CSC373— Algorithm Design, Analysis, and Complexity — Spring 2018

Suggested Solutions for Tutorial Exercise 3: Divide and Conquer

1. Divide and . . .: Max Step. Given an array of $n \geq 2$ integers, say $[x(1), \ldots, x(n)]$, we want to find the largest step d, which is defined to be the max of $x(j)-x(i)$ over all $j > i$. For example, for $x = [22, 5, 8, 10, -3, 1]$

$$d \;=\; x(4) \;-\; x(2) \;=\; 10 \;-\; 5 \;=\; 5.$$

1a) Divide: Finish the pseudo-code LS below for computing d. Your algorithm must make essential use of a divide and conquer strategy which splits the current problem roughly in half. We realize that there are simpler ways to compute d; the point here is for you to practice divide and, well, see what you get (if not actually rule the world).

Solution 1a) (comments not required, but some can be helpful to marker):
[d, mn, mx] = LS(x, a, b) // Insert two return values in function definition above.

// For the function defined further below, the largest step d is given by,
[d] = LS(x, 1, n)
// Assume a function call can return more values than are actually assigned to variables on the left.

```
[ d, mn, mx ] = LS(x, a, b)
    // Input: Array x(1..n) of integers, and indicies $1 \leq a < b \leq n$.
    //        We require $n \geq 2$.
    // Output: d is the argest step d = x(j) - x(i), for any i, j with $a \leq i < j \leq b$, and
    // mn, mx the min and max (respectively) values in x(a..b)

    if b - a == 1           // Base Case: Length of sublist is only 2.
        return [x(b) - x(a), min(x(a), x(b)), max(x(a), x(b))]

    k = a+ceil((b-a)/2)                    // Since $b - a \geq 2$ it follows that $a < k < b$.

    [d1, mn1, mx1] = LS(x, a, k)           // Find largest step in first half and range
    if $b > k + 1$                         // List in second half has 2 or more elements.
        [d2, mn2, mx2] = LS(x, k+1, b)     // Find largest step in second half and range
    else                                   // Second half is length one. No step is possible.
        d2 = d1; mn2 = x(b); mx2 = x(b);   // Harmless to set d2 = d1 in this case.

    // The desired result $d$ is maximum of three quantities:
    //     the max steps in each of the left and right halves,
    //     and the max step from the left to the right half, namely mx2 - mn1.
    // We also return the minimum and maximum of both halves.
    return [max(d1,d2, mx2-mn1), min(mn1, mn2), max(mx1, mx2) ]
```

1b) Runtime. Given an array x of length n, let $T(n)$ be the runtime of $LS(x, 1, n)$ in part (a). Write an equation expressing $T(n)$ in terms of $T(k)$ for some integer(s) $k < n$. Use the Master Theorem to determine the order $T(n)$. (Hint: You should be able to achieve $O(n)$, which is the same order as a simpler approach.)

Solution: The recurrence relation for the runtime is $T(n) \;=\; T(\text{ceil}(n/2)) \;+\; T(\text{floor(n/2)}) \;+\; O(1)$

Explanation: LS is recursively called twice on lists of length roughly n/2. Not counting these recursive calls, the rest of LS runs in constant time, hence the O(1) above. The Master Thm applies with a = b = 2 and d

$= 0$. So $a > b^d$ and the first case applies, so the runtime is $T(n) = O(n)$. (Which is the same as can be done with a simple forward loop through the data, but we exercised out divide-and-conquer muscles.)

2. Divide and Conquer: Power Mod. For positive integers y and b, we define the operation y mod b as: $z = y$ mod b if and only if $z = y - jb$ where j is the maximum integer such that $jb \le y$. So, in particular, $0 \le y$ mod $b < b$. We are interested in computing the **power-mod**, which is defined as

$$\text{powerMod}(y, n, b) \equiv y^n \mod b, \tag{1}$$

where we are given integers $y > 0$, $n > 0$, and $b > 0$.

