**CS 344: Design and Analysis of Computer Algorithms** **Rutgers: Fall 2019**

# Homework #2 Solution

October 6, 2019

*Instructor: Sepehr Assadi*

**Problem 1.** Your goal in this problem is to analyze the *runtime* of the following (imaginary) recursive algorithms for some (even more imaginary) problem:

(A) Algorithm $A$ divides an instance of size $n$ into 3 subproblems of size $n/2$ each, recursively solves each one, and then takes $O(n)$ time to combine the solutions and output the answer.

(B) Algorithm $B$ divides an instance of size $n$ into 2 subproblems, one with size $n/2$ and one with size $n/3$, recursively solves each one, and then takes $O(n)$ time to combine the solutions and output the answer.

(C) Algorithm $C$ divides an instance of size $n$ into 4 subproblems of size $n/5$ each, recursively solves each one, and then takes $O(n^2)$ time to combine the solutions and output the answer.

(D) Algorithm $D$ divides an instance of size $n$ in to 2 subproblems of size $n-1$ each, recursively solves each one, and then takes $O(1)$ time to combine the solutions and output the answer.

For each algorithm, write a recurrence that captures its runtime and *use the recursion tree method* to solve this recurrence and find the *tightest asymptotic* upper bound on runtime of the algorithm. **(25 points)**

**Solution.** For each part, let $T(n)$ be the worst case time each of the algorithms on an input of size $n$.

(A) *Recurrence for algorithm A:* $T(n) \leq 3 \cdot T(n/2) + O(n) \leq 3 \cdot T(n/2) + c \cdot n$ for some constant $c$.

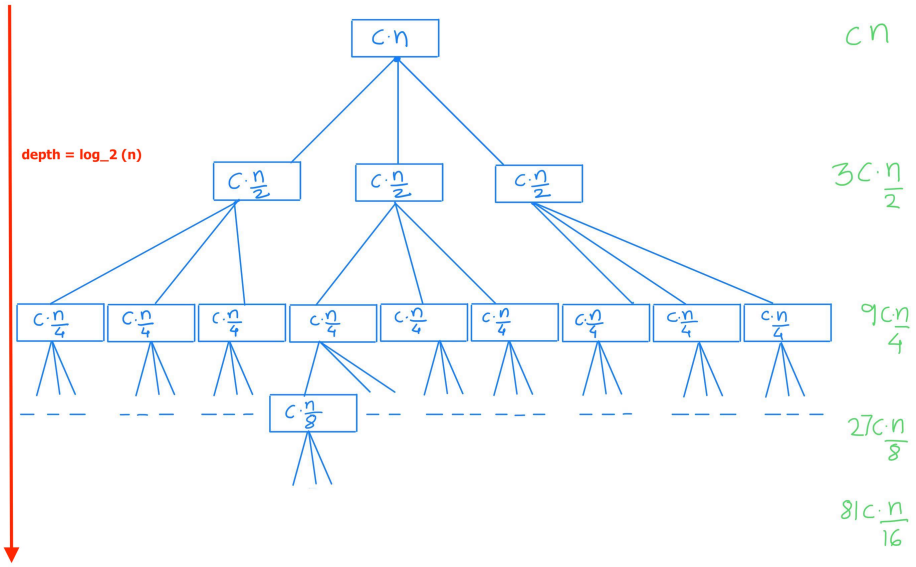


Figure 1: Recursion tree for algorithm $A$.

We have that the problem instances at each level are getting halved in size, as we go down the tree. This can happen at most $\log_2 n$ times until we reach a problem of constant size, which takes constant time to solve. Thus, there are $\log_2 n$ levels in the recursion tree (the depth of the tree is $\log_2 n$).

The root node is at depth 0. At depth $i$, there are $3^i$ problems, each of size $n/2^i$, and each such problem incurs a cost of $c \cdot n/2^i$. Thus, the total cost of the tree is:

$$\begin{aligned}
T(n) &\leq \sum_{i=0}^{\log_2 n} (3^i \cdot c \cdot \frac{n}{2^i}) = c \cdot n \cdot \sum_{i=0}^{\log_2 n} (\frac{3}{2})^i \\
&\leq c \cdot n \cdot \frac{(\frac{3}{2})^{\log_2 n + 1} - 1}{\frac{3}{2} - 1} && \text{(by the formula for sum of geometric series)} \\
&= c \cdot n \cdot 2 \cdot (\frac{3}{2}) \cdot (2^{\log_2 (\frac{3}{2}) \cdot \log_2 n} - 1) && \text{(as } a^b = 2^{\log a \cdot b} \text{ for all } a, b) \\
&\leq 3c \cdot n \cdot 2^{(\log_2 3 - 1) \cdot \log_2 n} && \text{(as } \log (\tfrac{a}{b}) = \log a - \log b \text{ for all } a, b \text{ and } \log_2 2 = 1) \\
&= 3c \cdot n \cdot n^{\log_2 3 - 1} = 3c \cdot n^{\log_2 3} = O(n^{\log_2 3}).
\end{aligned}$$

