CS 577-3: Introduction to Algorithms Fall 2022

Practice Problems for Midterm Exam 1

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

Here are some problems that you can use to prepare for the exam. Also, don't forget the regular problems on the homework assignments that you haven't done yet.

1. Let P be a set of n distinct points in the plane. A point (x, y) in P is called undominated if for every other point (x', y') in P, either $x' < x$ or $y' < y$ (or both).

   Design an $O(n \log n)$ algorithm for finding all of the undominated points.

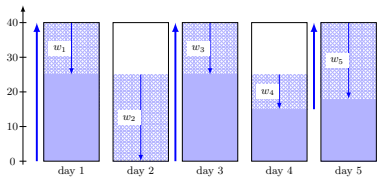
2. You are given a binary string $a[1 \ldots n]$ and want to find three ones that are at equal distance from each other, i.e., three positions $1 \le i < j < k \le n$ such that $a[i] = a[j] = a[k] = 1$ and $j - i = k - j$.

   Design an $O(n \log n)$ algorithm that finds such a triple or reports that none exists.

   Hint: Use polynomial multiplication.

3. You run a chemical lab, and are planning the supply of purified water for a period of n days. You know that you will need $w_i \le 100$ gallons during day i. To accommodate this, you will acquire a storage tank with a capacity of some c gallons. The tank will be filled to capacity some days throughout the period, each time at the start of the day. It is always filled to capacity at the start of the first day. It should never happen that the demand for a day cannot be met.

   As an example, consider a tank with capacity $c = 40$ and a period of $n = 5$ days with $w_1 = 15$, $w_2 = 25$, $w_3 = 15$, $w_4 = 10$, and $w_5 = 22$. As illustrated below, it works to have fills at the start of days 1, 3, and 5.



   (a) One problem is deciding how large the capacity c of the tank should be for a given bound on the number of fills. You know that an optimal schedule in this setting is to postpone each fill as much as possible without letting the supply run out. Given n, f, and the values w_i for $i \in [n]$, what is the minimum capacity $c \in \mathbb{N}$ so that this schedule leads to no water shortages and at most f fills?

   Continuing the example above, for $f = 3$ the figure indicates that $c = 40$ works. If we try $c = 39$, then we need to fill at the start of days 1, 2, 3, and 5. So, the answer is $c = 40$.

   Design an algorithm that runs in time $O(n \log n)$.

   (b) Some of the days a fill would be more disruptive for the lab than others. You have quantified these as a daily loss of revenue ℓ_i that the lab incurs when a fill happens at the start of day i for $i \in [n]$. Given c, n, and the values w_i and ℓ_i for $i \in [n]$, you would like to know how small you can make your total loss of revenue due to the fills without having any supply shortage. As the tank is always filled at the start of day 1, you always incur a cost of at least ℓ_1. Design an algorithm that runs in time $O(n^2)$ and space $O(n)$.

   As an example, consider the above setting with $\ell_1 = 0$, $\ell_2 = 1$, $\ell_3 = 100$, $\ell_4 = 20$, and $\ell_5 = 30$. The schedule in the figure costs $\ell_1 + \ell_3 + \ell_5 = 130$ loss in revenue. By instead refilling at the start of days 1, 2, and 4, the loss in revenue is reduced to $\ell_1 + \ell_2 + \ell_4 = 21$. This is optimal, so the answer is 21.

   (c) Assume you have an algorithm for part (a) that you can call as a blackbox. Design an algorithm that retrieves a solution and outputs a fill schedule that achieves the minimum total loss in revenue in (a). Your algorithm can make $O(n)$ calls to the blackbox and spend $O(n)$ time outside of the calls.

   For the example from part (a), the output for would be: days 1, 2, and 4.

4. In the two-player game "Two Ends", n cards are laid out in a row. On each card, face up, is written a positive integer. Players take turns removing a card from either end of the row and placing the card in their pile, until all cards are removed. The score of a player is the sum of the integers of the cards in his/her pile.

   Design an algorithm that takes the sequence of n positive integers, and determines the score of each player when both play optimally. Your algorithm should run in time $O(n^2)$ and space $O(n)$.

5. You are given an arithmetic expression containing n integers and $n - 1$ operators, each either $+$, $-$, or $\times$. You are also given a positive integer m. Your goal is to find an order to perform the operations such that the result is a multiple of m, or report that no such order exists.