Suppose we can use a somewhat limited built-in mod function that runs in constant time, but it does not apply to really large input integers y. In particular, assume that for some positive constant Y, y mod b can only be computed with the built-in function when $|y| \le Y$. In this problem we will also assume that $b^2 < Y$.

The difficulty in implementing powerMod is that it might be the case that y^n is much larger than Y and we therefore cannot directly use the built-in mod function. Instead we will need to make use of the following property of the (mathematical) mod operation:

Property 1. For any two integers $u, v > 0$, we have $(uv \bmod b) = ([(u \bmod b)(v \bmod b)] \bmod b)$.

2a. For more practice with proofs, prove Property 1. (Note that here we are using the mathematical mod function, not the restricted built-in function.)

Solution 2a) Let j be the max integer such that $j \le u/b$, and k be the max integer such that $k \le v/b$. Then, by the definition of mod, $u = jb + (u \bmod b)$ and similarly $v = kb + (v \bmod b)$. Therefore

$$\begin{aligned}(uv \mod b) &= ((jb + (u \mod b))(kb + (v \mod b))) \mod b \\ &= ([mb + (u \mod b)(v \mod b)] \mod b), \text{ for some integer } m, \\ &= [(u \mod b)(v \mod b)] \mod b\end{aligned}$$

Here, on the right hand side of the first equation above, three of the terms in the product are integer multiples of b. These terms are collected into the term mb in the second line. The last equation above uses the fact that, for any integer k, $((x + kb) \bmod b) = (x \bmod b)$. ■

2b. Divide: We wish to compute $a = y^n$ mod b for any integer $n > 0$, and y and b are as above. Use a divide and conquer approach to compute a. Your algorithm must run in $\Theta(\log(n))$ time. Take careful note of the restriction on the built-in mod function described above. Clearly explain your algorithm and show how you derived its runtime estimate. (Hint: On you first try you may obtain a $\Theta(n)$ algorithm. Use the Master Theorem to identify what you need to speed up to get $\Theta(\log(n))$.)

Sample Solution 2b) The key idea is to recursively call the function powerMod(y, m, d) **only once** per recursive step. (If you recursively call powerMod twice the algorithm will be $\Omega(n)$.)

We can do this by writing $y^n = y^m y^m y^\delta$, where $m = \text{floor}(n/2)$ and $\delta = 0$ or 1. We can then apply Property 1, making sure to do only one recursive call for (y^m mod b). See the algorithm below.

```
$r$ = powerMod( $y$, $n$, $b$)
    % Precondition: $y, n, b$ postive integers, with $y$ and $b^2 \le Y$,
    % where $Y$ is the max allowable argument of $Y$ in the
    % built in mod function, $Y$ mod $b$.

    if n == 1
        return (y mod b)

    m = floor(n/2)                    % So $n - 1 \le m + m \le n$
```

```
r = powerMod(y, m, b)
r = ((r * r) mod b)                 % Now $r = ((y^{2m}) \bmod b)$ by Property 1.
if $2 * m < n$                      % In this case $y^n = y^{2m}y$.
    r = (r * (y mod b)) mod b       % So $r = (y^{2m})(y) \bmod b$ by Property 1.

return r
```

The recurrence relation for the runtime of this algorithm is $T(n) = T(floor(n/2)) + \Theta(1)$. Therefore, the Master Theorem applies with $a = 1$, $b = 2$ and $d = 0$. In this case we have $a = b^d = 1$, so the theorem ensures $T(n) = \Theta(\log n)$.

3. Dynamic Programming for Schedule Blocking. The following problem was on Assignment 1, but this time we approach it with dynamic programming.

The input information for this problem is the same as the interval scheduling problem considered in class. That is, suppose you are given a set of jobs, $\{J(k)\}_{k=1}^{K}$ where each job $J(k)$ must run in the (real-valued) time interval $(s_k, f_k) \subset \mathbb{R}$. Here we assume that these interval endpoints satisfy $0 \le s_k < f_k$ where s_k and f_k are non-negative integers for all k.