(B) *Recurrence for algorithm B:* $T(n) \leq T(n/2) + T(n/3) + O(n) \leq T(n/2) + T(n/3) + c \cdot n$ for some constant $c$.

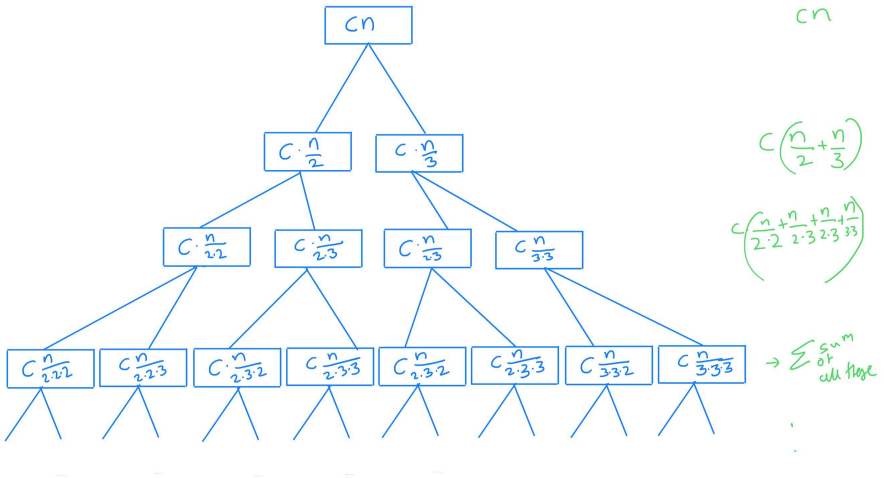


Figure 2: Recursion tree for algorithm $B$.

We have that the problem instances at each level are becoming either halved or a third in size, as we go down the tree. This is going to be an unbalanced tree, which means that the length of the shortest path from different leaves to the root node can be different.

The root node is at depth 0. At depth $i$, there are $2^i$ problems, each of different size, and each such problem incurs a cost proportional to its size while combining solutions to its subproblems. The longest distance between a leaf and the root is $\log_2 n$, the shortest one is $\log_3 n$.

Observe that the sum at a depth of i is $(1/2 + 1/3)^i \cdot c \cdot n$. This is true for all level where the tree is still full, i.e. $\log_3 n$. For the remaining levels, the sum is at most equal to this number. Since this will be a geometric series with common ratio less than one, we can upper bound by summing to infinity and have,

$$T(n) \leq \sum_{i=0}^{\infty} c \cdot n \cdot \frac{5}{6}^i = c \cdot n \cdot \frac{1}{1 - \frac{5}{6}} = O(n).$$

(C) *Recurrence for algorithm C:* $T(n) \leq 4 \cdot T(n/5) + O(n^2) \leq 4 \cdot T(n/5) + c \cdot n^2$ for some constant $c$.

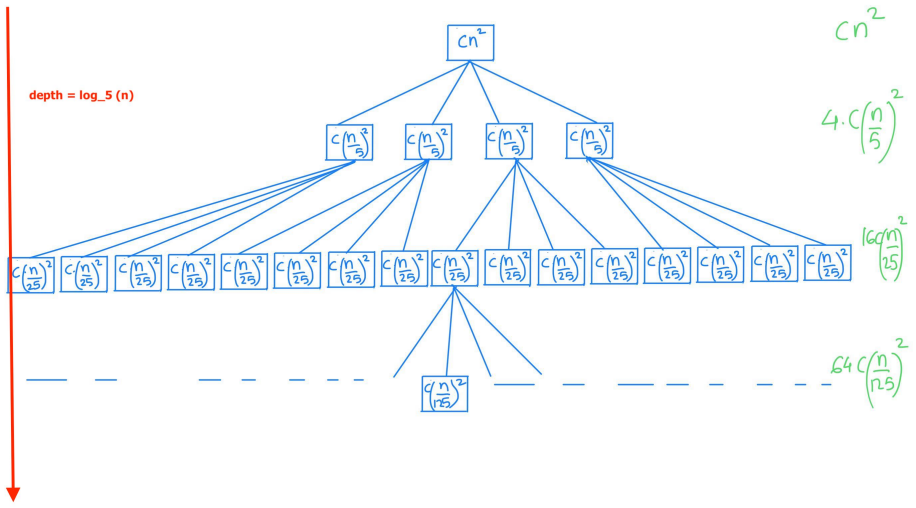


Figure 3: Recursion tree for algorithm $C$.