   For example, for the expression $6 \times 3 + 2 \times 5$, this is possible for $m = 7$, namely as follows: $(6 \times 3) + (2 \times 5) = 28$. For the same expression this is not possible for $m = 8$ as none of the five possible orderings yield a multiple of 8: $(6 \times 3) + (2 \times 5) = 28$, $((6 \times (3 + 2)) \times 5) = 150$, $((6 \times 3) + 2) \times 5 = 100$, $6 \times (3 + (2 \times 5)) = 78$, $6 \times ((3 + 2) \times 5) = 150$.

   Design an algorithm that runs in time polynomial in n and m.

CS 577-3: Introduction to Algorithms Fall 2022

Solutions to the Midterm 1 Practice Problems

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

Problem 1

We use divide-and-conquer. We need to determine how to divide the problem into some number of subproblems. We can do this by dividing the set of points into two halves based on their coordinates. Define an ordering of the points based on the x-coordinate with ties broken by the y-coordinate. More precisely, $(x, y) \le (x', y')$ if and only if $x < x'$, or $x = x'$ and $y \le y'$. The "left" subset contains the points less than or equal to the median, and the "right" subset the points greater than the median. Call these sets P_L and P_R, respectively. We recursively compute the sets of undominated points for P_L and P_R and then combine these two subsets in such a way that we end up with the set of undominated points for the whole set.

Let the two computed undominated sets be U_L and U_R. The set U of undominated points for the entire set P is a subset of $U_L \cap U_R$. To figure out which points in $U_L \cap U_R$ are in U, we make use of the fact that all points in U_R are larger than all points in U_L:

- Points in U_R are not dominated by any points in P_L. This holds since for points $p_l \in P_L$ and $p_r \in U_R$, either the x-coordinate of p_r is greater than the one for p_l or they are the same and the y-coordinate for p_r is greater.
- A point in U_L is not dominated by any point in P_R if and only if its y-coordinate is greater than the maximum y-coordinate in P_R, which is also the maximum y-coordinate in U_R.

Since we can calculate the maximum y-coordinate of U_R in a linear scan, we can combine the solutions of the subproblems to produce the complete solution U in linear time.

Algorithm Our algorithm makes use of the SPLIT procedure, which can be implemented in linear time using the linear-time algorithm for finding the median.

SPLIT(P)

Input: A nonempty set of distinct points P
Output: Two sets P_L and P_R which partition P, and where:

- $|P_L| = \lfloor |P|/2 \rfloor$
- $|P_R| = \lceil |P|/2 \rceil$
- $(x, y) \in P_L, (x', y') \in P_r \Rightarrow (x < x' \vee (x = x' \wedge y < y'))$

Correctness If $|P| = 1$, then the sole element is undominated. Otherwise, $|P| > 1$, and we use the above recursive approach, the correctness of which was argued in the two bullet points above.

Algorithm 2 UNDOMINATED(P)

Input: A nonempty set of distinct points P
Output: The set of undominated points in P

```
1:  procedure UNDOMINATED(P)
2:      if |P| = 1 then
3:          return P
4:      else
5:          (P_L, P_R) ← SPLIT(P)
6:          (U_L) ← UNDOMINATED(P_L)
7:          (U_R) ← UNDOMINATED(P_R)
8:          U ← U_R
9:          max ← largest y-coordinate of a point in U_R
10:         for each element (x, y) in U_L do
11:             if y > max then
12:                 add (x, y) to U
13:         return U
```

Complexity We use the recursion tree method. Each non-leaf node in the recursion tree has two children, and the input size n shrinks by half with each level of recursion, so the tree has the same shape as MERGESORT. At any internal node in the tree, the non-recursive work done is the call to SPLIT plus some linear amount of work. The leaves do constant work. As in the analysis of MERGESORT, it follows that the total work done is $O(n \log n)$.

Note that since our algorithm takes $O(n \log n)$ time anyways, we could have sorted the points according to the order we defined by first sorting them by their x-coordinate, and sort those with the same x-coordinate by their y-coordinate. We can then directly access the median element and split P into P_L and P_R without using linear-time selection as in SPLIT.

