References

Collaborated with Crystal Huang.

Question 1: Fast sorting

1. Assume you are given an array of n integers in the range $\{1, \ldots, (\log n)^{\log n}\}$. Show how to sort this array in time $O(n \log \log n)$.

Solution: *Definitions.* Let RadixSort(A, b) denote the well-known radix sorting algorithm that sorts an array of n keys $A[1, \ldots, n]$ using radix b. By definition, RadixSort(A, b) sorts $A[1, \ldots, n]$ in time $\Theta((n + b) \cdot w)$, where w is the maximum key length in A.

Let $A^*[1, \ldots, n]$ be an n-element array where, for all $a \in A^*$, we have $a \in \mathbb{Z}$ and $1 \leq a \leq (\log n)^{\log n}$.

Claim. RadixSort(A^*, n) sorts $A^*[1, \ldots, n]$ in time $O(n \log \log n)$.

Proof. For all $a \in A^*$, the length of a in radix b is $\log_b a$. Thus, for all $a \in A^*$, the length of a in radix $b = n$ is $\log_n a$. By construction, the maximum value in A^* is $(\log n)^{\log n}$. Thus w, the maximum key length in A^* is

$$\begin{aligned} w &= \log_b (\log n)^{\log n} \\ &= \log_n (\log n)^{\log n} \\ &= \log n \log_n (\log n) \\ &= \log n \cdot \frac{\log \log n}{\log n} \\ &= \log \log n. \end{aligned}$$

Now the running time of RadixSort(A^*, n) is

$$\begin{aligned} \Theta((n + b) \cdot w) &= \Theta((n + n) \cdot \log \log n) \\ &= \Theta(n \log \log n) \\ &= O(n \log \log n). \end{aligned}$$

Hence, we can sort $A^*[1, \ldots, n]$ using RadixSort(A^*, n) in $O(n \log \log n)$ time. □

2. Assume you are given an array of n integers with many duplicates, so that you know that there are at most $\log n$ distinct elements in the array. Show how to sort this array in time $O(n)$ if you can use an arbitrary amount of memory, i.e., arrays of arbitrary sizes).

Solution:

Axiom I. Assume that an arbitrarily-sized array can be initialized with all-0 values in constant time.

Axiom II. Assume that a dynamically-sized list can be initialized in constant time.

Algorithm I. The well-known algorithm MergeSort(L) where L is an n-element list. It is given that MergeSort sorts L in time $\Theta(n \log n)$.

Algorithm II. FrequencySort(A) where $A[1, \ldots, n]$ is an n-element array with at most $\log n$ distinct elements:

Visit each element in A to determine the maximum value m.

Initialize an array $B[1, \ldots, m]$ where $B[i] = 0$ for all $1 \leq i \leq m$.

Initialize also a list of integers L with maximum length $\log n$.

For each element $a \in A$:

- if $B[a] = 0$, then append a to L,
- assign $B[a] \leftarrow B[a] + 1$.

Perform MergeSort(L).

Let $i \leftarrow 1$.

For each element $\ell \in L$:

- for $j = 1$ to $B[\ell]$:
  - assign $A[i] \leftarrow \ell$,
  - assign $i \leftarrow i + 1$.

Proposition I.

Claim. FrequencySort sorts $A[1, \ldots, n]$.

Proof. On each iteration of the loop, $B[a]$ is incremented at the end of the loop body. This means that at the beginning of the loop body if $B[a] = 0$, a is unique thus far. So, a is appended to L. Thus, L contains the unique elements in a. By construction of A, the maximum length of L is $\log n$. When the loop completes, $B[a]$ contains the number of times a is found in A.

MergeSort(L) sorts L, the unique elements in A.

The second loop visits each element in the now-sorted list L and uses the frequencies stored in B to reconstruct the original array A. For each $\ell \in L$, the inner loop copies the exact number of duplicate elements of ℓ that exist in A, which is given by $B[\ell]$. The index variable i is incremented to maintain the position in A.

Since each element in A was accounted for, either by adding an element to L or by incrementing an existing value in B, we know that by iterating through the $\log n$ values of L and reconstructing A in this way, we visit all n elements in A. Since each $\ell = \ell$ and since L is sorted, the elements are copied into A in sorted order.

Consequently, A contains its original elements now in sorted order at the end of the algorithm. □

Proposition II.

Claim. FrequencySort executes in $O(n)$ time.

Proof. First, the algorithm determines the maximum value in A, which is an $O(n)$ process.