The problem instances at each level are becoming 1/5-th in size, as we go down the tree. Thus, there are $\log_5 n$ levels in the recursion tree (the depth of the tree is $\log_5 n$).

The root node is at depth 0. At depth $i$, there are $4^i$ problems, each of size $n/5^i$, and each such problem incurs a cost of $c \cdot (n/5^i)^2$ while combining solutions to its subproblems. Hence, by summing up all the values and since the resulting series is converging:

$$T(n) \leq \sum_{i=0}^{\log_5 n} (4^i \cdot c \cdot (\frac{n}{5^i})^2) = c \cdot n^2 \cdot \sum_{i=0}^{\log_5 n} (\frac{4}{25})^i \leq c \cdot n^2 \cdot \sum_{i=0}^{\infty} (\frac{4}{25})^i = O(n^2).$$

(D) *Recurrence for algorithm D:* $T(n) \leq 2 \cdot T(n-1) + O(1) \leq 2 \cdot T(n-1) + c$ for some constant $c$.

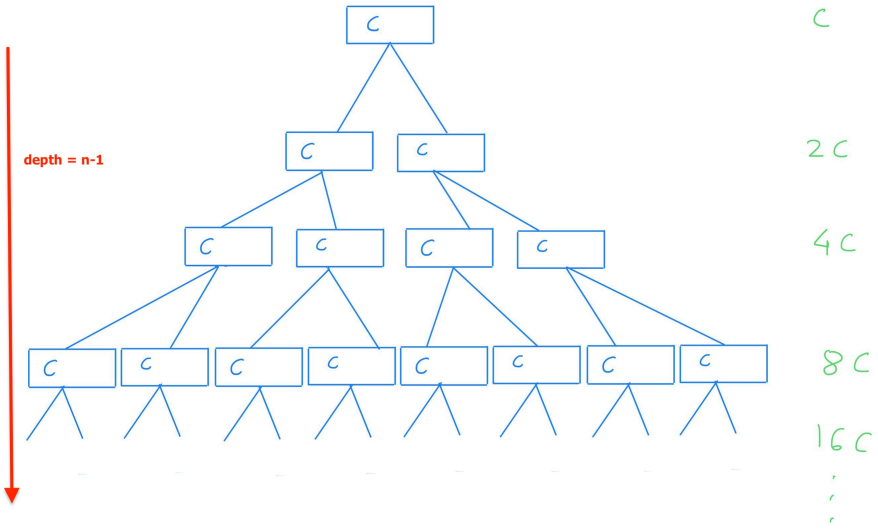


Figure 4: Recursion tree for algorithm $D$.

We have that the problem instances at each level are getting reduced by 1 in size, as we go down the

tree. This can happen at most $n-1$ times until we reach a problem of constant size, which takes constant time to solve. Thus, there are $n-1$ levels in the recursion tree (the depth of the tree is $n-1$).

The root node is at depth 0. At depth $i$, there are $2^i$ problems, each of size $n-i$, and each such problem incurs a cost of $c$ while combining solutions to its subproblems. Thus, the total cost for a problem of size $n$, using the recursion tree is

$$T(n) \leq \sum_{i=0}^{i=n-1} (2^i \cdot c) = c \cdot \sum_{i=0}^{i=n-1} (2^i) = c \cdot \frac{2^n - 1}{1} = O(2^n).$$

(by the formula for sum of geometric series)

---

**Problem 2.** You are given a permutation of $\{1, \ldots, n\}$ in an array $A[1:n]$. Design a *randomized* algorithm that finds an index of some *even* number in this array in only $O(1)$ *expected runtime*. You can assume that creating a random number from 1 to $n$ can be done in $O(1)$ time. Note that the array $A$ is already in the memory and your algorithm does not need to 'read' this array. **(25 points)**

*Example:* Suppose the input is $[3, 2, 4, 5, 1]$; then the algorithm can return either index 2 or index 3.