Problem 2

We observe that

$$\begin{aligned}
&(\exists i, j, k \in [n])\, a[i] = a[j] = a[k] = 1 \text{ and } j - i = k - j \neq 0 \\
\Leftrightarrow\ &(\exists i, j, k \in [n])\, a[i] = a[j] = a[k] = 1 \text{ and } i + k = 2j \text{ and } i \neq k \\
\Leftrightarrow\ &(\exists j \in [n])\, a[j] = 1 \text{ and } (\exists i, k \in [n])\, (a[i] = a[k] = 1 \text{ and } i + k = 2j \text{ and } i \neq k) \\
\Leftrightarrow\ &(\exists j \in [n])\, a[j] = 1 \text{ and coefficient of } x^{2j} \text{ in } (p_a(x))^2 \text{ exceeds } 1,
\end{aligned}$$

where $p_a(x) \doteq \sum_{d \in [n]} a[d] x^d$. The last step follows because

$$\begin{aligned}
(p_a(x))^2 &= \sum_{i \in [n]} a[i] x^i \cdot \sum_{k \in [n]} a[k] x^k \\
&= \sum_{i,k \in [n]} a[i] a[k] x^{i+k} \\
&= \sum_s c_s x^s \text{ where } c_s = \sum_{i,k \in [n] : i+k=s} a[i] a[k],
\end{aligned}$$

which shows that the coefficient x^{2j} in $(p_a(x))^2$ equals the number of pairs $(i,j) \in [n] \times [n]$ with $i + k = 2j$ for which $a[i] = a[k] = 1$. If $a[j] = 1$, this number always has a contribution of 1 from the pair $(i,k) = (j,j)$, and has another contribution iff there exist $i, k \in [n]$ with $i \neq k$ such that $a[i] = a[k] = 1$.

The above observation leads to the following pseudocode for finding a suitable value of $j \in [n]$ (or report that none exists):

$p_a(x) \leftarrow \sum_{d \in [n]} a[d]x^d$
$c(x) \doteq \sum_s c_s x^s \leftarrow p_a(x) \cdot p_a(x)$
$j \leftarrow 1$
while $j \leq n$ **cand** $(a[j] = 0$ **or** $c_{2j} = 1)$ **do** $j \leftarrow j + 1$
if $j \leq n$ **then**
    return j
else
    return no solution

Once we found a suitable $j \in [n]$ (if any), we can find suitable values of $i, k \in [n]$ by trying equidistant positions on both sides of j until $a[i] = a[k] = 1$, which is guaranteed to happen by the choice of j.

$i \leftarrow j - 1$; $k \leftarrow j + 1$
while $a[i] = 0$ **or** $a[k] = 0$ **do**
    $i \leftarrow i - 1$; $k \leftarrow k + 1$
return (i, j, k)

Correctness follows from the above observation. As for the running time, it takes $O(n \log n)$ time to compute $c(x)$ from $p_a(x)$ using polynomial multiplication via the FFT. All other operations run in time $O(n)$: constructing $p_a(x)$, the linear search for a suitable $j \in [n]$, and the linear search for a suitable pair $(i,k) \in [n] \times [n]$ for the found value of j.

Problem 3

Part (a)

We can test whether a given candidate capacity $c \in \mathbb{N}$ is sufficient by simulating the optimal schedule day by day and making sure we never have a water shortage (because the demand w_i exceeds the capacity c) and do not need more than f fills in total. This can be done in time $O(n)$. For completeness we provide the pseudocode below, where the variable t keeps track of the number of fills thus far, and W the current water level of the tanks.

Note that if capacity c suffices then every larger capacity also suffices. Thus, we can determine the smallest capacity that suffices using a binary search.

As the daily demand is no larger than 100 and there are n days, a capacity of $100n$ definitely suffices. Therefore, the range for c can be limited to all nonnegative integers up to $100n$. The resulting binary search takes $\lceil \log_2(100n + 1) \rceil = O(\log n)$ rounds. Each round involves running Capacity-Suffices once and takes $O(n)$ time. Thus, the overall running time is $O(n \log n)$.

Algorithm 3 Capacity-Suffices$(n, r, c, w_1, \ldots, w_n)$

Input: $n, f, c \in \mathbb{N}$, $w_i \in \mathbb{R}$ for $i \in [n]$
Output: whether capacity c is sufficient to accommodate the demands $w_1, \ldots, w_n$ for n successive days with complete fills at the start of at most f days including the first day.