Initializing B and L is an $O(1)$ process by Axioms I and II.

For each of the n elements in A, comparing $B[a]$ to 0, appending a to L, and incrementing $B[a]$ are all $O(1)$ processes. Thus, the first loop is performed in $O(n)$ time.

Since L contains only the unique values in A, the maximum length of L is $\log n$.

From Algorithm I, performing MergeSort on L is an $O(\log n \log\log n)$ process.

Initializing i is an $O(1)$ process.

For each of the $\log n$ elements in L, the number of inner loop iterations is given by the corresponding frequency in B. By construction, the sum of all frequencies is n. For the innermost loop body, the total number of iterations is n. Assigning $A[i]$ and incrementing i are $O(1)$ processes. Therefore, the second loop is performed in $O(n)$ time.

We conclude that the asymptotic running time of FrequencySort is $O(n)+O(\log n \log\log n)+O(1) = O(n)$. □

Question 2: Rod cutting with cost

Recall the Rod Cutting problem from the lecture: Suppose we have a rod of length n inches and we also have an array of prices P, where $P[i]$ denotes the selling price (\$) of a piece that is i inches long. Using dynamic programming (DP), one can efficiently find the maximal revenue obtained by cutting the rod and selling the individual pieces.

Suppose now we have to pay a cost of \$1 per cut. Define the *profit* we make as the revenue minus the total cost of cutting. We want an algorithm that finds a way to cut the rod that maximizes our **profit**. For example, if we have $P = (2, 5, 6, 7)$ and $n = 4$, then cutting it into two rods of length 2 yields optimal profit $9 = P[2] + P[2] - 1$.

1. Describe recursive formula $R(i)$ for the maximal profit when having a rod of length i. (You can use the values of P inside your formula.) Do not forget the base case and justify the correctness of your formula.

Solution: *Definitions.* Consider the recursive function $R(i)$ for integers $i > 0$:

$$R(i) = \begin{cases} 0, & i = 0; \\ \max\left(P[i], \max\limits_{1 \le j < i} \left(P[j] + R[i-j] - 1\right)\right) & i > 0. \end{cases}$$

Claim. For integers $i \geq 0$, the recursive function $R(i)$ computes the maximal profit, in dollars, for a rod with length i inches.

Proof. We can demonstrate the claim by induction on i.

Basis. Consider $i = 0$. A 0-inch rod cannot be sold or cut, so the maximum profit is \$0. In this case, $R(0) = 0$, so the claim holds in the base case.

Hypothesis. Consider $0 < i \leq k$. Assume that $R(k)$ computes the maximal profit, in dollars, for a rod with length k inches.

Inductive step. Consider $i = k + 1$. The maximum profit for a $(k + 1)$-inch rod is *either* the revenue obtained by selling that rod without cutting it, or the maximum profit obtainable by cutting it, whichever is greater.

- Consider the case when we do not cut the rod. Then the maximum profit is $P[k + 1]$.
- Consider the cases when we cut the rod. The rod can be cut after each 1-inch segment from segment 1 to segment $(k + 1) - 1$.

  For each of these positions j, two rods are produced: one of length j, and another of length $(k+1)-j$. The maximum profit in this case is the revenue obtained from selling a j-inch rod, plus the maximum profit obtainable from the remaining portion of the rod, less the \$1 cost of cutting the rod.

  By the strong induction hypothesis, $R[(k+1)-j]$ is the maximum profit for the remaining $((k + 1) - j)$-inch rod.

  The rod should be cut at the position with the largest total profit.

  In this case, $R(k + 1) = \max\limits_{1 \le j < k+1} (P[j] + R[(k + 1) - j] - 1)$, so the claim holds.

In all cases, the claim holds.

Hence, by the principle of mathematical induction, $R(i)$ computes the maximal profit, in dollars, for a rod with length i inches. □

2. Implement your strategy using top-down DP with memoization.

Solution:

Environment. Let MEMO be a global array which stores the values of the subproblems. Initialize MEMO with all 0 values.

Algorithm. RODCUT(i, P) for integer $i \geq 0$ and array of prices P. RODCUT returns the maximal profit in dollars for a rod with length i inches:

- if $i = 0$, then return 0,
- if MEMO$[i] \neq 0$, then return MEMO$[i]$.
- otherwise,
  - let $a \leftarrow P[i]$,
  - for $j = 1$ to $i - 1$, assign $a \leftarrow \max(a, P[j] + \text{RODCUT}(i - j, P) - 1)$,
  - assign MEMO$[i] \leftarrow a$,
  - return a.