**Solution.** *Algorithm:* Pick a random number $p$ from $\{1, \ldots, n\}$; if $A[p]$ is even, return $p$, otherwise repeat from the beginning.

*Proof of Correctness:* The algorithm would only return a number $p$ if $A[p]$ is even.

*Runtime Analysis:* There are several ways to analyze the runtime of this algorithm. We mention two different ones below. The main observation for both is that in a permutation of $\{1, \ldots, n\}$ there are at least $\lfloor \frac{n}{2} \rfloor \geq \frac{n}{3}$ even numbers. Hence, picking a random number has a probability of roughly (but not exactly) $1/2$, and at least $1/3$ (this is a very loose lower bound but suffices for our purpose), to hit an even number.

**solution** (1) Define $S(A)$ to be the runtime of the algorithm on an array $A$. As this is a randomized algorithm, $S(A)$ is a random variable depending on the random choices. We want to compute $E[S(A)]$. The algorithm samples $p$ uniformly at random and by the observation above, this number hit an even element in $A$ with probability at least $1/3$; in this case we terminate, otherwise, with probability at most $2/3$, we solve the problem from scratch on $A$ again. This gives the following recurrence then:

$$\begin{aligned}\mathbf{E}[S(A)] &= \Pr_p(A[p] \text{ is even}) \cdot \mathbf{E}[S(A) \mid A[p] \text{ is even}] + \Pr_p(A[p] \text{ is odd}) \cdot \mathbf{E}[S(A) \mid A[p] \text{ is odd}] \\ &\leq \Pr_p(A[p] \text{ is even}) \cdot c + \Pr_p(A[p] \text{ is odd}) \cdot \mathbf{E}[S(A) + c]\,. \qquad (c \text{ is a constant})\end{aligned}$$

The above inequality is because when $A[p]$ is even we terminate which means we only spend $O(1) \leq c$ time in this case and when $A[p]$ is odd, we repeat the entire process from scratch (which takes $S(A)$ time again) but also we spend additional $O(1) \leq c$ time (because we did the sampling and checking whether $A[p]$ was even or not). By using the fact that $\Pr(A[p] \text{ is even}) = 1 - \Pr(A[p] \text{ is odd})$ and $\Pr(A[p] \text{ is even}) \geq 1/3$, we have:

$$\mathbf{E}[S(A)] \leq \Pr_p(A[p] \text{ is even}) \cdot c + \Pr_p(A[p] \text{ is odd}) \cdot \mathbf{E}[S(A) + c] \leq c + \frac{2}{3} \cdot \mathbf{E}[S(A)]\,.$$

This implies that $\mathbf{E}[S(A)] \leq 3c = O(1)$. As such, on any array $A$ which is a permutation of $\{1, \ldots, n\}$ (for all values of $n > 1$), the expected worst case runtime of the algorithm is $O(1)$.

**solution** (2) Another way to look at this problem is that the runtime of the algorithm is proportional to the number of times we need to sample $p$ before we hit an even number. Let $X$ be a random variable for this number.

As each sample has the probability of hitting an even number at least $1/3$, $\mathbf{E}[X] \leq 3$ (you may have seen this already as the Poisson distribution) as proven below

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} \Pr(X = i) \cdot i \leq \sum_{i=1}^{\infty} (\frac{2}{3})^{i-1} \cdot \frac{1}{3} \cdot i = \frac{1}{2} \cdot \sum_{i=1}^{\infty} (\frac{2}{3})^{i} \cdot i = 3. \qquad (\textstyle\sum_i^{\infty} i \cdot a^i = \frac{a}{(1-a)^2})$$

As the runtime of the algorithm, for every value of $X$, is equal to $O(X)$, we obtain that the expected worst case runtime is $O(1)$.

---

**Problem 3.** You are given an array $A[1:n]$ of $n$ positive *real* numbers (not necessarily distinct). We say that $A$ is satisfiable if after sorting the array $A$, $A[i] \geq i$ for all $1 \leq i \leq n$. Design an $O(n)$ time algorithm that determines whether or not $A$ is satisfiable. **(25 points)**

*Example:* $[1.5, 4.1, 0.5, 5.3]$ is *not* satisfiable: after sorting we get $[0.5, 1.5, 4.1, 5.3]$ and $A[1] = 0.5 < 1$. But array $[1.5, 4.1, 2.4, 5.3]$ is satisfiable: after sorting we get $[1.5, 2.4, 4.1, 5.3]$ and for $i \in \{1, 2, 3, 4\}$, $A[i] \geq i$.