$W \leftarrow c$
$t \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
    if $w_i > c$ **or** $(w_i > W$ **and** $t \geq f)$ **then**
        return false
    if $w_i \leq W$ **then**
        $W \leftarrow W - w_i$
    else
        $W \leftarrow c - w_i$
        $t \leftarrow t + 1$
return true

Part (b)

We can apply the principle of optimality based on the fact that a refill on day d breaks up the problem into two independent subproblems of the same type, namely the first $d - 1$ days and the last $n - d$ days.

Moreover, if we consider the *first* refill d (if any), then the loss in revenue during the first $d - 1$ days is ℓ_1. It only remains to add the minimum total loss in revenue for the subproblem defined by the last $n - d$ days. Note that having the first refill on day d is feasible iff the total demand during the first $d - 1$ days does not exceed the capacity c. The case of no refill can be considered as having a refill on day $n+1$ at no cost. The result is the minimum over all feasible choices $d \in \{2, \ldots, n+1\}$ for the first refill.

The subproblems that arise throughout the recursion all correspond to suffixes of $[n]$. This leads to the following subproblems for $k \in [n + 1]$: $\mathrm{OPT}(k)$ denotes the minimum total loss in revenue for the period consisting of days $\{k, \ldots, n\}$, or ∞ if there is no feasible schedule. Note that the latter will happen iff at least one of the daily demands exceeds the capacity c.

By the above discussion, we have the following recurrence:

$$\mathrm{OPT}(k) = \min\left(\infty, \min_{k<d\leq n+1 : \sum_{i=k}^{d-1} w_i \leq c} (\ell_k + \mathrm{OPT}(d))\right). \tag{1}$$

The recurrence allows us to compute the values $\mathrm{OPT}(k)$ from $k = n$ down to $k = 1$ using the initialization $\mathrm{OPT}(n + 1) = 0$. We return $\mathrm{OPT}(1)$.

Each application of the recurrence involves taking the minimum over $O(n)$ terms. Checking the condition $\sum_{i=k}^{d-1} w_i \leq c$ for a given term takes time $O(n)$ by itself. However, by keeping track of a running sum, the total time for checking the condition can be kept to $O(n)$ for a single application of the recurrence. Thus, the running time per application of the recurrence is $O(n)$. As there are $O(n)$ applications, the overall running time is $O(n^2)$.

For completeness, here is pseudocode for evaluating (1).

Apart from the memory space to store the n values of the array OPT, the evaluation of (1) as described above only takes space $O(1)$. Thus, the overall space need is $O(n)$.

Algorithm 4

$L \leftarrow \infty$; $W \leftarrow c$; $d \leftarrow k + 1$
while $d \leq n + 1$ **cand** $w_{d-1} \leq W$ **do**
    $L \leftarrow \min(L, \mathrm{OPT}(d))$; $W \leftarrow W - w_d$; $d \leftarrow d + 1$
return $\ell_k + L$

Part (c)

We use the OPT notation from part (a). We only need to consider the case where $\mathrm{OPT}(1) < \infty$; there is no optimal schedule otherwise.

The first fill day is always day 1. The first refill day d in an optimal schedule can be constructed by figuring out a choice of d on the right-hand side of (1) that yields the minimum for $k = 1$ and satisfies the constraint that $\sum_{i=k}^{d-1} w_i \leq c$; there is no refill if $d = n + 1$ yields the minimum. If we search for the first $d > 1$ for which $\mathrm{OPT}(1) = \ell_1 + \mathrm{OPT}(d)$, the constraint is guaranteed to be satisfied. If $d \leq n$, we do a refill on day d, and proceed with $k = d$ to find the second refill day in the same way, etc., until we hit $d = n + 1$.

For completeness, here is pseudocode implementing this approach.

Algorithm 5

$S \leftarrow \emptyset$; $k \leftarrow 1$
for $d = 2$ to $n + 1$ **do**
    if $\mathrm{OPT}(k) = \ell_k + \mathrm{OPT}(d)$ **then**
        $S \leftarrow S \cup \{k\}$; $k \leftarrow d$
return S

Each use of an OPT value can be replaced by a call to the corresponding instance of problem (a). By storing the OPT values, the number of calls to the blackbox for (a) is no more than the different OPT values, which is $n+1$. As d increases by one in each iteration of the pseudocode and the amount of work per iteration outside of the calls is $O(1)$, the total time spent outside of the calls is $O(n)$.

Problem 4

Since both players are playing optimally, Alice will choose the move that would maximize her *overall* gain, and Bob on the other hand, will choose the move that would minimize Alice's *overall* gain.

