that A is *unimodal*: There exists some index p such that $A[p]$ is the maximum element in A, and values from index 1 to $p-1$ in increasing order, and from $p+1$ to n are in decreasing order. That is, $A[1] < A[2] < \cdots < A[p]$, and $A[p+1] > A[p+2] > \cdots > A[n]$.

   (a) Describe a brute force method to compute the index p. What running time does it have? Do you think the problem can be solved faster? How fast do you think the problem can be solved?

FIND-P($A[1..n]$)

```
1: for i = 2 to n do
2:     if A[i] < A[i − 1] then
3:         return i − 1
4: return n
```

I think the algorithm can be solved in $O(\lg n)$ time.

   (b) With divide and conquer, the problem can be solved in $O(\log n)$ worst-case time. What recurrence would you guess represents this running time?

   This problem can be solved in $\Theta(\lg n)$ worst-case time. I would propose the recurrence

$$T(n) = T(n/2) + \Theta(1).$$

   (c) Devise a divide and conquer algorithm with this running time and briefly describe it.

   I propose an algorithm based on binary search. We check each midpoint value $A[\text{mid}]$ and check its neighboring elements $A[\text{mid} - 1]$ and $A[\text{mid} + 1]$. If $A[\text{mid}]$ is the peak, then we are done. Otherwise, we continue searching on either the elements before mid or after mid.

(d) Give pseudocode for your algorithm.

Assumption: The peak is neither $A[1]$ nor $A[n]$.

```
RECURSIVE-FIND-P(A, a, b)
1: mid = (a + b)/2
2: if A[mid − 1] < A[mid] and A[mid + 1] < A[mid] then
3:     return mid
4: else if A[mid − 1] > A[mid] > A[mid + 1] then
5:     return RECURSIVE-FIND-P(A, a, mid)
6: else
7:     return RECURSIVE-FIND-P(A, mid, b)
```

(e) Describe an invariant that holds throughout your algorithm, and state why your invariant proves correctness.

The peak index p is always between a and b, exclusive. Throughout search, since p is never outside of this range, we do not erroneously skip over p, it is always in our search range. Since this is the case, and we repeatedly decrease our range a, b we will eventually find p.

9. Suppose you work for an investment firm, and it is your job to analyze stock histories. You are given a particular stock, and an array $A[1..n]$ of its daily price over n days—one price per day. The firm would now like you to analyze when it *should have* bought and sold stock, by choosing one day to buy stock and one day to sell the stock to maximize profit.

   (a) Come up with a formal definition of the problem: What is the input, what is the output?
   Input: An array $A[1..n]$ floating point numbers representing stock prices.
   Output: Indices $i < j$, such that $A[j] - A[i]$ is maximum, over all pairs of indices.

   (b) Now come up with a small example for the problem that you can solve easily. For example, for $n = 6$.
   $$[1, 2, 3, 4, 5, 6]$$
   Buy at index 1, sell at index 6, for a profit of $6 - 1 = 5$.

   (c) Can you come up with a correct, but naïve algorithm, that solves the problem in $O(n^2)$ time?
   Compare all pairs of indices $i < j$, maintain a maximum profit and 2 indices throughout all comparisons, and return the pair that maximizes profit at the end. Pseudocode:

```
BUY-SELL(A[1..n])
 1: max = 0
 2: buy = 0
 3: sell = 0
 4: for i = 1 to n − 1 do
 5:     for j = i to n do
 6:         if A[j] − A[i] > max then
 7:             max = A[j] − A[i]
 8:             buy = i
 9:             sell = j
10: return (buy, sell)
```

(d) What is an invariant that implies correctness of your algorithm?

When evaluating position i, we have computed the maximum profit when making a buy on any day in $1..i-1$.

(e) Now let's try to solve the problem with divide and conquer. Briefly describe how you will do each of the following steps. The combine step is critical to the efficiency of this problem.

i. **Divide:** Divide $A[1..n]$ into two subarrays $A[1..n/2]$ and $A[n/2+1..n]$.

ii. **Conquer:** Recursively solve the subarrays $A[1..n/2]$ and $A[n/2+1..n]$.

iii. **Combine:** We select the buy and sell days with the most profit by comparing the maximum profit from 3 possibilities: the maximum profit from subarray $A[1..n/2]$, the maximum profit from $A[n/2+1..n]$, and when we buy in $A[1..n/2]$ and sell in $A[n/2+1..n]$. The latter is determined by selecting the day with minimum stock price in $A[1..n/2]$, and the day with maximum stock price in $A[n/2+1..n]$.