**Solution.** *Algorithm:*

1. Iterate over the array and for each $1 \leq i \leq n$ replace $A[i]$ with $\lfloor A[i] \rfloor$; if $\lfloor A[i] \rfloor > n$, replace it with $n$.
2. Sort the resulting array using counting sort.
3. Iterate over the sorted array and check if $A[i] \geq i$ for all $i$: if so, output satisfiable otherwise not.

*Proof of Correctness:* Let $B$ be the resulting array after step (1) of the algorithm above. We first prove that the answer to the array $A$ and $B$ is identical always. Suppose we sort both $A$ and $B$ and in the following we refer to the sorted versions of these two arrays.

For any entry $i$, if $n \geq A[i] \geq i$, then $B[i] \geq i$ also: this is because $i$ is an integer and $B[i] = \lfloor A[i] \rfloor$. Moreover, if $A[i] > n$, then since the largest index $i$ can anyway be equal to $n$, by setting $B[i] = n$ (in the algorithm), we still have $B[i] \geq i$. This implies that if $A$ is satisfiable then $B$ is also satisfiable. Finally, note that since $B[i] \leq A[i]$ for all $i$, if $B$ is satisfiable then $A$ will be satisfiable also, proving that the answer to both arrays will always be the same.

The rest of the proof follows easily because we sort the array $B$ in the algorithm and check whether it is satisfiable or not directly on the sorted array.

*Runtime Analysis:* The first line of the algorithm takes $O(n)$ time to iterate over the array. The second line involves counting sort on an array of $n$ non-negative integers with largest value equal to $n$ which can be done in $O(n)$ time (recall that in the class, we analyzed counting sort on positive integers; to extend it to non-negative integers, i.e., allowing for inclusion of 0 also, simply add all numbers in the array by one to make them positive, sort them, and then subtract them by one). The final step also takes $O(n)$ time, hence making the total runtime $O(n)$.

---

**Problem 4.** Alice and Bob are delivering for the Pizza store that has $n$ orders today. From past experience, they both know that if Alice delivers the $i$-th order, the tip would be $A[i]$ dollars, and if Bob delivers, the tip would be $B[i]$ dollars. Alice and Bob are close friends and their goal is to maximize the total tip they can get and then distribute it evenly between themselves. The problem is that Alice can only handle $a$ of the orders and Bob can only handle $b$ orders. They do not know how to do this so they ask for your help.

Design a dynamic programming algorithm that given the arrays $A[1:n], B[1:n]$, and integers $1 \leq a, b \leq n$, finds the largest amount of tips that Alice and Bob can collect together subject to their constraints. The runtime of your algorithm should be $O(n \cdot a \cdot b)$ in the worst case. **(25 points)**

*Example:* Suppose $A = [2, 4, 5, 1]$, $B = [1, 7, 2, 5]$, $a = 2$ and $b = 1$. Then the answer is 14 by Alice delivering $2, 3$ and Bob delivering 4 (so $4 + 5 + 5 = 14$), or Alice delivering $1, 3$ and Bob delivering 2 ($2 + 5 + 7 = 14$).

**Solution.** *Specification of the recursive formula (in plain English):*

For every $0 \leq i \leq n$, and $0 \leq j \leq a$, and $0 \leq k \leq b$, we define:

- $T(i, j, k)$: the maximum tip Alice and Bob can collect together by delivering a subset of $\{1, \ldots, i\}$ assuming Alice can deliver at most $j$ delivery and Bob can deliver at most $k$ delivery.

The final solution to the problem is then $T(n, a, b)$.

*Recursive solution for the formula:* We design the following recursive solution for all values of $i, j, k$:

$$T(i,j,k) = \begin{cases} 0 & \text{if } i = 0 \text{ or } \textit{both } j = k = 0 \\ \max\{T(i-1,j,k), T(i-1,j-1,k) + A[i]\} & \text{if } i, j > 0 \text{ but } k = 0 \\ \max\{T(i-1,j,k), T(i-1,j,k-1) + B[i]\} & \text{if } i, k > 0 \text{ but } j = 0 \\ \max\{T(i-1,j,k), T(i-1,j-1,k) + T(i-1,j,k-1) + B[i]\} & \text{otherwise} \end{cases}$$

We now prove the correctness of this formula for each part separately.