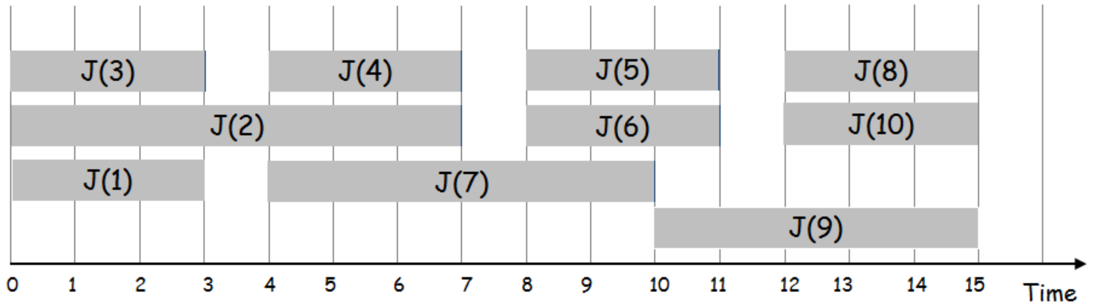
We will refer to these jobs simply by their indicies $1, 2, \ldots K$. We will abuse this notation slightly and simply refer to $J(k)$ as "job k".

A subset of jobs $C \subseteq \{1, 2, \ldots K\}$ is said to be **compatible** iff for each pair $k, j \in C$, with $k \neq j$, we have $(s_k, f_k) \cap (s_j, f_j) = \emptyset$. Otherwise C is said to be incompatible.

Here we consider the first version of the blocking set considered on Asgn#1. that is,

B must be compatible. A subset $B \subseteq \{1, 2, \ldots K\}$ is said to be a **blocking subset** iff for each $k \notin B$ and $1 \le k \le K$, it must be the case that job k is not compatible with at least one job in B. Moreover, we require that B itself is compatible.

Problem. Given input intervals of the above form $\{J(k)\}_{k=1}^{K}$, find a minimimum-sized blocking subset B.



For the example above we have $K = 10$ input intervals. Note that, since we consider the jobs to be executed in open time intervals, the two jobs $J(7)$ and $J(9)$ are compatible even though one starts and the other ends at time 10. Observe that $C = \{3, 7, 10\}$ forms a blocking subset since there is no additional job that is compatible with jobs already in C. Can we find a blocking subset that has size less than $|C| = 3$? Indeed, $B = \{2, 9\}$ is such a blocking subset of size two, and this turns out to be of the minimum possible size for this example. So a possible solution for this case would be $B = \{2, 9\}$.

3a) A new instructor, Professor Field, suggests you try dynamic programming on this problem. To set this up, she suggests that you first sort the intervals by finish time. Suppose this has been done, and we have renumbered the sorted intervals, so now $f_k \le f_j$ for all $k < j$. For each job $k \in \{1, 2, \ldots, K\}$, define the **set**

of immediate-precursor jobs as

$$P(k) = \{j \mid f_j \leq s_k \text{ and there is no intervening compatible job } i > j \text{ with } f_j \leq s_i < f_i \leq s_k\}. \tag{2}$$

It is also useful to define a termination set of jobs which do not appear in any $P(k)$, say

$$T = \{j \mid 1 \leq j \leq K \text{ and, there no compatible job } i > j \text{ with } f_j \leq s_i\}. \tag{3}$$

Prof. Field suggests that it is possible to compute all the $P(k)'s$ and T in $O(dK)$ time, where d is the "depth" of the given input set. (That is, d is the maximum number of jobs in the input that need to run at any given real-valued time, t. For the example above, $d = 3$.) Clearly describe such an algorithm.