(f) What recurrence do you need to reach running time $O(n \lg n)$? How can you implement your steps so that you achieve this running time?

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

Divide into 2 equally-sized subproblems, solve each, and combine solutions by computing the minimum price in $A[1..n/2]$ and maximum price in $A[n/2+1..n]$ in time $\Theta(n)$.

(g) Put together the final algorithm with pseudocode. Does your invariant from your naïve method still hold? If you have a new invariant, does it also apply to your naïve algorithm?

RECURSIVE-BUY-SELL($A[1..n]$, a, b)

1: **if** $a = b$ **then**
2: buy $= a$
3: sell $= b$
4: **return** (buy, sell)
5: mid $= (a + b)/2$
6: $(\text{buy}_1, \text{sell}_1)$ = RECURSIVE-BUY-SELL(A, a, mid)
7: $(\text{buy}_2, \text{sell}_2)$ = RECURSIVE-BUY-SELL(A, mid $+ 1$, b)
8: $\text{buy}_3 = \text{argmin}_{k \in [a..\text{mid}]}\{A[k]\}$ ▷ Choose index k such that $A[k]$ is minimized
9: $\text{sell}_3 = \text{argmax}_{k \in [\text{mid}+1..n]}\{A[k]\}$ ▷ Choose index k such that $A[k]$ is maximized
10: **return** (buy, sell)

The invariant for this problem changes slightly, but remains the same in spirit. When evaluating subarray $A[a..b]$, we have computed the maximum profit when making a buy and sell on any day in $[a..b]$.

(h) It turns out the problem can be solved in time $O(n)$, and it is possible to achieve this running time with divide and conquer. How can you adjust the recurrence from (f) to achieve this running time? Describe a divide and conquer algorithm that has this running time.

My new recurrence is:

$$T(n) = 2T(n/2) + \Theta(1).$$

Which we can achieve by keeping track of the min and max of both the left and right subproblems and returning these as a part of the recursion. Then it takes time $O(1)$ to compute the best solution spanning between the left and right halves.

10. Suppose that you are given an array $A[1..n]$ consisting of all integers 0 to n—except one. Also assume that n is a power of two (e.g., $\exists k \in \mathbb{N}$ such that $n = 2^k$). It will be your job to find the missing integer. A standard solution is to make a new auxiliary array $B[0..n]$, where you put each integer from A indexed by itself (i.e., $B[0] = 0, \ldots, B[i] = i$). The missing integer from A is the index j in B where $B[j] \neq j$, which can be found by iterating over all elements of B. This algorithm takes $O(n)$ total time. Now let's suppose we have a restriction on how integers are compared. Traditionally, we can access full integers and compare them in constant time. Instead, suppose we can only access integers one bit at a time, using the following operation, which takes constant time: given integers i and j, you can retrieve the j-th bit of the integer $A[i]$ in constant time. Show that we can still compute the missing number in $O(n)$ operations.

    Note that you can't compute all the integer values in $O(n)$ time. In the RAM model, we assume that each value is represented by $\Theta(\lg n)$ bits. Computing all the integers from all bits would take $\Theta(n \lg n)$ time.

    Hints at getting started:

    (a) Use divide and conquer, define your steps by following a similar strategy for the previous problem.

    (b) Figure out what recurrence(s) will help you reach $O(n)$ time.

    (c) What information can you use can you divide up the problem? How small should the subproblems be?

    (d) What do the elements in your subproblem have in common?

    Beginning with the 0-th bit, we divide the integers into two sets; those with a 0 as the 0-th bit (let's call this set S_0), and those with a 1 for the 0-th bit (let's call this set S_1). Since the number of numbers between 0 and n (inclusive) is $(n+1) = 2^k$, there are $(n+1)/2$ values with a 0 and $(n+1)/2$ values with a 1 for the 0-th bit in the range $[0..n]$. Since there is one number missing, one of S_0 and S_1 will have cardinality $(n+1)/2 - 1 = 2^{k-1} - 1$, which tells us the 0-th bit of the missing integer. We then recursively perform this algorithm on the 1-th bit on the set (S_0 or S_1), which has cardinality $2^{k-1} - 1$. We continue until we recover all $\lg n + 1$ bits, and we have our missing integer. It takes linear time to check the j-th bit of all numbers, and we make a total of $\lg n$ recursive calls, each of which splits the problem in half. Note further that n is odd, and therefore $(n+1)/2 - 1$ is $n/2 - 1/2 = \lfloor n/2 \rfloor$. Therefore, our recurrence is

$$T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n),$$

    which solves to $T(n) = O(n)$.

(Use this page to show your work for problem 10)