- If $i = 0$: there is no delivery left, hence Alice and Bob cannot collect any more tip. Similarly, if both $j = k = 0$, then neither Alice nor Bob can deliver any more and thus they will not collect any more tips. Hence, the first condition of the formula is correct.
- If $i, j > 0$ but $k = 0$: in this case, Alice and Bob have two different options: (1) Simply skip delivery $i$ altogether. Since we are considering deliveries in $\{1, \ldots, i\}$, after skipping delivery $i$, we are left with $\{1, \ldots, i-1\}$ and neither Alice nor Bob used any of their deliveries (so $j, k$ will not change) and they will not collect any tip from $i$. As such, by choosing this option, they will collect $T(i-1, j, k)$ tip from the remaining deliveries. (2) Alice delivers $i$: in this case they are again left with deliveries $\{1, \ldots, i-1\}$ but now Alice can only deliver $j - 1$ deliveries although they now collect a tip of $A[i]$ from Alice delivering this one. Hence, the will collect $T(i-1, j-1, k) + A[i]$ tip in this case. Finally, there is no other option left for Alice and Bob because $k = 0$ and thus Bob cannot deliver anymore.
- If $i, k > 0$ but $j = 0$: this is identical to the above case with the difference that the second option is now Bob delivering $i$ and hence collecting $B[i]$ tips which in total becomes $T(i-1, j, k-1) + B[i]$.
- Otherwise, i.e., if all $i, j, k > 0$: again, this case is a combination of all the options above: (1) no one delivers $i$, hence collecting $T(i-1, j, k)$, (2) Alice delivers $i$ hence collecting $T(i-1, j-1, k) + A[i]$, and (3) Bob delivers $i$, hence collecting $T(i-1, j, k-1) + B[i]$, the same exact way as the recursive formula.

This concludes the proof of correctness.

*Algorithm:* There are two different ways of giving this dynamic programming algorithm: doing either a (top-down) memoization or a bottom-up dynamic programming. Both are valid solutions and you only need to write down one of them as your solution. But for completeness, we give both solutions here:

**solution** (1) *Memoization*: We pick an array $T[0:n][0:a][0:b]$ initially set to 'undefined'. Then we define the following memoize algorithm:

`memT`$(i, j, k)$:

(a) If $T[i][j][k] \neq$ 'undefined' return $T[i][j][k]$.

(b) If $i = 0$ or $j = k = 0$, then $T[i][j][k] = 0$;

(c) Else, if $i, j > 0$ and $k = 0$, $T[i][j][k] = \max\{\texttt{memT}(i-1, j, k), \texttt{memT}(i-1, j-1, k) + A[i]\}$;

(d) Else, if $i, k > 0$ and $j = 0$, $T[i][j][k] = \max\{\texttt{memT}(i-1, j, k), \texttt{memT}(i-1, j, k-1) + B[i]\}$;

(e) Else, $T[i][j][k] = \max\{\texttt{memT}(i-1, j, k), \texttt{memT}(i-1, j-1, k) + A[i], \texttt{memT}(i-1, j, k-1) + B[i]\}$.

(f) Return $T[i][j][k]$.

The answer to the problem now follows from $\texttt{memT}(n, a, b)$. The correctness already follows from the previous part.

**solution** (2) *Bottom-Up Dynamic Programming*: We use the following bottom-up dynamic programming solution. The main step is to determine the *evaluation order* of computing subproblems so that we always have the correct value of the recursive (inner) subproblem once we want to compute the outer one. It is easy to verify that $T(i, j, k)$ is updated only from entries $T(i', j', k')$ where all $i' < i$, $j' \leq j$, and $k' \leq k$ (of course not from all such entries). So we only need to make sure smaller values of $(i, j, k)$ triples are computed first.

(a) We pick an array $T[0 : n][0 : a][0 : b]$ initially set be all 0 (to account for the base cases directly).

(b) For $i = 1$ to $n$:

For $j = 1$ to $a$:

For $k = 1$ to $b$:

$T[i][j][k] = \max\{T[i-1][j][k], T[i-1][j-1][k] + A[i], T[i-1][j][k-1] + B[i]\}$.

(c) Return $T[n][a][b]$.

The correctness follows from the fact that we use a valid evaluation order and thus whenever $T[i][j][k]$ is computed, the cells it is updated from already has their correct value.

*Runtime analysis*: There are $O(n \cdot a \cdot b)$ subproblems and each one takes $O(1)$ to compute, hence the total runtime is $O(n \cdot a \cdot b)$.

---