Let $A(i,j)$ be the optimal gain for Alice when it is Alice's turn and the numbers remaining are $a_i, a_{i+1}, \ldots, a_j$. Here Alice only has two choices: (1) pick a_i or (2) pick a_j. Alice will choose the one with larger overall gain.

(1) Alice picks a_i: Bob will have $a_{i+1}, a_{i+2}, \ldots, a_j$ to choose from. Here Bob also has two choices: pick a_{i+1} or a_j. If Bob chooses a_{i+1}, then Alice's gain would be $A(i+2, j)$; and if Bob chooses a_j, Alice's gain would be $A(i+1, j-1)$. Bob will choose the one that could minimize Alice's gain, and thus Alice will get the minimum of $A(i+2, j)$ and $A(i+1, j-1)$ because of Bob's strategy, thus yielding an overall gain of

$$a_i + \min\{A(i+2, j), A(i+1, j-1)\}$$

in this case.

(2) Alice picks a_j: by a similar argument Alice would achieve an overall gain of

$$a_j + \min\{A(i, j-2), A(i+1, j-1)\}.$$

Between the two choices (1) and (2), Alice will choose the larger one. Therefore, we have

$$A(i,j) = \max\{a_i + \min\{A(i+2, j), A(i+1, j-1)\}, \\ a_j + \min\{A(i, j-2), A(i+1, j-1)\}\}.$$

The base cases are $A(i,i) = a_i$ and $A(i, i+1) = \max\{a_i, a_{i+1}\}$, where Alice only has 1 or 2 numbers to choose from. One can use an $n \times n$ table to compute all values of $A(i,j)$ and return $A(1,n)$ as the final answer for Alice in time $O(n^2)$. The final gain for Bob will then be

$$\sum_{i=1}^{n} a_i - A(1,n).$$

Note the space complexity can be further reduced to $O(n)$ by only keeping the previous two diagonals.

Problem 5

Let a_i indicate the i-th number, and $\circ_{i,i+1}$ be the operator between the i-th and the $(i+1)$-th number. For $i \leq j$, define $\mathrm{OPT}(i,j)$ to be the set of numbers that can be produced from the subexpression from the i-th number to the j-th number modulo m. We want to know if 0 is in $\mathrm{OPT}(1,n)$.

We can compute $\mathrm{OPT}(i,j)$ using the following recurrence. For two sets of numbers modulo m, M_1 and M_2, and operator $\circ$, define $M_1 \circ M_2$ to be a new set consisting of $(m_1 \circ m_2 \bmod m)$ for each choice of m_1 in M_1 and m_2 in M_2. Then we have

$$\mathrm{OPT}(i,j) = \begin{cases} \{a_i \bmod m\} & i = j \\ \bigcup_{i \leq k < j} \mathrm{OPT}(i,k) \circ_{k,k+1} \mathrm{OPT}(k+1, j) & i < j \end{cases} \tag{2}$$

Time and Space Analysis There are $O(n^2)$ subproblems to solve. For each subproblem, we need to compute the results of $O(n)$ pairs of sets. Each set in a pair contains at most m numbers, so for fixed k, computing $\mathrm{OPT}(i,k) \circ_{k,k+1} \mathrm{OPT}(k+1, j)$ takes no more than $O(m^2)$ time.[1] The whole algorithm therefore takes $O(n^3 m^2)$ time to compute, which is polynomial.

As for space, there are $O(n^2)$ unique subproblems. Each subproblem takes $O(m)$ space because it stores all possible values for the subexpression modulo m. So our space complexity is $O(n^2 m)$. We cannot improve our space complexity by using the iterative approach and "forgetting" about part of our OPT table along the way because we look back at all split points k for $i \leq k < j$.

[1] There are data structures that allow for this step to be faster, but this analysis is sufficient for this problem.

Solution Retrieval Option 1 In order to retrieve our solution (the ordering of operations such that the result is a multiple of m) recall that we have access to our 2D OPT table. Now we can walk through the array from $\text{OPT}(1, n)$ to our base cases, seeing which choice we would have made at each step. For starters, if $\text{OPT}(1, n)$ does not include 0, then we can say that there was no possible ordering of operations that resulted in a multiple of m. Otherwise, we will find out our last operation (as well as the values modulo m that the left and right subexpressions need to evaluate to) by looking through all of the options considered in our recurrence. Once we find the last operation, we then recursively compute the last operations used in the left and right subexpressions to yield their values modulo m, continuing until we reach our base cases.