In the following, we want to implement a solution using *bottom-up* DP, i.e., using an array to store the solutions rather than recursion. For your DP algorithm, use the name MAXPROFIT for the (potentially multi-dimensional) array which stores values of the subproblems.

3. Describe in words what MAXPROFIT should be and state its dimension.

Solution:

Environment. Let MAXPROFIT$[0, \ldots, i]$ be a one-dimensional $(i + 1)$-element array which stores the values of the subproblems.

MAXPROFIT needs sufficient capacity to contain all subproblems for a rod of length i, from the trivial rod of length 0 to the solution of length i.

4. Implement your solution using bottom-up dynamic programming.

Solution:

Algorithm. RodCut2(i, P) for integer $i \geq 0$ and array of prices P. RodCut2 returns the maximal profit in dollars for a rod with length i inches:

Initialize array MaxProfit$[0, \ldots, i]$.

Assign MaxProfit$[0] \leftarrow 0$.

For $k = 1$ to i:

- $a \leftarrow P[k]$,
- for $j = 1$ to $k - 1$, assign $a \leftarrow \max(a, P[j] + \text{MaxProfit}(k - j) - 1)$,
- MaxProfit$[k] \leftarrow a$.

Return MaxProfit$[i]$.

5. Justify the run time of your algorithm to compute MaxProfit as a big-Θ expression.

Solution:

Axiom. Assume that an arbitrarily-sized array can be initialized with all-0 values in constant time.

Proposition. *Claim.* RodCut2 executes in $\Theta(n^2)$ time.

Proof. First, the algorithm initializes MaxProfit and assigns MaxProfit$[0]$, which are $\Theta(1)$ by the Axiom.

The outer for loop body executes i-many times. For each of the i iterations, assigning a and assigning MaxProfit$[k]$ are $\Theta(1)$.

The inner loop body executes $(k-1)$-many times, which is itself dependent on i. For each inner loop iteration, assigning a and determining the maximum of two arguments are $\Theta(1)$ operations.

Since the inner and outer loop both depend on i, and the innermost loop body only performs constant operations, the overall complexity of the loop is $\Theta(i^2)$.

We conclude that the asymptotic running time of RodCut2 is $\Theta(i^2) + \Theta(1) = \Theta(i^2)$. $\square$

Question 3: Subset sum

Let $A = [a_1, \ldots, a_n]$ be an array of n natural numbers. Given a number $t \in \mathbb{N}$, we say t is a *subset sum* of A if there is a *subset of indices* $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} a_i = t$. For example, if $A = [1, 3, 5, 7]$ then 12 is a subset sum of A since $5 + 7 = 12$, but 2 is not. (Note that each value can only be included in the sum at most once.)

We want to design an algorithm that determines whether a given number $t \in \mathbb{N}$ is a subset sum of A. Specifically, when given $A = [a_1, \ldots, a_n]$ and $t \in \mathbb{N}$ as input, the algorithm should output 1 if t is a subset sum of A, and 0 otherwise. We will use (bottom-up) dynamic programming with a two-dimensional array to solve this problem. The subproblem in your DP algorithm should be SubsetSum, which is defined as

$$\textsc{SubsetSum}[i][t] = \begin{cases} 1 & \text{if } t \text{ is a subset sum of the first } i \text{ elements } A_{1:i} = [a_1, \ldots, a_i], \\ 0 & \text{otherwise.} \end{cases}$$

Also, let us define $M = \sum_{i=1}^{n} a_i$, which is the sum of all numbers in A. The quantity M may appear in the size of the DP array and the run-time of your algorithm.

1. Suppose $A = [2, 1, 3]$. Fill out the following table for A. Note that we define the sum over the empty set to be 0. The pre-filled entries in the top left corner correspond to $\textsc{SubsetSum}[0][0] = 1$ and $\textsc{SubsetSum}[0][1] = 0$.

i \ t	0	1	2	3	4	5	6
0	1	0	0	0	0	0	0
1	1	0	1	0	0	0	0
2	1	1	1	1	0	0	0
3	1	1	1	1	1	1	1

Table 1: Fill in the entries for $\textsc{SubsetSum}[i][t]$.

2. In general, what is the size of the DP array containing values for SubsetSum?

> **Solution:** The maximum value for i is n and the minimum value for i is 0. Similarly, the maximum value for t is $M = \sum_{j=1}^{n} a_j$ (when $i = n$), and the minimum value for t is 0 (when $i = 0$). Therefore, the horizontal t-dimension must be $M + 1$ and the vertical i-dimension must be $n + 1$. Thus, the size of the dynamic programming array containing values for SubsetSum is $(n + 1) \times (M + 1)$.

