(a) Supopose $s_1 = 10$ and $s_i = 1$ for all $i > 1$; and $x_i = 11$ for all i. Then the optimal solution should re-boot in every other day, thereby processing 10 terabytes every two days.

(b) This problem has quite a few correct dynamic programming solutions; we describe several of the more natural ones here.

1. Let $\mathsf{Opt}(i, j)$ denote the maximum amount of work that can be done starting from day i through day n, given the last reboot occurred j days prior, i.e., the system was rebooted on day $i - j$.

   On each day, there are two options:

   - *Reboot*: which means you don't process anything on day i and day $i + 1$ is the first day after the reboot. Hence, the optimal solution in this case is

     $$\mathsf{Opt}(i, j) = \mathsf{Opt}(i + 1, 1).$$

   - *Continue Processing*: which means that on day i you process the minimum of x_i and s_j. Hence, the optimal solution in this case is

     $$\mathsf{Opt}(i, j) = min\{x_i, s_j\} + \mathsf{Opt}(i + 1, j + 1).$$

   On the last day, there is no advantage gained in rebooting and hence

   $$\mathsf{Opt}(n, j) = min\{x_n, s_j\}$$

   The Algorithm:

   Set $\mathsf{Opt}(n, j) = min\{x_n, s_j\}$, for all j from 1 to n
   for $i = n - 1$ downto 1
   for $j = 1$ to i
   $\mathsf{Opt}(i, j) = max\{$ $\mathsf{Opt}(i + 1, 1)$, $min\{x_i, s_j\}$+$\mathsf{Opt}(i + 1, j + 1)\}$
   endfor_j
   endfor_i
   return Opt(1,1)

   Running Time: Note that the *max* is over only 2 values and hence is a constant time operation. Since there are only $O(n^2)$ values being calculated, and each one takes $O(1)$ time to calculate, the algorithm takes $O(n^2)$ time.

2. Let $\mathsf{Opt}(i, j)$ to be the maximum number of terabytes that can be processed from days 1 to i, given that the last reboot occurred j days prior to the current day.

   When $j > 0$ (i.e., the system is not rebooted on day i), $min\{x_i, s_j\}$ terabytes are processed and hence,

   $$\mathsf{Opt}(i, j) = \mathsf{Opt}(i - 1, j - 1) + min\{x_i, s_j\}$$

[1] ex736.816.103

When $j = 0$ (i.e., the system is rebooted on day i), no processing is done on day i. Also, the previous reboot could have happened on any of the days prior to day i. Hence,

$$\mathsf{Opt}(i, 0) = max_{k=1}^{i-1}\{\mathsf{Opt}(i-1, k)\}$$

Strictly speaking k should run from 0 to $i-1$, i.e., the last reboot could have happened either on day $i-1$ or on day $i-2$ and so on ... or on day 0 (which means no previous reboot). In our case, however, it is not advantageous to reboot on 2 successive days – you might as well do some computation on the first day and reboot on the second day. Since there is a reboot on day i, we can be sure that there is no reboot on day $i-1$, and hence k starts from 1.

The base case for the recursion is:

$$\mathsf{Opt}(0, j) = 0, \forall j = 0, 1, \ldots, n$$

A simple algorithm calculating the $\mathsf{Opt}(i, j)$ values can be designed as before taking care that i runs from 1 to n. The final value to be returned is $max_{j=1}^{n}\{\mathsf{Opt}(n, j)\}$.

Running Time: All $\mathsf{Opt}(i, j)$ values take $O(1)$ time when $j \neq 0$. $\mathsf{Opt}(i, 0)$ values take $O(n)$ time. Hence the algorithm runs in $n^2 \times O(1) + n \times O(n) = O(n^2)$ time.

3. Let $\mathsf{Opt}(i)$ denote the maximum number of bytes that can be processed starting from day 1 to day i. Suppose that the system was last rebooted on day $j < i$ ($j = 0$ means there was no reboot). Then since day $j+1$, the total number of bytes processed will be $b_{ji} = \Sigma_{k=1}^{i-j} min\{x_{j+k}, s_k\}$. (*Remember that there is no use rebooting on the last day.*) The total work processed till day i would then be $\mathsf{Opt}(j-1) + b_{ji}$. To get the maximum number of bytes processed, maximize over all values of $j < i$. Hence,

$$\mathsf{Opt}(i) = max_{j=0}^{i-1}\{\mathsf{Opt}(j)\} + b_{ji}$$

The base case:

$$\mathsf{Opt}(0) = 0$$

Compute $\mathsf{Opt}(i)$ values starting from $i = 1$ and return the value of $\mathsf{Opt}(n)$.

Running Time: Each b_{ji} value takes $O(n)$ time to calculate, and since there are $O(n^2)$ such values being calculated, the algorithm takes, $O(n^3)$ time.