In the general case that we are trying to compute the last operation used for the subexpression from the i-th number to the j-th number such that the subexpression evaluates to r, this means considering all operations $\circ_{k,k+1}$ for $i \leq k < j$ as well as all values in the sets $\text{OPT}(i, k)$ and $\text{OPT}(k + 1, j)$. Specifically, we need to find a value k such that there exists an $x \in \text{OPT}(i, k)$ and a $y \in \text{OPT}(k + 1, j)$ where $(x) \circ_{k,k+1} (y) = r$ modulo m. Once we have found such a value k, we can add $\circ_{k,k+1}$ to our ordering of operations. We then compute the last operation used for the subexpression from the i-th number to the k-th number such that the subexpression evaluates to x, and we compute the last operation used for the subexpression from the $(k+1)$-st number to the j-th number such that the subexpression evaluates to y. We provide pseudocode in Algorithm 6 for completeness. The final output is given by $\textsc{RecoverSolution}(1, n, 0)$.

Algorithm 6

Input: indices $1 \leq i \leq j \leq n$ indicating the subexpression to consider, and a target value r. Also, global memory access to the original input and the table OPT filled according to its specification.

Output: A list $L = (\ell_1, \ldots, \ell_{n-1})$ indicating the order in which operations must be evaluated so that $a_i \circ_{i,i+1} \cdots \circ_{j-1,j} a_j$ evaluates to v modulo m. More precisely, operation $\circ_{\ell_i,\ell_i+1}$ needs to be the i-th operation evaluated in the expression.

```
1: procedure RecoverSolution(i, j, r)
2:     if i = j then
3:         Return ∅                                   ▷ No operation to be made
4:     for k = i to j − 1 do
5:         for (x, y) ∈ OPT[i, k] × OPT[k + 1, j] do
6:             if x ∘_{k,k+1} y = r (mod m) then
7:                 L ← RecoverSolution(i, k, x)       ▷ find sequence to obtain x in (i, k)
8:                 R ← RecoverSolution(k + 1, j, y)   ▷ find sequence to obtain y in (k + 1, j)
9:                 return (L, R, k)                   ▷ put it all together (∘_{k,k+1} should be last)
```

The key difference between calculating the time complexity for retrieving the optimal ordering and the time complexity for finding if an ordering is possible is the number of subexpressions we consider. We don't have to consider a subexpression for all $O(n^2)$ pairs $\text{OPT}(i, j)$, we only have to consider the subexpressions on the path to our optimal solution. Think of a recursion tree analysis in which each node is a subexpression we consider. The top node corresponds to $\text{OPT}(1, n)$ and the n leaf nodes correspond to $\text{OPT}(i, i)$. At each level of the recursion tree, we have split our expression into some number of segments. Within each segment, we try a number of split points equal to the length of the segment. So the total number of split points considered at a given level

of the recursion tree is $O(n)$. For each split point, the left and right sets each contain at most m numbers, so computing $(x) \circ_{k,k+1} (y)$ takes $O(m^2)$. In all, we do $O(nm^2)$ work per level and there are at most n levels, giving us a time complexity for retrieving the solution of $O(n^2m^2)$.

Solution Retrieval Option 2 Although the time complexity for the retrieval performed above does not affect the overall complexity, we can speed up our approach by keeping track of more information during the computation of the OPT table and using an appropriate data structure. In the cell $\text{OPT}(i, j)$ we could store not only the set of possible result values for a subexpression but also the values of k, x, and y that realized each possible result value (k being the split point, x the value of the left subexpression, and y the value of the right subexpression).

In this case, when we walk through the OPT table to retrieve our solution, we only need to look through the possible values for a subexpression, find the one that we want, and say that we used the operator corresponding to $\circ_{k,k+1}$, continuing on to investigate how we could have yielded x from the subexpression $i...k$ and y from the subexpression $(k + 1)...j$ in the same manner. This way, we need to access no more than n cells of the OPT table. In each cell we just need to locate the information for the desired result value. If we represent the set of possible values by a characteristic array of length m, we can locate the information in time $O(1)$. Thus, he running time for retrieval becomes $O(n)$. Note that this approach requires the same number of cells for the OPT table, but each cell contains more memory space: $O(\log n + 2 \log m)$ bits instead of 1 bit.