3. State the base cases for $i = 0$, i.e., state all the values for $i = 0$.

> **Solution:** Consider $i = 0$. When $i = 0$, the first i elements $A_{1:i} = [a_1, \ldots, a_i]$ is an empty subset of A. The only subset sum that can be made is 0. Thus, $\textsc{SubsetSum}[0][0] = 1$ and $\textsc{SubsetSum}[0][t] = 0$ for all $0 < t \leq M$.

4. Give and justify the recurrence SubsetSum should satisfy.

Solution: *Claim.* Let R denote the recurrence for SubsetSum. Then

$$R(i,t) = \begin{cases} 1, & i = 0 \text{ and } t = 0, \\ 0, & i = 0, \text{ and } t \neq 0, \\ R(i-1,t), & i \neq 0, t \neq 0, \text{ and } t - a_i < 0, \\ R(i-1,t) \vee R(i-1,t-a_i), & \text{otherwise.} \end{cases}$$

Proof. We can demonstrate the claim for all $0 \leq t \leq M$ using induction on $0 \leq i \leq n$.

Basis. Consider $i = 0$. When $i = 0$, the first i elements $A_{1:i} = [a_1, \ldots, a_i]$ is an empty subset of A. The only subset sum that can be made is 0. Thus, $R(0,0) = 1$ and $R(0,t) = 0$ for all $0 < t \leq M$.

Thus for all $0 \leq t \leq M$, the claim holds in the base case.

Hypothesis. Consider $0 < i \leq k < n$. Assume that R is the recurrence for SubsetSum for all $0 \leq t \leq M$.

Inductive step. Consider $i = k + 1$. Let $0 \leq t \leq M$. We want to determine if t is a subset sum of $[a_1, \ldots, a_{k+1}]$. We can either include a_{k+1} in this subset or not include a_{k+1} in this subset.

- Suppose a_{k+1} is not included in the subset. Then the result should be 1 if t is a subset sum of $[a_1, \ldots, a_k]$, and 0 otherwise. By the inductive hypothesis, $R(k,t)$ provides this result.
- Suppose instead a_{k+1} is included in the subset. If we were able to produce the sum $t - a_{k+1}$ from the first k elements, then the inclusion of a_{k+1} brings the subset sum to $(t - a_{k+1}) + a_{k+1} = t$, so it is possible to achieve a subset sum of t using the first $k + 1$ elements.

  If $t - a_{k+1} < 0$, then it is certainly impossible to produce a negative sum using only the natural numbers in A, so this case does not apply.

  If, however, $t - a_{k+1} \geq 0$, then by the strong induction hypothesis, $R(k, t - a_{k+1})$ gives 1 if we were able to procue the sum $t - a_{k+1}$ from the first k elements, and 0 otherwise.

  Thus $R(k, t - a_{k+1})$ gives the desired result.

If the subset sum t is achievable by including a_{k+1} or by not including a_{k+1}, then the result should be 1, and 0 otherwise. The logical *or* of the two results above provides this behavior.

This completes the inductive step for all $0 \leq t \leq M$.

Hence, by the principle of mathematical induction, we have R as the correct recurrence for SubsetSum for all $0 \leq t \leq M$, for all $0 \leq i \leq n$. □

5. For a bottom-up DP implementation, in which order do you have to fill the two-dimensional table for your algorithm to work? Should you fill it row by row or column by column? And within each row/column, does the order matter?

Solution: For a bottom-up dynamic programming implementation of this SubsetSum algorithm, we should fill the two-dimensional table from top-to-bottom and from left-to-right. However, it does not matter whether we fill it in row-major or column-major order.

Each cell in the table depends upon a cell in the previous (above) row. Thus, if we were to fill the table from bottom-to-top, we would be missing information required for the computation. That is to say, there would be cells which depend upon uninitialized cells.

Some cells in the table depend upon cells in a prior (leftward) column of the previous row. If we fill the table in row-major order, the entire previous column will be filled, meaning that there is no missing information. Similarly, if we fill the table in column-major order, all rows including the previous (above) row will be filled for all prior (leftward) columns, meaning no required information is missing in this case either.

However, if we were to fill the table from right-to-left, there would be cells which depend upon uninitialized cells in previous (leftward) columns. This required infomration would be missing.

We conclude that we must fill this table from top-to-bottom and from left-to-right.