Sketch of Solution 3a). *How about this?* Place each interval start and end time into a structure which represents the following three items: time (i.e., f_k, or s_k, whether it is a start or stop, and k, the interval number). Sort these structures by increasing time $O(K\log(K))$. Loop through this list of structures maintaining something like a rolling list (similar to a queue but with an operation that all items on or earlier than a specified item in the queue can be deleted in O(1) time). Whenever you add a finish time on the queue, say f_k, then all jobs that were added with times on or earlier than s_k can be deleted from the rolling list. Whenevery you add a start time, say s_k, you can construct $P(k)$ from this queue. (The queue should never be longer than $O(d)$.)

3b) Prof. Field then suggests that a dynamic programming solution may solve this problem if, working forward through the jobs in finish time, you simply keep track of one integer score, say $N(k) = |B(k)|$, per job. Moreover, here $B(k)$ is a minimum sized blocking set for the jobs $i \in \{1, 2, \ldots, k\}$ **where this blocking set $B(k)$ must contain job k**. That is, it must be the case that $k \in B(k)$. (Hint: The sets $P(k)$ defined in part (3a) are useful for marching forward in k and T may be useful as well.) Describe such an algorithm for computing these $N(k)$ for all $k \in \{1, 2, \ldots, K\}$. Give enough detail so that another group could implement it. What is the running time?

Sketch of Solution 3b). Loop forward through finish times f_k, computing

$$N(k) = \begin{cases} 0 & \text{if } P(k) = \emptyset, \\ \min_{j \in P(k)}[N(j) + 1] & \text{otherwise.} \end{cases} \quad . \tag{4}$$

This runs in $O(Kd)$ time, where $d = \max\{|P(k)| \mid 1 \leq k \leq K\}$.

3c) Given $N(k)$ for all k, describe an algorithm for computing a minimum-sized blocking set for the original problem. What is the running time of this last step?

Sketch of Solution 3c).

```
S = getSolution(T, P, N)
    $S \leftarrow \{\}$
    If $T$ is empty
        return S
    Find the $kinT$ for which $N(k)$ is minimum
        (i.e., $k = \text{argmin}_{j \in T} N(j)$).
    $S \leftarrow \{k\}$
    while $P(k)$ is not empty
        $j = \text{argmin}_{i \in P(k)} N(i)$
        **Assertion**: $N(k) == N(j) + 1$
        $k \leftarrow j$
        $S \leftarrow S \cup \{k\}$
    end while
    return $S$
```

This also runs in $O(Kd)$ time.

3d) During the forward pass through the computation of $N(k)$, at what values of k can you state that there must be an optimum solution for the whole problem that contains a particular job j, for some j that either equals k or conflicts with job k? That is, when can you ever decide that a specific paritial solution that you have after iteration k "is promising"? (For example, if we did not have job $J(7)$ in the above example then, after you process the fourth job, you know job $J(2)$ must be in any optimum solution.) To determine such points you may consider the conflict set for job k, and some of the jobs with even later start times (if any). (**Update:** It appears that in order to determine these points of certainty, it is useful to look at properties of rolling list in part (3a) near the time the k^{th} interval is pushed on the rolling list.)

Sketch of Solution 3d). *How about this?* Whenever the running list in part 3a is such that all jobs on the list include their finish times. That is, the set of jobs between real-valued times t at which none of the input jobs are scheduled to run (i.e., t such that, $|\{k \mid 1 \leq k \leq K \text{ and } t \in (s_k, f_k)\}| = 0$). The point here is that we can solve the sub-problems between each successive pair of such times without concern about could happen outside these times. That is, we could solve each of these subproblems in parallel.

Specifically, to answer (3b), suppose there is a k at which f_k was just pushed on the rolling list (in the manner described in 3a), and the resulting list was such that all jobs on that list were already finished. Define $T(k)$ to be the set of jobs on this list. Then job j, where $j = \text{argmin}_{i \in T(k)} N(i)$, must be in any optimum solution. In fact, we can compute an entire "promising" solution from this point back, just as in (3c).